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.

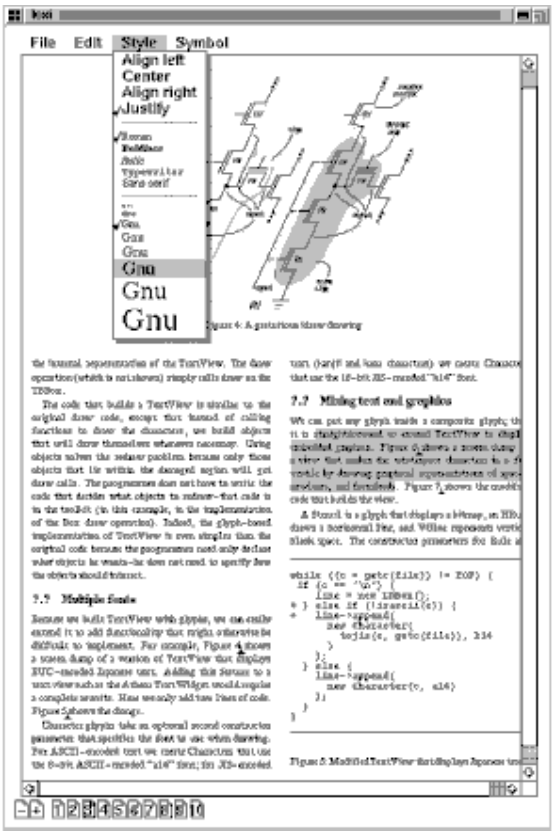


Figure 2.1: Lexi's user interface

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).

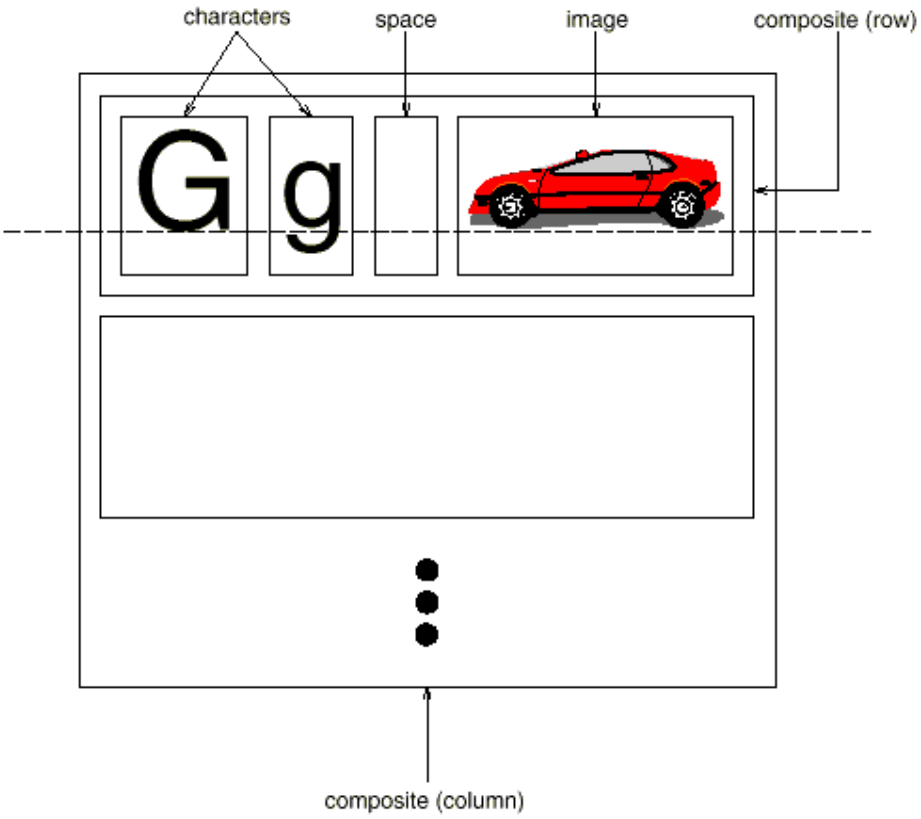


Figure 2.2: Recursive composition of text and graphics

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.

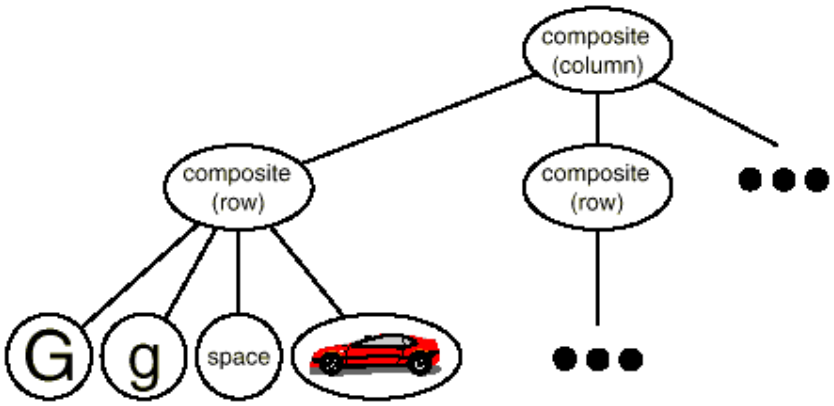


Figure 2.3: Object structure for recursive composition oftext and graphics

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]

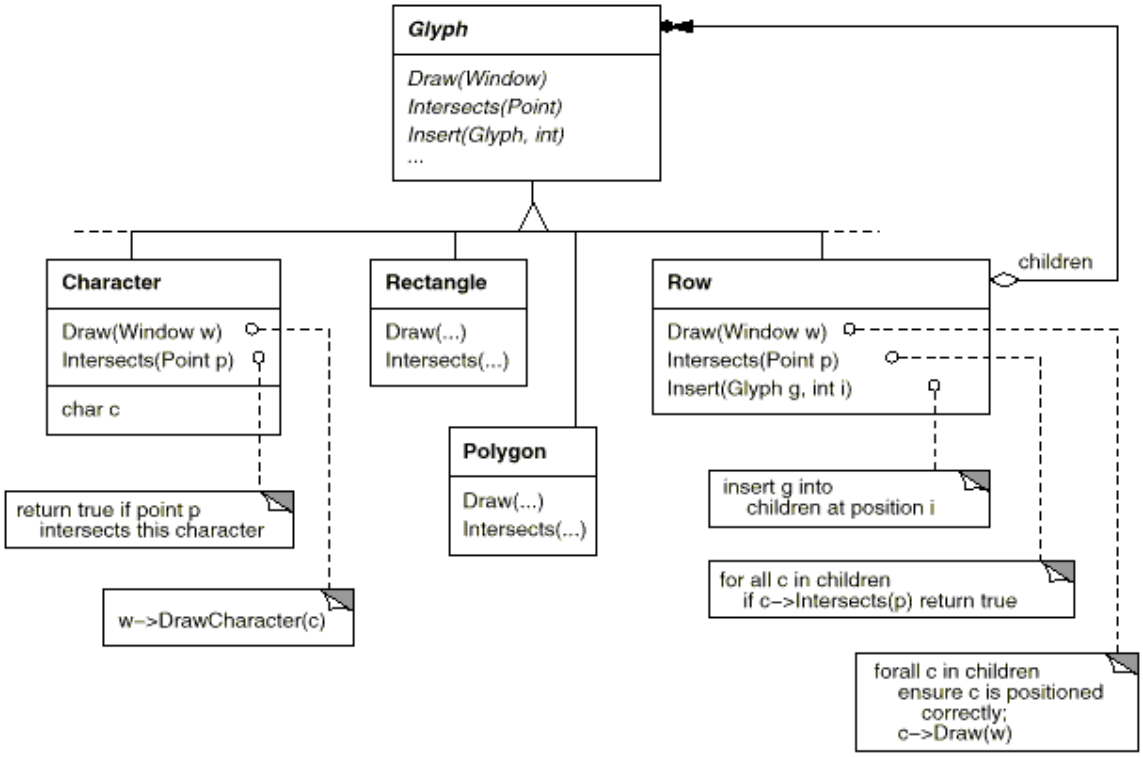


Figure 2.4: Partial Glyph class hierarchy

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

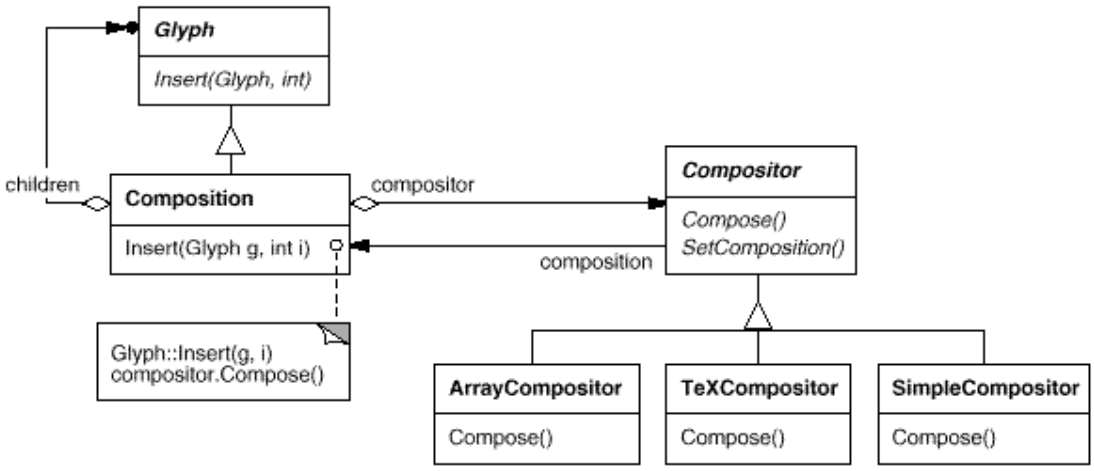


Figure 2.5: Composition and Compositor class relationships

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.

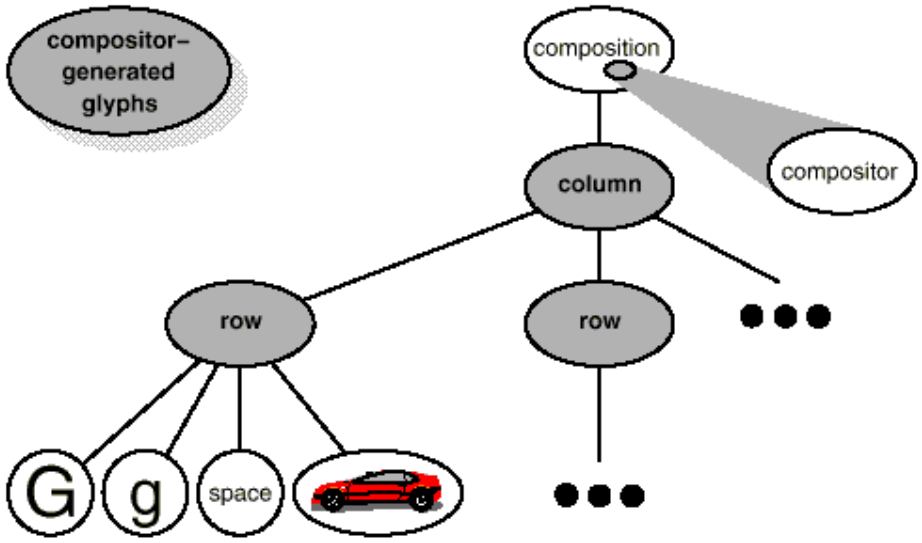


Figure 2.6: Object structure reflectingcompositor-directed linebreaking

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);
}
```

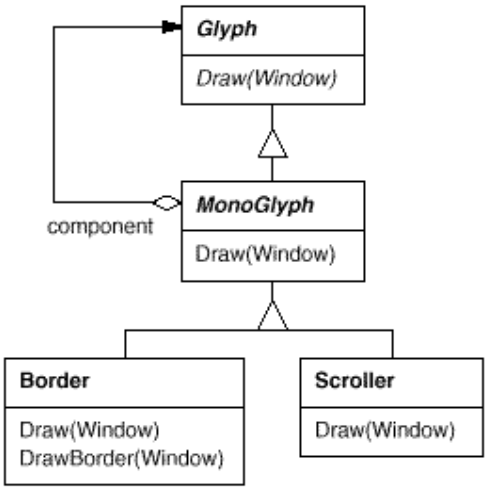


Figure 2.7: MonoGlyph class relationships

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.

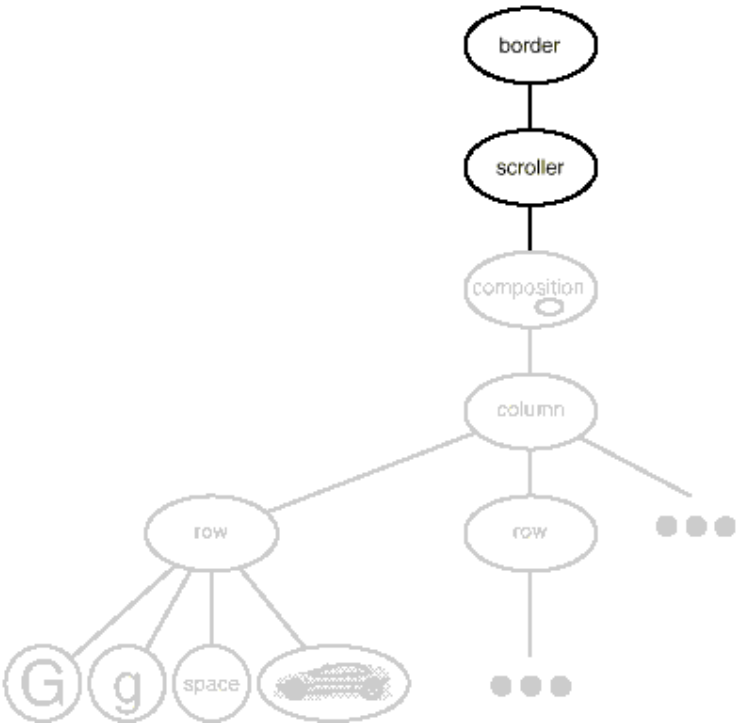


Figure 2.8: Embellished object structure

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.

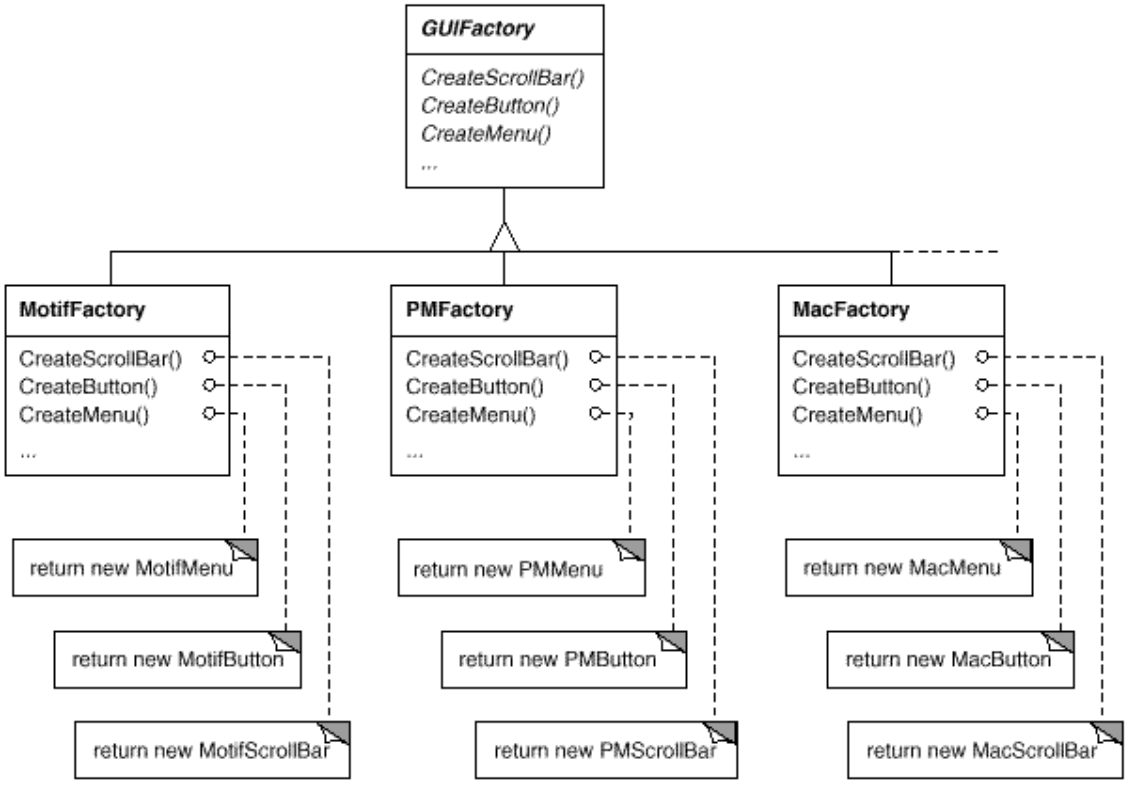


Figure 2.9: GUIFactory class hierarchy

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.

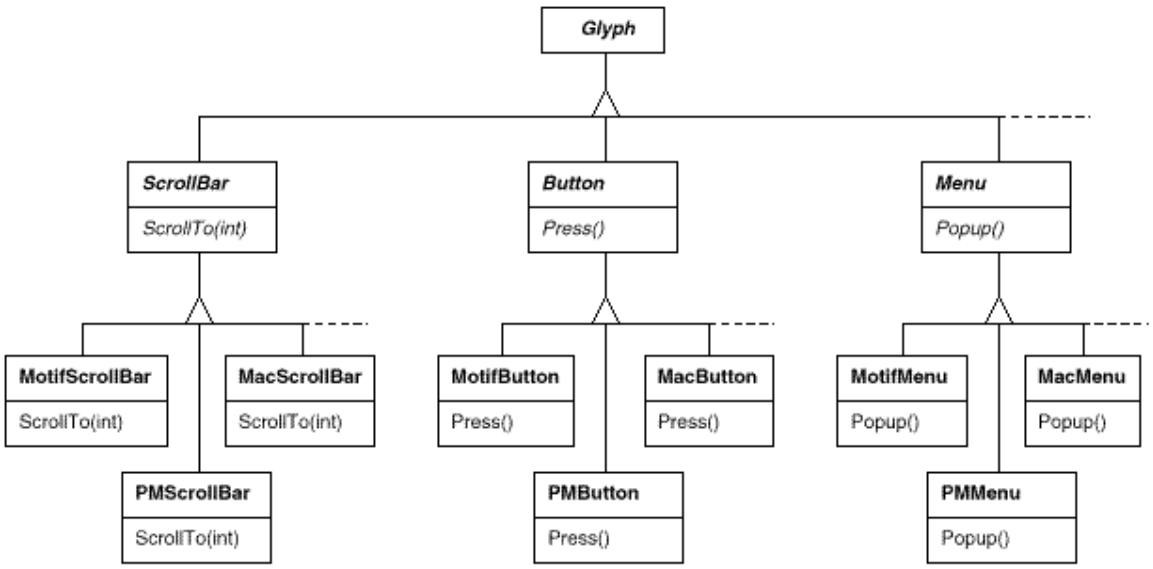


Figure 2.10: Abstract product classes and concrete subclasses

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.

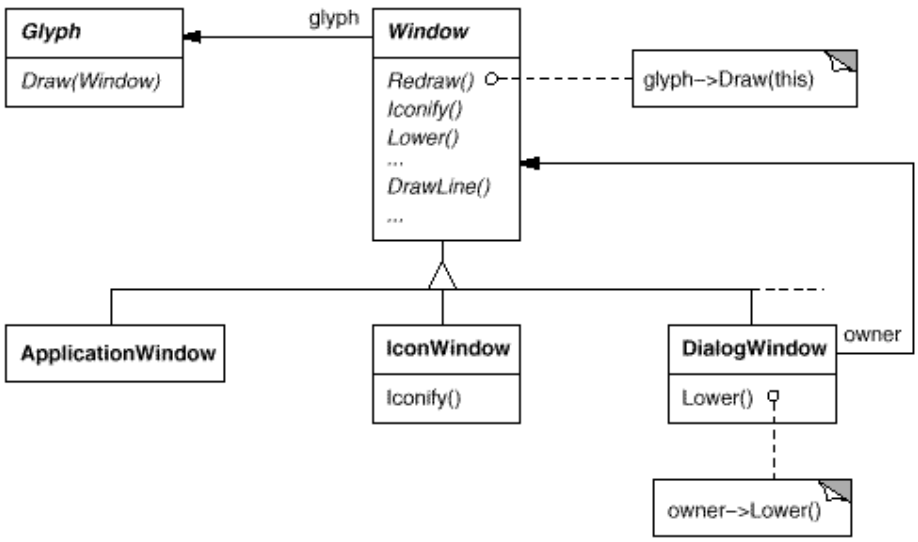
Neither extreme is a viable solution, so our design will fallsomewhere between the two. The Window class will provide a convenientinterface that supports the most popular windowing features. BecauseLexi will deal with this class directly, the Window class must alsosupport the things Lexi knows about, 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?

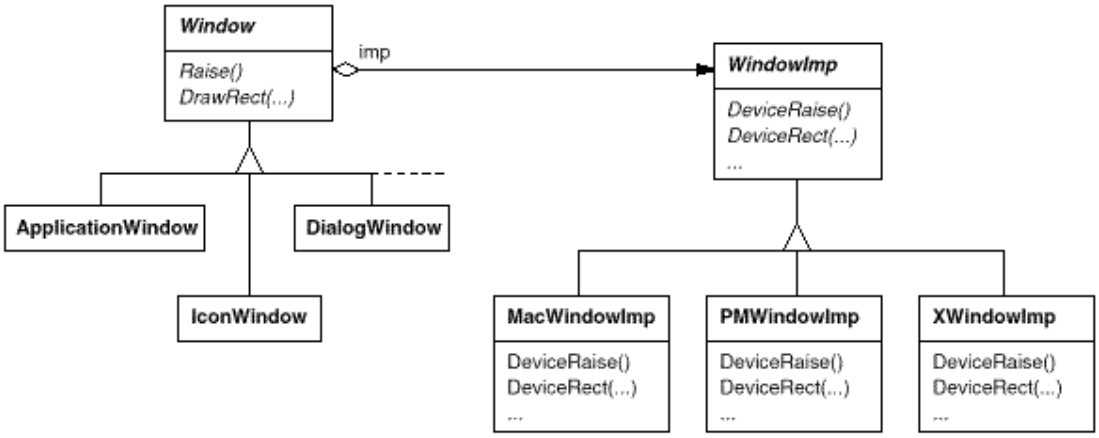
One approach is to implement multiple versions of the Window class andits subclasses, one version for each windowing platform. We'd have tochoose the version to use when we build Lexi for a given platform.But imagine the maintenance headaches we'd have keeping track ofmultiple classes, all named "Window" 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.

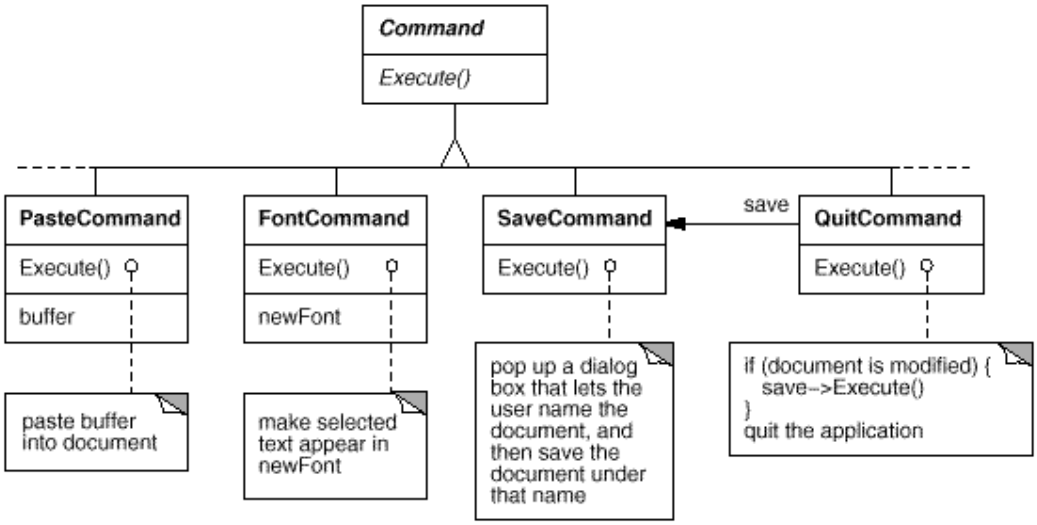


Figure 2.11: Partial Command class hierarchy

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.

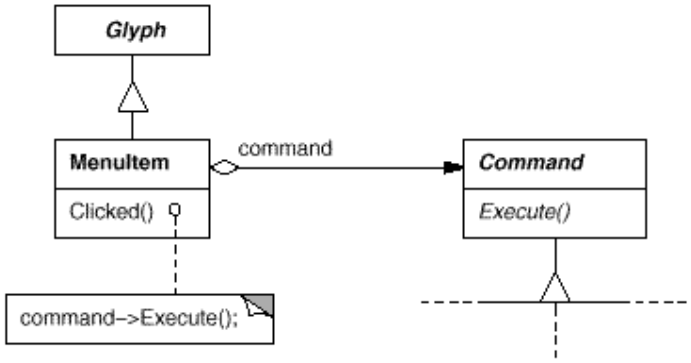


Figure 2.12: MenuItem-Command relationship

Undoability

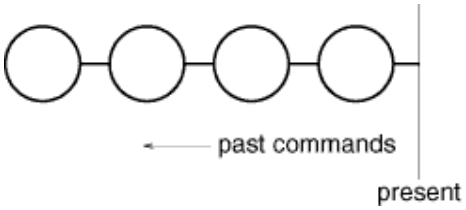
Undo/redo is an important capability in interactive applications. Toundo and redo commands, we add an Unexecute operation to Command'sinterface. Unexecute reverses the effects of a preceding Executeoperation using whatever undo information Execute stored. In the caseof a FontCommand, for example, the 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.

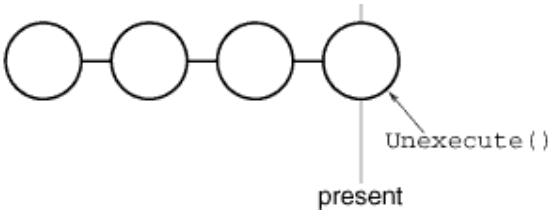
Command History

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:

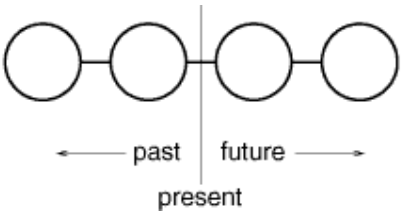


Each circle represents a Command object. In this case the user hasissued four commands. The leftmost command was issued first, followedby the second-leftmost, and so on until the most recently issuedcommand, which is rightmost. The line marked "present" keeps trackof the most recently executed (and unexecuted)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

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

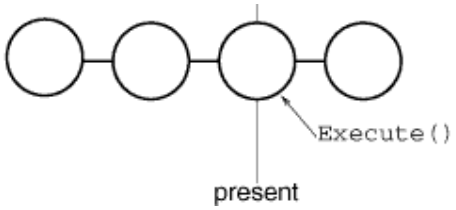


After unexecuting the command, we move the "present" line onecommand to the left. If the user chooses undo again, the next-mostrecently issued command will be undone in the same way, and we're leftin the state depicted here:

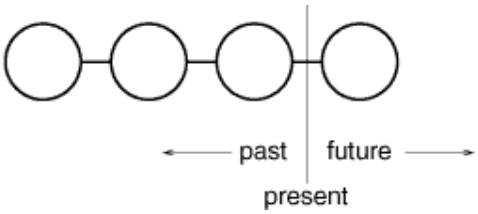


You can see that by simply repeating this procedure we get multiplelevels of undo. The number of levels is limited only by the length ofthe command history.

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.

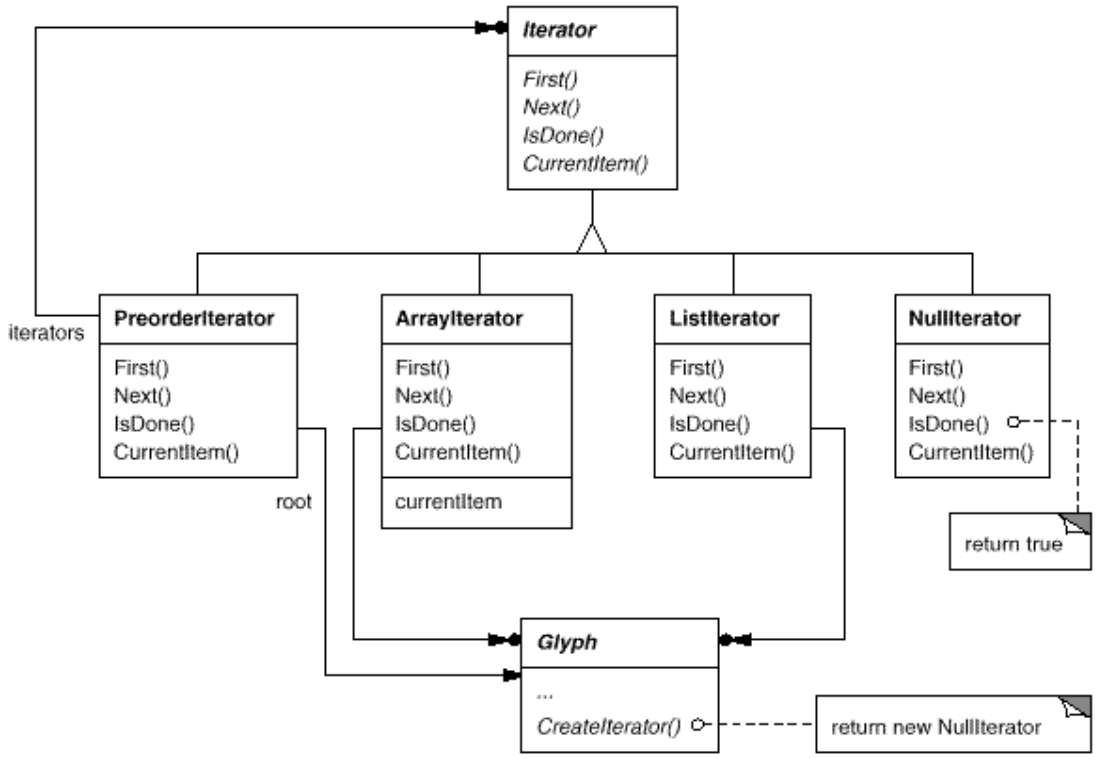


Figure 2.13: Iterator class and subclasses

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.

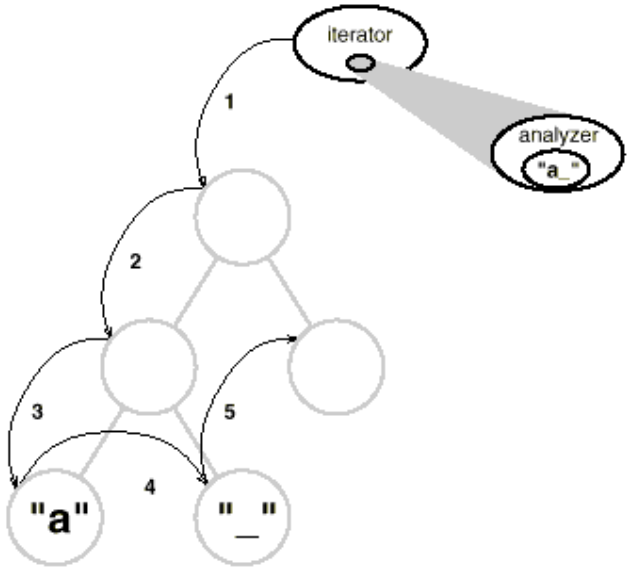
Therefore a given analysis must be able to distinguish different kinds ofglyphs. An obvious approach is to put the analytical capability into theglyph classes themselves. For each analysis we can add one or moreabstract operations to the Glyph class and have subclasses implementthem in accordance with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

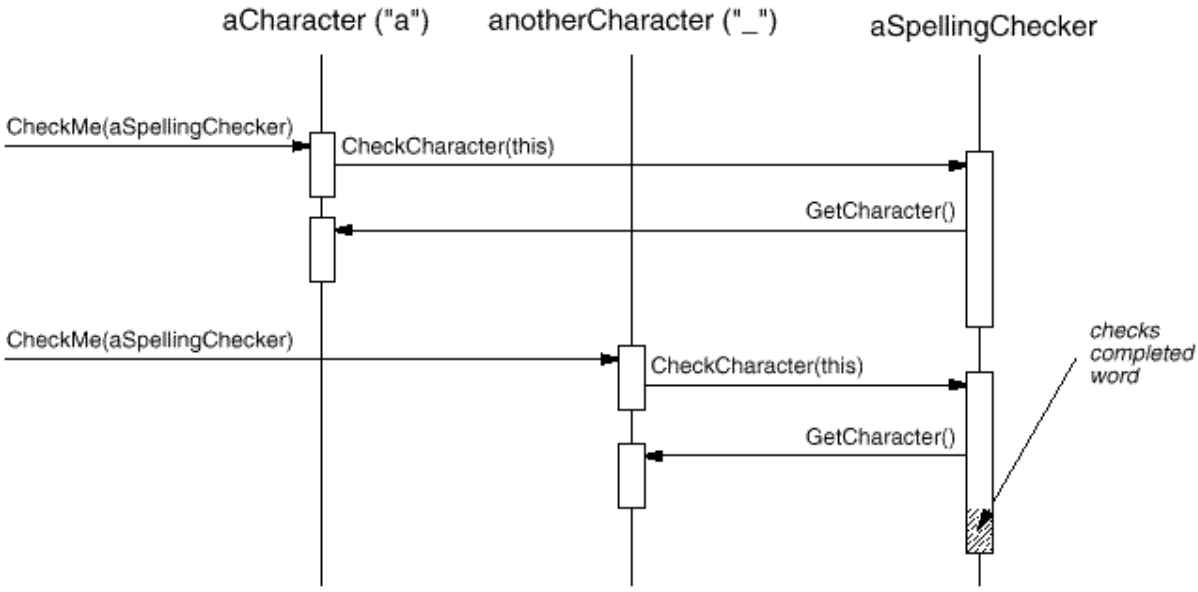
Notice we've defined a special GetCharCode operation onjust the Character class. The spelling checker can deal withsubclass-specific operations without resorting to type tests orcasts—it lets us treat objects specially.

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
};
```

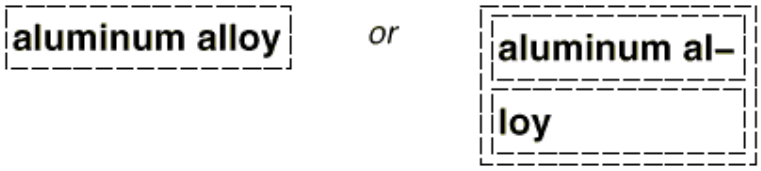
Concrete subclasses of Visitor perform different analyses.For example, we could have a SpellingCheckingVisitorsubclass for checking spelling, and a HyphenationVisitorsubclass for hyphenation. SpellingCheckingVisitor wouldbe implemented exactly as we implemented SpellingCheckerabove, except the operation 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,

5. Bridge (171) to allow multiplewindowing platforms,
6. Command (263) for undoable useroperations,
7. Iterator (289) for accessing andtraversing object structures, and
8. Visitor (366) for allowing anopen-ended number of analytical capabilities without complicatingthe document structure's implementation.

None of these design issues is limited to document editingapplications like Lexi. Indeed, most nontrivial applications willhave occasion to use many of these patterns, though perhaps to dodifferent things. A financial analysis application might useComposite to define investment portfolios made up of subportfolios andaccounts of different sorts. A compiler might use the Strategypattern to allow different register allocation schemes for differenttarget machines. Applications with a graphical user interface willprobably apply at least Decorator and Command just as we have here.

While we've covered several major problems in Lexi's design, thereare lots of others we haven't discussed. Then again, this bookdescribes more than just the eight patterns we've used here. So asyou study the remaining patterns, think about how you might use eachone in Lexi. Or better yet, think about using them in your owndesigns!

---

[1]Lexi's design is based on Doc, a text editingapplication developed byCalder [CL92].

[2]Authors often view the document in terms of its*logical* structure as well, that is, in terms of sentences,paragraphs, sections, subsections, and chapters. To keep thisexample simple, our internal representation won't store informationabout the logical structure explicitly. But the design solution wedescribe works equally well for representing such information.

[3]Calder was the first to use the term "glyph" in thiscontext [CL90].Most contemporary document editors don't use an object for everycharacter, presumably for efficiency reasons. Calder demonstratedthat this approach is feasible in his thesis [Cal93]. Our glyphs are lesssophisticated than his in that we have restricted ours to stricthierarchies for simplicity. Calder's glyphs can be shared to reducestorage costs, thereby forming directed-acyclic graph structures.We can apply the Flyweight (218)pattern to get the same effect, but we'll leave that as an exercisefor the reader.

[4]The interface we describehere is purposely minimal to keep the discussion simple. A completeinterface would include operations for managing graphical

attributessuch as color, font, and coordinate transformations, plus operationsfor more sophisticated child management.

[5]An integer index is probably not the best way to specifya glyph's children, depending on the data structure the glyph uses.If it stores its children in a linked list, then a pointer into thelist would be more efficient. We'll see a better solution to theindexing problem in Section 2.8, when we discuss document analysis.

[6]The user will have even more to say about thedocument's *logical* structure—the sentences, paragraphs,sections, chapters, and so forth. The *physical* structure is lessinteresting by comparison. Most people don't care where the linebreaksin a paragraph occur as long as the paragraph is formatted properly.The same is true for formatting columns and pages. Thus users end upspecifying only high-level constraints on the physical structure,leaving Lexi to do the hard work of satisfying them.

[7]The compositor must get the character codes ofCharacter glyphs in order to compute the linebreaks. In Section 2.8 we'll see howto get this information polymorphically without adding acharacter-specific operation to the Glyph interface.

[8]That is, redoing an operation that was just undone.

[9]Conceptually, the client is Lexi's user, but inreality it's another object (such as an event dispatcher) thatmanages inputs from the user.

[10]We could use function overloading to give each of these memberfunctions the same name, since their parameters already differentiatethem. We've given them different names here to emphasize theirdifferences, especially when they're called.

[11]IsMisspelled implements the spellingalgorithm, which we won't detail here because we've made itindependent of Lexi's design. We can support different algorithmsby subclassing SpellingChecker; alternatively, we canapply the Strategy (349) pattern (as we did for formatting in Section 2.3) to supportdifferent spelling checking algorithms.

[12]"Visit" is just a slightly moregeneral term for "analyze." It foreshadows the terminology we use inthe design pattern we're leading to.

Design Pattern Catalog

3. Creational Patterns

Creational design patterns abstract the instantiation process.They help make a system independent of how its objects are created,composed, and represented. A class creational pattern uses inheritanceto vary the class that's instantiated, whereas an object creationalpattern will delegate instantiation to another object.

Creational patterns become important as systems evolve to depend moreon object composition than class inheritance. As that happens,emphasis shifts away from hard-coding a fixed set of behaviors towarddefining a smaller set of fundamental behaviors that can be composedinto any number of more complex ones. Thus creating objects withparticular behaviors requires more than simply instantiating a class.

There are two recurring themes in these patterns. First, they allencapsulate knowledge about which concrete classes the system uses.Second, they hide how instances of these classes are created and puttogether. All the system at large knows about the objects is theirinterfaces as defined by abstract classes. Consequently, thecreational patterns give you a lot of flexibility in *what* getscreated, *who* creates it, *how* it gets created, and *when*. They let you configure a system with "product" objects thatvary widely in structure and functionality. Configuration can bestatic (that is, specified at compile-time) or dynamic (atrun-time).

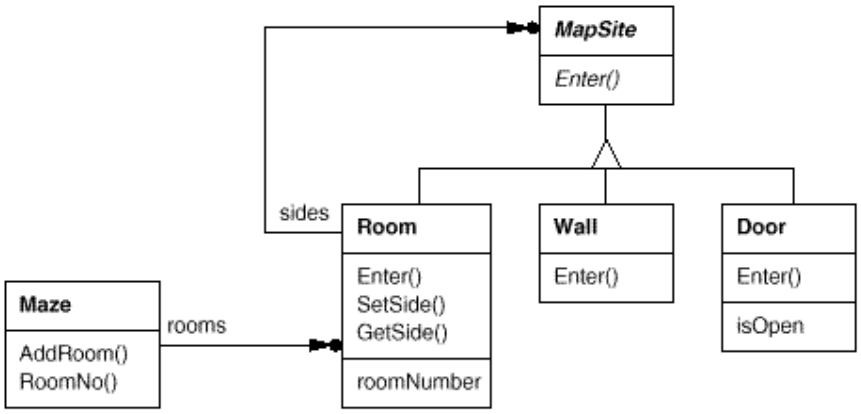
Sometimes creational patterns are competitors. For example,there are cases when either Prototype (133)or Abstract Factory (99) couldbe used profitably. At other times they are complementary: Builder (110) can use one of the otherpatterns to implement which components get built. Prototype (133) can use Singleton (144) in its implementation.

Because the creational patterns are closely related, we'll study allfive of them together to highlight their similarities and differences.We'll also use a common example—building a maze for a computergame—to illustrate their implementations. The maze and the game willvary slightly from pattern to pattern. Sometimes the game will besimply to find your way out of a maze; in that case the player willprobably only have a local view of the maze. Sometimes mazes containproblems to solve and dangers to overcome, and these games may providea map of the part of the maze that has been explored.

We'll ignore many details of what can be in a maze and whether a mazegame has a single or multiple players. Instead, we'll just focus onhow mazes get created. We define a maze as a set of rooms. A roomknows its neighbors; possible neighbors are another room, a wall, or adoor to another room.

The classes Room, Door, and Walldefine the components of the maze used in all our examples. We defineonly the parts of these classes that are important for creating amaze. We'll ignore players, operations for displaying and wanderingaround in a maze, and other important functionality that isn'trelevant to building the maze.

The following diagram shows the relationships between these classes:



Each room has four sides. We use an enumeration Direction inC++ implementations to specify the north, south, east, and west sides ofa room:

```
enum Direction {North, South, East, West};
```

The Smalltalk implementations use corresponding symbols to representthese directions.

The class MapSite is the common abstract class for all thecomponents of a maze. To simplify the example, MapSite definesonly one operation, Enter. Its meaning depends on what you'reentering. If you enter a room, then your location changes. If you try toenter a door, then one of two things happen: If the door is open, you gointo the next room. If the door is closed, then you hurt your nose.

```
class MapSite {
public:
virtual void Enter() = 0;
};
```

Enter provides a simple basis for more sophisticated gameoperations. For example, if you are in a room and say "Go East," thegame can simply determine which MapSite is immediately to theeast and then call Enter on it. The subclass-specificEnter operation will figure out whether your location changedor your nose got hurt. In a real game, Enter could take theplayer object that's moving about as an argument.

Room is the concrete subclass of MapSite thatdefines the key relationships between components in the maze. Itmaintains references to other MapSite objects and stores aroom number. The number will identify rooms in the maze.

```
class Room : public MapSite {
public:
Room(int roomNo);

MapSite* GetSide(Direction) const;
void SetSide(Direction, MapSite*);

virtual void Enter();

private:
MapSite* _sides[4];
int _roomNumber;
};
```

The following classes represent the wall or door that occurs on eachside of a room.

```
class Wall : public MapSite {
public:
Wall();
virtual void Enter();
};

class Door : public MapSite {
public:
Door(Room* = 0, Room* = 0);

virtual void Enter();
Room* OtherSideFrom(Room*);

private:
Room* _room1;
Room* _room2;
bool _isOpen;
};
```

We need to know about more than just the parts of a maze. We'll alsodefine a Maze class to represent a collection of rooms.Maze can also find a particular room given a room numberusing its RoomNo operation.

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

void AddRoom(Room*);
Room* RoomNo(int) const;
private:
// ...
};
```

RoomNo could do a look-up using a linear search, a hash table,or even a simple array. But we won't worry about such details here.Instead, we'll focus on how to specify the components of a maze object.

Another class we define is MazeGame, which creates the maze.One straightforward way to create a maze is with a series of operationsthat add components to a maze and then interconnect them. Forexample, the following member function will create a maze consistingof two rooms with a door between them:

```
Maze* MazeGame::CreateMaze () {
Maze* aMaze = new Maze;
Room* r1 = new Room(1);
Room* r2 = new Room(2);
Door* theDoor = new Door(r1, r2);

aMaze->AddRoom(r1);
aMaze->AddRoom(r2);

r1->SetSide(North, new Wall);
r1->SetSide(East, theDoor);
r1->SetSide(South, new Wall);
r1->SetSide(West, new Wall);

r2->SetSide(North, new Wall);
r2->SetSide(East, new Wall);
r2->SetSide(South, new Wall);
r2->SetSide(West, theDoor);

return aMaze;
}
```

This function is pretty complicated, considering that all it does is createa maze with two rooms. There are obvious ways to make it simpler. Forexample, the Room constructor could initialize the sideswith walls ahead of time. But that just

moves the code somewhere else.The real problem with this member function isn't its size but its *inflexibility*. It hard-codes the maze layout. Changing the layoutmeans changing this member function, either by overriding it—whichmeans reimplementing the whole thing—or by changing parts ofit—which is error-prone and doesn't promote reuse.

The creational patterns show how to make this design more *flexible*, not necessarily smaller. In particular, they will make iteasy to change the classes that define the components of a maze.

Suppose you wanted to reuse an existing maze layout for a new gamecontaining (of all things) enchanted mazes. The enchanted maze game hasnew kinds of components, like DoorNeedingSpell, a door thatcan be locked and opened subsequently only with a spell; andEnchantedRoom, a room that can have unconventional items init, like magic keys or spells. How can you change CreateMazeeasily so that it creates mazes with these new classes of objects?

In this case, the biggest barrier to change lies in hard-coding theclasses that get instantiated. The creational patterns providedifferent ways to remove explicit references to concrete classesfrom code that needs to instantiate them:

- If CreateMaze calls virtual functions instead of constructorcalls to create the rooms, walls, and doors it requires, then you canchange the classes that get instantiated by making a subclass ofMazeGame and redefining those virtual functions. This approachis an example of the Factory Method (121) pattern.
- If CreateMaze is passed an object as a parameter to use tocreate rooms, walls, and doors, then you can change the classes ofrooms, walls, and doors by passing a different parameter. This is anexample of the Abstract Factory (99) pattern.
- If CreateMaze is passed an object that can create a new mazein its entirety using operations for adding rooms, doors, and walls tothe maze it builds, then you can use inheritance to change parts ofthe maze or the way the maze is built. This is an example of the Builder (110) pattern.
- If CreateMaze is parameterized by various prototypical room,door, and wall objects, which it then copies and adds to the maze,then you can change the maze's composition by replacing theseprototypical objects with different ones. This is an example of the Prototype (133) pattern.

The remaining creational pattern, Singleton (144), canensure there's only one maze per game and that all game objects haveready access to it—without resorting to global variables orfunctions. Singleton also makes it easy to extend or replace the mazewithout touching existing code.

Abstract Factory

Intent

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

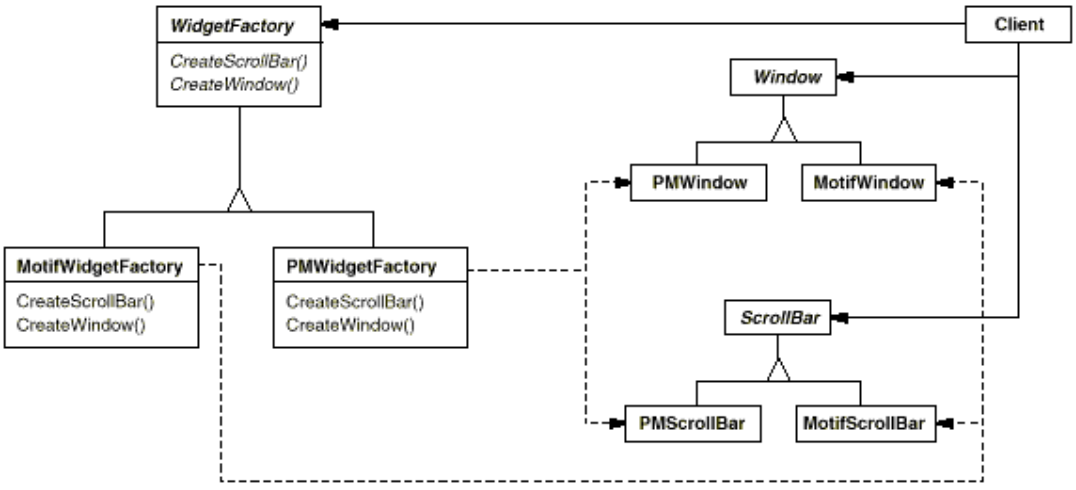
Also Known As

Kit

Motivation

Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons. To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel later.

We can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget. There's also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards. WidgetFactory's interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients aren't aware of the concrete classes they're using. Thus clients stay independent of the prevailing look and feel.



There is a concrete subclass of WidgetFactory for each look-and-feel standard. Each subclass implements the operations to create the appropriate widget for the look and feel. For example, the CreateScrollBar operation on the MotifWidgetFactory instantiates and returns a Motif scroll bar, while the corresponding operation on the PMWidgetFactory returns a scroll bar for Presentation Manager. Clients create widgets solely through the WidgetFactory interface and have no knowledge of the classes that implement widgets for a particular look and feel. In other words, clients only have to commit to an interface defined by an abstract class, not a particular concrete class.

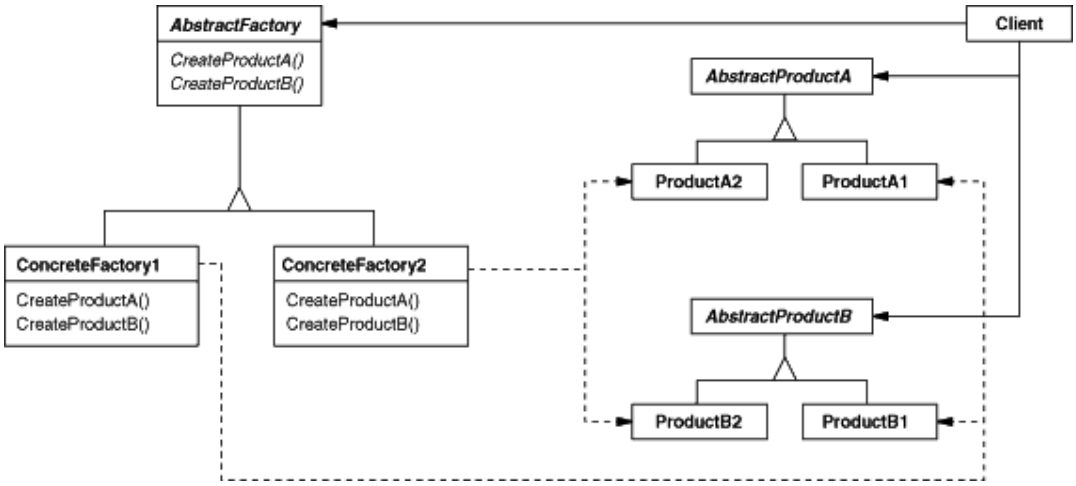
A WidgetFactory also enforces dependencies between the concrete widget classes. A Motif scroll bar should be used with a Motif button and a Motif text editor, and that constraint is enforced automatically as a consequence of using a MotifWidgetFactory.

Applicability

Use the Abstract Factory pattern when

- a system should be independent of how its products are created, composed, and represented.
- a system should be configured with one of multiple families of products.
- a family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure



Participants

- **AbstractFactory** (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- **AbstractProduct** (Window, ScrollBar)
  - declares an interface for a type of product object.
- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Collaborations

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences

The Abstract Factory pattern has the following benefits and liabilities:

1. *It isolates concrete classes.* The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.
2. *It makes exchanging product families easy.* The class of a concrete factory appears only once in an application—that is, where it's instantiated. This makes it easy to change the concrete factory an application uses. It can use different product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once. In our user interface example, we can switch from Motif widgets to Presentation Manager widgets simply by switching the corresponding factory objects and recreating the interface.
3. *It promotes consistency among products.* When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.
4. *Supporting new kinds of products is difficult.* Extending abstract factories to produce new kinds of Products isn't easy. That's because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses. We discuss one solution to this problem in the Implementation section.

Implementation

Here are some useful techniques for implementing the Abstract Factory pattern.

1. *Factories as singletons.* An application typically needs only one instance of a ConcreteFactory per product family. So it's usually best implemented as a Singleton (144).
2. *Creating the products.* AbstractFactory only declares an *interface* for creating products. It's up to ConcreteProduct subclasses to actually create them. The most common way to do this is to define a factory method (see Factory Method (121)) for each product. A concrete factory will specify

its products by overriding the factory method for each. While this implementation is simple, it requires a new concrete factory subclass for each product family, even if the product families differ only slightly.

If many product families are possible, the concrete factory can be implemented using the Prototype (133) pattern. The concrete factory is initialized with a prototypical instance of each product in the family, and it creates a new product by cloning its prototype. The Prototype-based approach eliminates the need for a new concrete factory class for each new product family.

Here's a way to implement a Prototype-based factory in Smalltalk. The concrete factory stores the prototypes to be cloned in a dictionary called partCatalog. The method make: retrieves the prototype and clones it:

```
make: partName
      ^ (partCatalog at: partName) copy
```

The concrete factory has a method for adding parts to the catalog.

```
addPart: partTemplate named: partName
   partCatalog at: partName put: partTemplate
```

Prototypes are added to the factory by identifying them with a symbol:

```
aFactory addPart: aPrototype named: #ACMEWidget
```

A variation on the Prototype-based approach is possible in languages that treat classes as first-class objects (Smalltalk and Objective C, for example). You can think of a class in these languages as a degenerate factory that creates only one kind of product. You can store *classes* inside a concrete factory that create the various concrete products in variables, much like prototypes. These classes create new instances on behalf of the concrete factory. You define a new factory by initializing an instance of a concrete factory with *classes* of products rather than by subclassing. This approach takes advantage of language characteristics, whereas the pure Prototype-based approach is language-independent.

Like the Prototype-based factory in Smalltalk just discussed, the class-based version will have a single instance variable partCatalog, which is a dictionary whose key is the name of the part. Instead of storing prototypes to be cloned, partCatalog stores the classes of the products. The method make: now looks like this:

```
make: partName
```

```
^ (partCatalog at: partName) new
```

3. *Defining extensible factories.* AbstractFactory usually defines a different operation for each kind of product it can produce. The kinds of products are encoded in the operation signatures. Adding a new kind of product requires changing the AbstractFactory interface and all the classes that depend on it.

   A more flexible but less safe design is to add a parameter to operations that create objects. This parameter specifies the kind of object to be created. It could be a class identifier, an integer, a string, or anything else that identifies the kind of product. In fact with this approach, AbstractFactory only needs a single "Make" operation with a parameter indicating the kind of object to create. This is the technique used in the Prototype- and the class-based abstract factories discussed earlier.

   This variation is easier to use in a dynamically typed language like Smalltalk than in a statically typed language like C++. You can use it in C++ only when all objects have the same abstract base class or when the product objects can be safely coerced to the correct type by the client that requested them. The implementation section of Factory Method (121) shows how to implement such parameterized operations in C++.

   But even when no coercion is needed, an inherent problem remains: All products are returned to the client with the *same* abstract interface as given by the return type. The client will not be able to differentiate or make safe assumptions about the class of a product. If clients need to perform subclass-specific operations, they won't be accessible through the abstract interface. Although the client could perform a downcast (e.g., with dynamic_cast in C++), that's not always feasible or safe, because the downcast can fail. This is the classic trade-off for a highly flexible and extensible interface.

Sample Code

We'll apply the Abstract Factory pattern to creating the mazes we discussed at the beginning of this chapter.

Class MazeFactory can create components of mazes. It builds rooms, walls, and doors between rooms. It might be used by a program that reads plans for mazes from a file and builds the corresponding maze. Or it might be used by a program that builds mazes randomly. Programs that build mazes take a MazeFactory as an argument so that the programmer can specify the classes of rooms, walls, and doors to construct.

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

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Recall that the member function CreateMaze builds a small maze consisting of two rooms with a door between them. CreateMaze hard-codes the class names, making it difficult to create mazes with different components.

Here's a version of CreateMaze that remedies that shortcoming by taking a MazeFactory as a parameter:

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());

    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

We can create EnchantedMazeFactory, a factory for enchanted mazes, by subclassing MazeFactory. EnchantedMazeFactory will override different member functions and return different subclasses of Room, Wall, etc.

```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n)  const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2)  const
        { return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};
```

Now suppose we want to make a maze game in which a room can have a bomb set in it. If the bomb goes off, it will damage the walls (at least). We can make a subclass of Room keep track of whether the room has a bomb in it and whether the bomb has gone off. We'll also need a subclass of Wall to keep track of the damage done to the wall. We'll call these classes RoomWithABomb and BombedWall.

The last class we'll define is BombedMazeFactory, a subclass of MazeFactory that ensures walls are of class BombedWall and rooms are of class RoomWithABomb. BombedMazeFactory only needs to override two functions:

```
Wall* BombedMazeFactory::MakeWall () const {
    return new BombedWall;
}

Room* BombedMazeFactory::MakeRoom(int n) const {
    return new RoomWithABomb(n);
}
```

To build a simple maze that can contain bombs, we simply call CreateMaze with a BombedMazeFactory.

```
MazeGame game;
BombedMazeFactory factory;
```

```
game.CreateMaze(factory);
```

CreateMaze can take an instance of EnchantedMazeFactory just as well to build enchanted mazes.

Notice that the MazeFactory is just a collection of factory methods. This is the most common way to implement the Abstract Factory pattern. Also note that MazeFactory is not an abstract class; thus it acts as both the AbstractFactory *and* the ConcreteFactory. This is another common implementation for simple applications of the Abstract Factory pattern. Because the MazeFactory is a concrete class consisting entirely of factory methods, it's easy to make a new MazeFactory by making a subclass and overriding the operations that need to change.

CreateMaze used the SetSide operation on rooms to specify their sides. If it creates rooms with a BombedMazeFactory, then the maze will be made up of RoomWithABomb objects with BombedWall sides. If RoomWithABomb had to access a subclass-specific member of BombedWall, then it would have to cast a reference to its walls from Wall* to BombedWall*. This downcasting is safe as long as the argument *is* in fact a BombedWall, which is guaranteed to be true if walls are built solely with a BombedMazeFactory.

Dynamically typed languages such as Smalltalk don't require downcasting, of course, but they might produce run-time errors if they encounter a Wall where they expect a *subclass* of Wall. Using Abstract Factory to build walls helps prevent these run-time errors by ensuring that only certain kinds of walls can be created.

Let's consider a Smalltalk version of MazeFactory, one with a single make operation that takes the kind of object to make as a parameter. Moreover, the concrete factory stores the classes of the products it creates.

First, we'll write an equivalent of CreateMaze in Smalltalk:

```
createMaze: aFactory
    | room1 room2 aDoor |
    room1 := (aFactory make: #room) number: 1.
    room2 := (aFactory make: #room) number: 2.
    aDoor := (aFactory make: #door) from: room1 to: room2.
    room1 atSide: #north put: (aFactory make: #wall).
    room1 atSide: #east put: aDoor.
    room1 atSide: #south put: (aFactory make: #wall).
```

```
        room1 atSide: #west put: (aFactory make: #wall).
        room2 atSide: #north put: (aFactory make: #wall).
        room2 atSide: #east put: (aFactory make: #wall).
        room2 atSide: #south put: (aFactory make: #wall).
        room2 atSide: #west put: aDoor.
        ^ Maze new addRoom: room1; addRoom: room2; yourself
```

As we discussed in the Implementation section, MazeFactory needs only a single instance variable partCatalog to provide a dictionary whose key is the class of the component. Also recall how we implemented the make: method:

```
make: partName
    ^ (partCatalog at: partName) new
```

Now we can create a MazeFactory and use it to implement createMaze. We'll create the factory using a method createMazeFactory of class MazeGame.

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: Room named: #room;
        addPart: Door named: #door;
        yourself)
```

A BombedMazeFactory or EnchantedMazeFactory is created by associating different classes with the keys. For example, an EnchantedMazeFactory could be created like this:

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: EnchantedRoom named: #room;
        addPart: DoorNeedingSpell named: #door;
        yourself)
```

Known Uses

InterViews uses the "Kit" suffix [Lin92] to denote AbstractFactory classes. It defines WidgetKit and DialogKit abstract factories for generating look-and-feel-specific user interface objects. InterViews also includes a LayoutKit that generates different composition objects depending on the layout

desired. For example, a layout that is conceptually horizontal may require different composition objects depending on the document's orientation (portrait or landscape).

ET++ [WGM88] uses the Abstract Factory pattern to achieve portability across different window systems (X Windows and SunView, for example). The WindowSystem abstract base class defines the interface for creating objects that represent window system resources (MakeWindow, MakeFont, MakeColor, for example). Concrete subclasses implement the interfaces for a specific window system. At run-time, ET++ creates an instance of a concrete WindowSystem subclass that creates concrete system resource objects.

Related Patterns

AbstractFactory classes are often implemented with factory methods (Factory Method (121)), but they can also be implemented using Prototype (133).

A concrete factory is often a singleton (Singleton (144)).

Builder

Intent

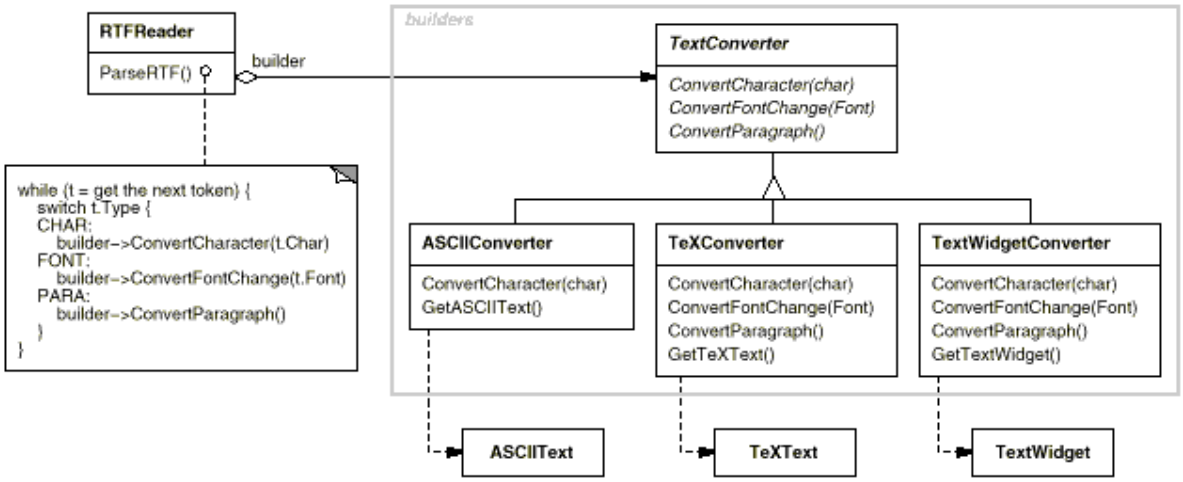
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Motivation

A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats. The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively. The problem, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.

A solution is to configure the RTFReader class with a TextConverter object that converts RTF to another textual representation. As the RTFReader parses the RTF document, it uses the TextConverter to perform the conversion. Whenever the RTFReader recognizes an RTF token (either plain text or an RTF control word), it issues a request to the TextConverter to convert the token. TextConverter objects are responsible both for performing the data conversion and for representing the token in a particular format.

Subclasses of TextConverter specialize in different conversions and formats. For example, an ASCIIConverter ignores requests to convert anything except plain text. A TeXConverter, on the other hand, will implement operations for all requests in order to produce a TeX representation that captures all the stylistic information in the text. A TextWidgetConverter will produce a complex user interface object that lets the user see and edit the text.



Each kind of converter class takes the mechanism for creating and assembling a complex object and puts it behind an abstract interface. The converter is separate from the reader, which is responsible for parsing an RTF document.

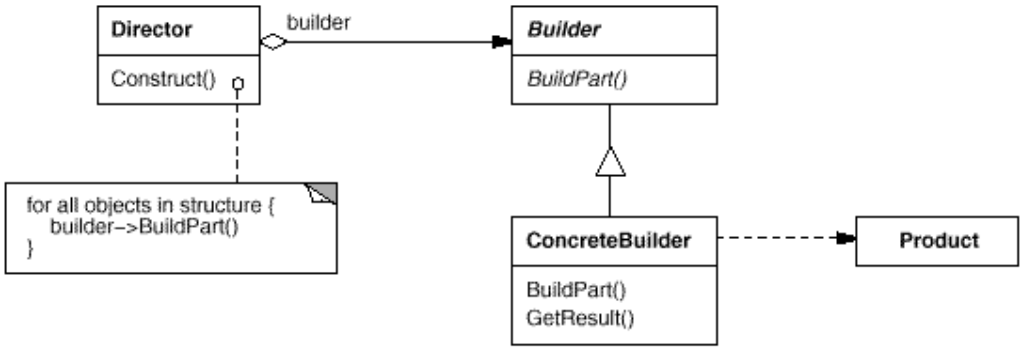
The Builder pattern captures all these relationships. Each converter class is called a **builder** in the pattern, and the reader is called the **director**. Applied to this example, the Builder pattern separates the algorithm for interpreting a textual format (that is, the parser for RTF documents) from how a converted format gets created and represented. This lets us reuse the RTFReader's parsing algorithm to create different text representations from RTF documents—just configure the RTFReader with different subclasses of TextConverter.

Applicability

Use the Builder pattern when

- the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- the construction process must allow different representations for the object that's constructed.

Structure



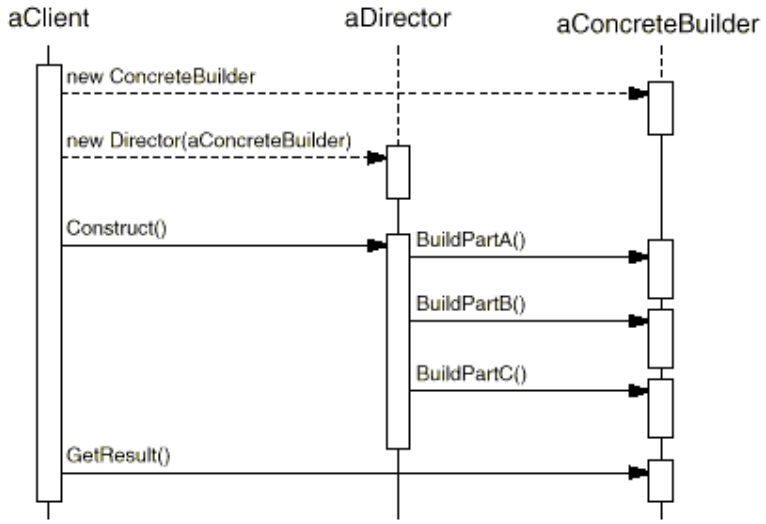
Participants

- **Builder** (TextConverter)
  - specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder** (ASCIIConverter, TeXConverter, TextWidgetConverter)
  - constructs and assembles parts of the product by implementing the Builder interface.
  - defines and keeps track of the representation it creates.
  - provides an interface for retrieving the product (e.g., GetASCIIText, GetTextWidget).
- **Director** (RTFReader)
  - constructs an object using the Builder interface.
- **Product** (ASCIIText, TeXText, TextWidget)
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Collaborations

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

The following interaction diagram illustrates how Builder and Director cooperate with a client.



Consequences

Here are key consequences of the Builder pattern:

1. *It lets you vary a product's internal representation.* The Builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled. Because the product is constructed through an abstract interface, all you have to do to change the product's internal representation is define a new kind of builder.
2. *It isolates code for construction and representation.* The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes that define the product's internal structure; such classes don't appear in Builder's interface.

   Each ConcreteBuilder contains all the code to create and assemble a particular kind of product. The code is written once; then different Directors can reuse it to build Product variants from the same set of parts. In the earlier RTF example, we could define a reader for a format other than RTF, say, an SGMLReader, and use the same TextConverters to generate ASCIIText, TeXText, and TextWidget renditions of SGML documents.

3. *It gives you finer control over the construction process.* Unlike creational patterns that construct products in one shot, the Builder pattern constructs

the product step by step under the director's control. Only when the product is finished does the director retrieve it from the builder. Hence the Builder interface reflects the process of constructing the product more than other creational patterns. This gives you finer control over the construction process and consequently the internal structure of the resulting product.

Implementation

Typically there's an abstract Builder class that defines an operation for each component that a director may ask it to create. The operations do nothing by default. A ConcreteBuilder class overrides operations for components it's interested in creating.

Here are other implementation issues to consider:

1. *Assembly and construction interface.* Builders construct their products in step-by-step fashion. Therefore the Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders.

   A key design issue concerns the model for the construction and assembly process. A model where the results of construction requests are simply appended to the product is usually sufficient. In the RTF example, the builder converts and appends the next token to the text it has converted so far.

   But sometimes you might need access to parts of the product constructed earlier. In the Maze example we present in the Sample Code, the MazeBuilder interface lets you add a door between existing rooms. Tree structures such as parse trees that are built bottom-up are another example. In that case, the builder would return child nodes to the director, which then would pass them back to the builder to build the parent nodes.

2. *Why no abstract class for products?* In the common case, the products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class. In the RTF example, the ASCIIText and the TextWidget objects are unlikely to have a common interface, nor do they need one. Because the client usually configures the director with the proper concrete builder, the client is in a position to know which concrete subclass of Builder is in use and can handle its products accordingly.
3. *Empty methods as default in Builder.* In C++, the build methods are intentionally not declared pure virtual member functions. They're defined

as empty methods instead, letting clients override only the operations they're interested in.

Sample Code

We'll define a variant of the CreateMaze member function that takes a builder of class MazeBuilder as an argument.

The MazeBuilder class defines the following interface for building mazes:

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }

    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

This interface can create three things: (1) the maze, (2) rooms with a particular room number, and (3) doors between numbered rooms. The GetMaze operation returns the maze to the client. Subclasses of MazeBuilder will override this operation to return the maze that they build.

All the maze-building operations of MazeBuilder do nothing by default. They're not declared pure virtual to let derived classes override only those methods in which they're interested.

Given the MazeBuilder interface, we can change the CreateMaze member function to take this builder as a parameter.

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

Compare this version of CreateMaze with the original. Notice how the builder hides the internal representation of the Maze—that is, the classes that define rooms, doors, and walls—and how these parts are assembled to complete the final maze. Someone might guess that there are classes for representing rooms and doors, but there is no hint of one for walls. This makes it easier to change the way a maze is represented, since none of the clients of MazeBuilder has to be changed.

Like the other creational patterns, the Builder pattern encapsulates how objects get created, in this case through the interface defined by MazeBuilder. That means we can reuse MazeBuilder to build different kinds of mazes. The CreateComplexMaze operation gives an example:

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    // ...
    builder.BuildRoom(1001);

    return builder.GetMaze();
}
```

Note that MazeBuilder does not create mazes itself; its main purpose is just to define an interface for creating mazes. It defines empty implementations primarily for convenience. Subclasses of MazeBuilder do the actual work.

The subclass StandardMazeBuilder is an implementation that builds simple mazes. It keeps track of the maze it's building in the variable _currentMaze.

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);

    virtual Maze* GetMaze();
private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};
```

CommonWall is a utility operation that determines the direction of the common wall between two rooms.

The StandardMazeBuilder constructor simply initializes _currentMaze.

```
StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}
```

BuildMaze instantiates a Maze that other operations will assemble and eventually return to the client (with GetMaze).

```
void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

Maze* StandardMazeBuilder::GetMaze () {
    return _currentMaze;
}
```

The BuildRoom operation creates a room and builds the walls around it:

```
void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}
```

To build a door between two rooms, StandardMazeBuilder looks up both rooms in the maze and finds their adjoining wall:

```
void StandardMazeBuilder::BuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
```

```
}
```

Clients can now use CreateMaze in conjunction with StandardMazeBuilder to create a maze:

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();
```

We could have put all the StandardMazeBuilder operations in Maze and let each Maze build itself. But making Maze smaller makes it easier to understand and modify, and StandardMazeBuilder is easy to separate from Maze. Most importantly, separating the two lets you have a variety of MazeBuilders, each using different classes for rooms, walls, and doors.

A more exotic MazeBuilder is CountingMazeBuilder. This builder doesn't create a maze at all; it just counts the different kinds of components that would have been created.

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);

    void GetCounts(int&, int&) const;
private:
    int _doors;
    int _rooms;
};
```

The constructor initializes the counters, and the overridden MazeBuilder operations increment them accordingly.

```
CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
```

```
}

void CountingMazeBuilder::BuildRoom (int) {
    _rooms++;
}

void CountingMazeBuilder::BuildDoor (int, int) {
    _doors++;
}

void CountingMazeBuilder::GetCounts (
    int& rooms, int& doors
) const {
    rooms = _rooms;
    doors = _doors;
}
```

Here's how a client might use a CountingMazeBuilder:

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

cout << "The maze has "
     << rooms << " rooms and "
     << doors << " doors" << endl;
```

Known Uses

The RTF converter application is from ET++ [WGM88]. Its text building block uses a builder to process text stored in the RTF format.

Builder is a common pattern in Smalltalk-80 [Par90]:

- The Parser class in the compiler subsystem is a Director that takes a ProgramNodeBuilder object as an argument. A Parser object notifies its ProgramNodeBuilder object each time it recognizes a syntactic construct. When the parser is done, it asks the builder for the parse tree it built and returns it to the client.

- ClassBuilder is a builder that Classes use to create subclasses for themselves. In this case a Class is both the Director and the Product.
- ByteCodeStream is a builder that creates a compiled method as a byte array. ByteCodeStream is a nonstandard use of the Builder pattern, because the complex object it builds is encoded as a byte array, not as a normal Smalltalk object. But the interface to ByteCodeStream is typical of a builder, and it would be easy to replace ByteCodeStream with a different class that represented programs as a composite object.

The Service Configurator framework from the Adaptive Communications Environment uses a builder to construct network service components that are linked into a server at run-time [SS94]. The components are described with a configuration language that's parsed by an LALR(1) parser. The semantic actions of the parser perform operations on the builder that add information to the service component. In this case, the parser is the Director.

Related Patterns

Abstract Factory (99) is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.

A Composite (183) is what the builder often builds.

Factory Method

Intent

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

Also Known As

Virtual Constructor

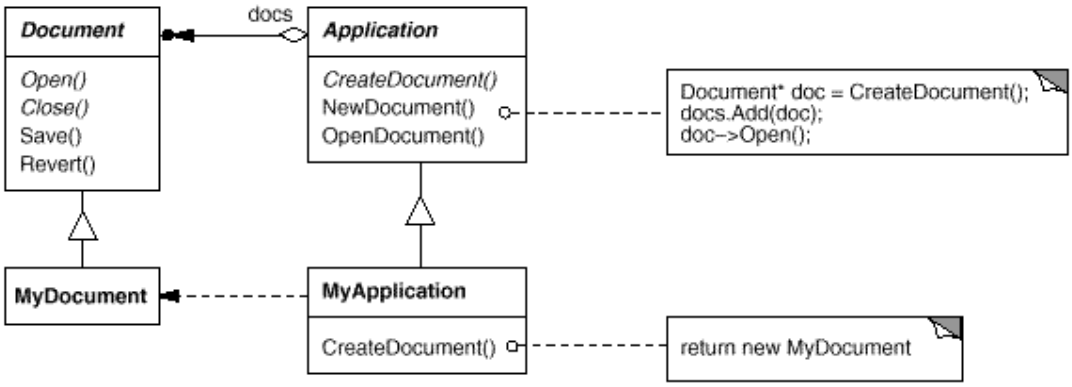
Motivation

Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.

Consider a framework for applications that can present multiple documents to the user. Two key abstractions in this framework are the classes Application and Document. Both classes are abstract, and clients have to subclass them to realize their application-specific implementations. To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument. The Application class is responsible for managing Documents and will create them as required—when the user selects Open or New from a menu, for example.

Because the particular Document subclass to instantiate is application-specific, the Application class can't predict the subclass of Document to instantiate—the Application class only knows *when* a new document should be created, not *what kind* of Document to create. This creates a dilemma: The framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.

The Factory Method pattern offers a solution. It encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework.



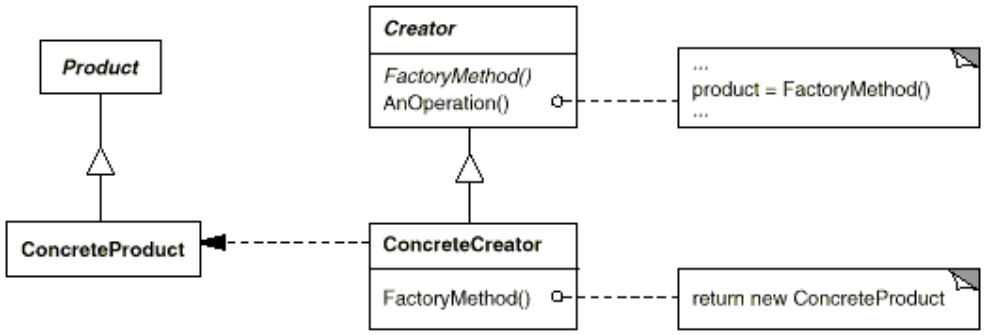
Application subclasses redefine an abstract CreateDocument operation on Application to return the appropriate Document subclass. Once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class. We call CreateDocument a **factory method** because it's responsible for "manufacturing" an object.

Applicability

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure



Participants

- **Product** (Document)
  - defines the interface of objects the factory method creates.
- **ConcreteProduct** (MyDocument)
  - implements the Product interface.
- **Creator** (Application)
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
- **ConcreteCreator** (MyApplication)
  - overrides the factory method to return an instance of a ConcreteProduct.

Collaborations

- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences

Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.

A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object. Subclassing is fine when the client has to subclass the Creator class anyway, but otherwise the client now must deal with another point of evolution.

Here are two additional consequences of the Factory Method pattern:

1. *Provides hooks for subclasses.* Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.

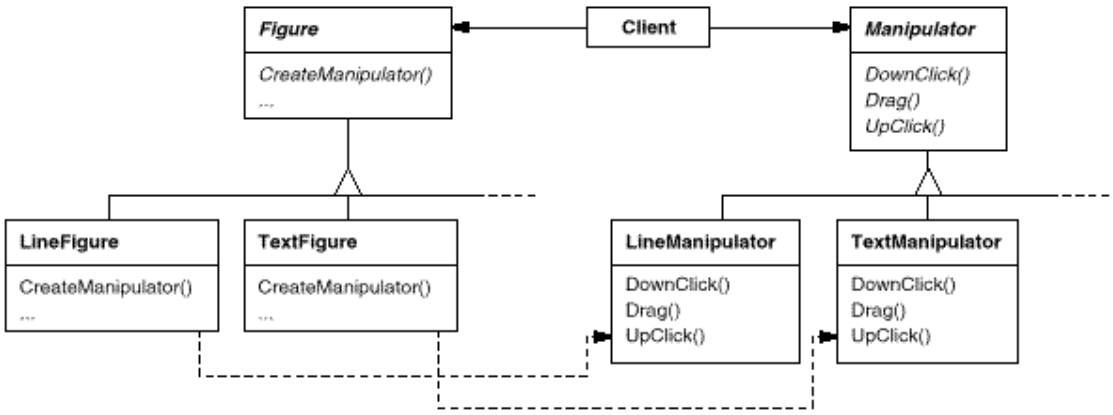
   In the Document example, the Document class could define a factory method called CreateFileDialog that creates a default file dialog object for opening an existing document. A Document subclass can define an application-specific file dialog by overriding this factory method. In this

case the factory method is not abstract but provides a reasonable default implementation.

2. *Connects parallel class hierarchies.* In the examples we've considered so far, the factory method is only called by Creators. But this doesn't have to be the case; clients can find factory methods useful, especially in the case of parallel class hierarchies.

   Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class. Consider graphical figures that can be manipulated interactively; that is, they can be stretched, moved, or rotated using the mouse. Implementing such interactions isn't always easy. It often requires storing and updating information that records the state of the manipulation at a given time. This state is needed only during manipulation; therefore it needn't be kept in the figure object. Moreover, different figures behave differently when the user manipulates them. For example, stretching a line figure might have the effect of moving an endpoint, whereas stretching a text figure may change its line spacing.

   With these constraints, it's better to use a separate Manipulator object that implements the interaction and keeps track of any manipulation-specific state that's needed. Different figures will use different Manipulator subclasses to handle particular interactions. The resulting Manipulator class hierarchy parallels (at least partially) the Figure class hierarchy:



   The Figure class provides a CreateManipulator factory method that lets clients create a Figure's corresponding Manipulator. Figure subclasses override this method to return an instance of the Manipulator subclass that's right for them. Alternatively, the Figure class may implement CreateManipulator to return a default Manipulator instance, and Figure

subclasses may simply inherit that default. The Figure classes that do so need no corresponding Manipulator subclass—hence the hierarchies are only partially parallel.

Notice how the factory method defines the connection between the two class hierarchies. It localizes knowledge of which classes belong together.

Implementation

Consider the following issues when applying the Factory Method pattern:

1. *Two major varieties.* The two main variations of the Factory Method pattern are (1) the case when the Creator class is an abstract class and does not provide an implementation for the factory method it declares, and (2) the case when the Creator is a concrete class and provides a default implementation for the factory method. It's also possible to have an abstract class that defines a default implementation, but this is less common.

   The first case *requires* subclasses to define an implementation, because there's no reasonable default. It gets around the dilemma of having to instantiate unforeseeable classes. In the second case, the concrete Creator uses the factory method primarily for flexibility. It's following a rule that says, "Create objects in a separate operation so that subclasses can override the way they're created." This rule ensures that designers of subclasses can change the class of objects their parent class instantiates if necessary.

2. *Parameterized factory methods.* Another variation on the pattern lets the factory method create *multiple* kinds of products. The factory method takes a parameter that identifies the kind of object to create. All objects the factory method creates will share the Product interface. In the Document example, Application might support different kinds of Documents. You pass CreateDocument an extra parameter to specify the kind of document to create.

   The Unidraw graphical editing framework [VL90] uses this approach for reconstructing objects saved on disk. Unidraw defines a Creator class with a factory method Create that takes a class identifier as an argument. The class identifier specifies the class to instantiate. When Unidraw saves an object to disk, it writes out the class identifier first and then its instance variables. When it reconstructs the object from disk, it reads the class identifier first.

Once the class identifier is read, the framework calls Create, passing the identifier as the parameter. Create looks up the constructor for the corresponding class and uses it to instantiate the object. Last, Create calls the object's Read operation, which reads the remaining information on the disk and initializes the object's instance variables.

A parameterized factory method has the following general form, where MyProduct and YourProduct are subclasses of Product:

```
class Creator {
public:
   virtual Product* Create(ProductId);
};

Product* Creator::Create (ProductId id) {
   if (id == MINE)  return new MyProduct;
   if (id == YOURS) return new YourProduct;
   // repeat for remaining products...

   return 0;
}
```

Overriding a parameterized factory method lets you easily and selectively extend or change the products that a Creator produces. You can introduce new identifiers for new kinds of products, or you can associate existing identifiers with different products.

For example, a subclass MyCreator could swap MyProduct and YourProduct and support a new TheirProduct subclass:

```
Product* MyCreator::Create (ProductId id) {
   if (id == YOURS)  return new MyProduct;
   if (id == MINE)   return new YourProduct;
      // N.B.: switched YOURS and MINE

   if (id == THEIRS) return new TheirProduct;

   return Creator::Create(id); // called if all others fail
}
```

Notice that the last thing this operation does is call Create on the parent class. That's because MyCreator::Create handles only YOURS, MINE, and THEIRS differently than the parent class. It isn't interested in other

classes. Hence MyCreator *extends* the kinds of products created, and it defers responsibility for creating all but a few products to its parent.

3. *Language-specific variants and issues.* Different languages lend themselves to other interesting variations and caveats.

   Smalltalk programs often use a method that returns the class of the object to be instantiated. A Creator factory method can use this value to create a product, and a ConcreteCreator may store or even compute this value. The result is an even later binding for the type of ConcreteProduct to be instantiated.

   A Smalltalk version of the Document example can define a documentClass method on Application. The documentClass method returns the proper Document class for instantiating documents. The implementation of documentClass in MyApplication returns the MyDocument class. Thus in class Application we have

```
clientMethod
   document := self documentClass new.

documentClass
   self subclassResponsibility
```

   In class MyApplication we have

```
documentClass
   ^ MyDocument
```

   which returns the class MyDocument to be instantiated to Application.

   An even more flexible approach akin to parameterized factory methods is to store the class to be created as a class variable of Application. That way you don't have to subclass Application to vary the product.

   Factory methods in C++ are always virtual functions and are often pure virtual. Just be careful not to call factory methods in the Creator's constructor—the factory method in the ConcreteCreator won't be available yet.

   You can avoid this by being careful to access products solely through accessor operations that create the product on demand. Instead of creating the concrete product in the constructor, the constructor merely initializes

it to 0. The accessor returns the product. But first it checks to make sure the product exists, and if it doesn't, the accessor creates it. This technique is sometimes called **lazy initialization**. The following code shows a typical implementation:

```
class Creator {
public:
   Product* GetProduct();
protected:
   virtual Product* CreateProduct();
private:
   Product* _product;
};

Product* Creator::GetProduct () {
   if (_product == 0) {
      _product = CreateProduct();
   }
   return _product;
}
```

4. *Using templates to avoid subclassing.* As we've mentioned, another potential problem with factory methods is that they might force you to subclass just to create the appropriate Product objects. Another way to get around this in C++ is to provide a template subclass of Creator that's parameterized by the Product class:

```
class Creator {
public:
   virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
   virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
   return new TheProduct;
}
```

With this template, the client supplies just the product class—no subclassing of Creator is required.

```
class MyProduct : public Product {
public:
   MyProduct();
   // ...
};

StandardCreator<MyProduct> myCreator;
```

5. *Naming conventions.* It's good practice to use naming conventions that make it clear you're using factory methods. For example, the MacApp Macintosh application framework [App89] always declares the abstract operation that defines the factory method as Class* DoMakeClass(), where Class is the Product class.

Sample Code

The function CreateMaze builds and returns a maze. One problem with this function is that it hard-codes the classes of maze, rooms, doors, and walls. We'll introduce factory methods to let subclasses choose these components.

First we'll define factory methods in MazeGame for creating the maze, room, wall, and door objects:

```
class MazeGame {
public:
   Maze* CreateMaze();

// factory methods:

   virtual Maze* MakeMaze() const
      { return new Maze; }
   virtual Room* MakeRoom(int n) const
      { return new Room(n); }
   virtual Wall* MakeWall() const
      { return new Wall; }
   virtual Door* MakeDoor(Room* r1, Room* r2) const
      { return new Door(r1, r2); }
};
```

Each factory method returns a maze component of a given type. MazeGame provides default implementations that return the simplest kinds of maze, rooms, walls, and doors.

Now we can rewrite CreateMaze to use these factory methods:

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());

    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

Different games can subclass MazeGame to specialize parts of the maze. MazeGame subclasses can redefine some or all of the factory methods to specify variations in products. For example, a BombedMazeGame can redefine the Room and Wall products to return the bombed varieties:

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
```

```
        { return new RoomWithABomb(n); }
};
```

An EnchantedMazeGame variant might be defined like this:

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

Known Uses

Factory methods pervade toolkits and frameworks. The preceding document example is a typical use in MacApp and ET++ [WGM88]. The manipulator example is from Unidraw.

Class View in the Smalltalk-80 Model/View/Controller framework has a method defaultController that creates a controller, and this might appear to be a factory method [Par90]. But subclasses of View specify the class of their default controller by defining defaultControllerClass, which returns the class from which defaultController creates instances. So defaultControllerClass is the real factory method, that is, the method that subclasses should override.

A more esoteric example in Smalltalk-80 is the factory method parserClass defined by Behavior (a superclass of all objects representing classes). This enables a class to use a customized parser for its source code. For example, a client can define a class SQLParser to analyze the source code of a class with embedded SQL statements. The Behavior class implements parserClass to return the standard Smalltalk Parser class. A class that includes embedded SQL statements overrides this method (as a class method) and returns the SQLParser class.

The Orbix ORB system from IONA Technologies [ION94] uses Factory Method to generate an appropriate type of proxy (see Proxy (233)) when an object requests a reference to a remote object. Factory Method makes it easy to replace the default proxy with one that uses client-side caching, for example.

Related Patterns

Abstract Factory (99) is often implemented with factory methods. The Motivation example in the Abstract Factory pattern illustrates Factory Method as well.

Factory methods are usually called within Template Methods (360). In the document example above, NewDocument is a template method.

Prototypes (133) don't require subclassing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation.

Prototype

Intent

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

Motivation

You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves. The editor framework may have a palette of tools for adding these music objects to the score. The palette would also include tools for selecting, moving, and otherwise manipulating music objects. Users will click on the quarter-note tool and use it to add quarter notes to the score. Or they can use the move tool to move a note up or down on the staff, thereby changing its pitch.

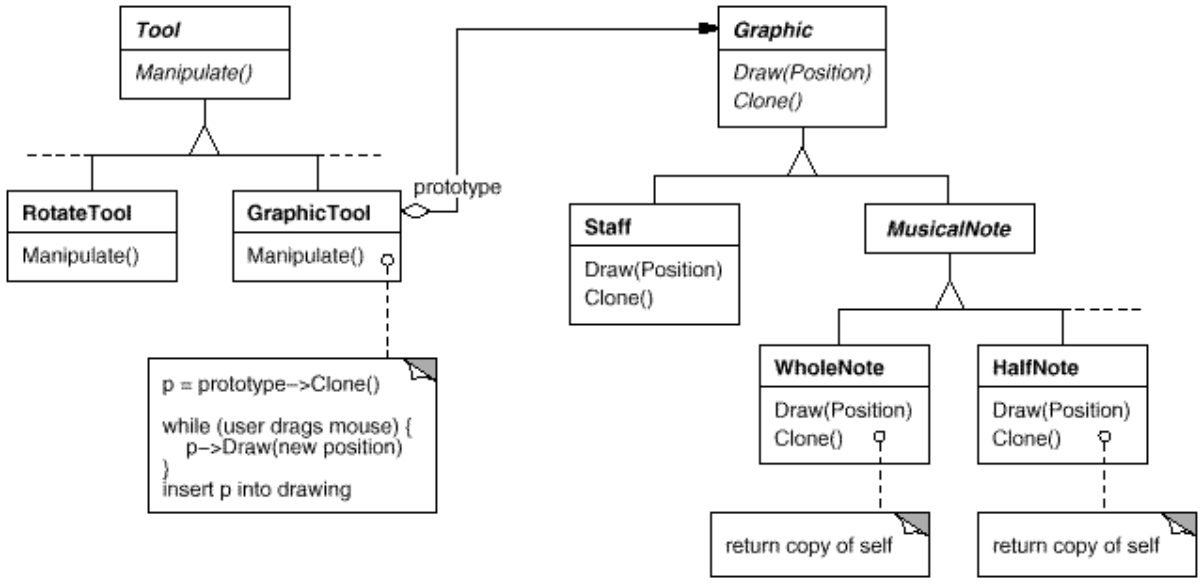
Let's assume the framework provides an abstract Graphic class for graphical components, like notes and staves. Moreover, it'll provide an abstract Tool class for defining tools like those in the palette. The framework also predefines a GraphicTool subclass for tools that create instances of graphical objects and add them to the document.

But GraphicTool presents a problem to the framework designer. The classes for notes and staves are specific to our application, but the GraphicTool class belongs to the framework. GraphicTool doesn't know how to create instances of our music classes to add to the score. We could subclass GraphicTool for each kind of music object, but that would produce lots of subclasses that differ only in the kind of music object they instantiate. We know object composition is a flexible alternative to subclassing. The question is, how can the framework use it to parameterize instances of GraphicTool by the *class* of Graphic they're supposed to create?

The solution lies in making GraphicTool create a new Graphic by copying or "cloning" an instance of a Graphic subclass. We call this instance a **prototype**. GraphicTool is parameterized by the prototype it should clone and add to the document. If all Graphic subclasses support a Clone operation, then the GraphicTool can clone any kind of Graphic.

So in our music editor, each tool for creating a music object is an instance of GraphicTool that's initialized with a different prototype. Each GraphicTool

instance will produce a music object by cloning its prototype and adding the clone to the score.



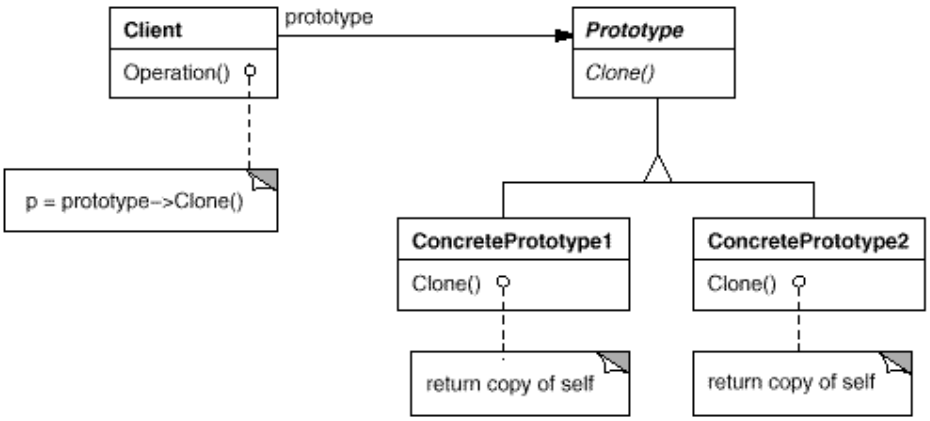
We can use the Prototype pattern to reduce the number of classes even further. We have separate classes for whole notes and half notes, but that's probably unnecessary. Instead they could be instances of the same class initialized with different bitmaps and durations. A tool for creating whole notes becomes just a GraphicTool whose prototype is a MusicalNote initialized to be a whole note. This can reduce the number of classes in the system dramatically. It also makes it easier to add a new kind of note to the music editor.

Applicability

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; *and*

- when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
- when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Structure



Participants

- **Prototype** (Graphic)
  - declares an interface for cloning itself.
- **ConcretePrototype** (Staff, WholeNote, HalfNote)
  - implements an operation for cloning itself.
- **Client** (GraphicTool)
  - creates a new object by asking a prototype to clone itself.

Collaborations

- A client asks a prototype to clone itself.

Consequences

Prototype has many of the same consequences that Abstract Factory (99) and Builder (110) have: It hides the concrete product classes from the client, thereby reducing the number of names clients know about. Moreover, these patterns let a client work with application-specific classes without modification.

Additional benefits of the Prototype pattern are listed below.

1. *Adding and removing products at run-time.* Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That's a bit more flexible than other

creational patterns, because a client can install and remove prototypes at run-time.

2. *Specifying new objects by varying values.* Highly dynamic systems let you define new behavior through object composition—by specifying values for an object's variables, for example—and not by defining new classes. You effectively define new kinds of objects by instantiating existing classes and registering the instances as prototypes of client objects. A client can exhibit new behavior by delegating responsibility to the prototype.

   This kind of design lets users define new "classes" without programming. In fact, cloning a prototype is similar to instantiating a class. The Prototype pattern can greatly reduce the number of classes a system needs. In our music editor, one GraphicTool class can create a limitless variety of music objects.

3. *Specifying new objects by varying structure.* Many applications build objects from parts and subparts. Editors for circuit design, for example, build circuits out of subcircuits.[1] For convenience, such applications often let you instantiate complex, user-defined structures, say, to use a specific subcircuit again and again.

   The Prototype pattern supports this as well. We simply add this subcircuit as a prototype to the palette of available circuit elements. As long as the composite circuit object implements Clone as a deep copy, circuits with different structures can be prototypes.

4. *Reduced subclassing.* Factory Method (121) often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don't need a Creator class hierarchy at all. This benefit applies primarily to languages like C++ that don't treat classes as first-class objects. Languages that do, like Smalltalk and Objective C, derive less benefit, since you can always use a class object as a creator. Class objects already act like prototypes in these languages.
5. *Configuring an application with classes dynamically.* Some run-time environments let you load classes into an application dynamically. The Prototype pattern is the key to exploiting such facilities in a language like C++.

   An application that wants to create instances of a dynamically loaded class won't be able to reference its constructor statically. Instead, the run-time environment creates an instance of each class automatically when it's loaded, and it registers the instance with a prototype manager (see the Implementation section). Then the application can ask the prototype manager

for instances of newly loaded classes, classes that weren't linked with the program originally. The ET++ application framework [WGM88] has a run-time system that uses this scheme.

The main liability of the Prototype pattern is that each subclass of Prototype must implement the Clone operation, which may be difficult. For example, adding Clone is difficult when the classes under consideration already exist. Implementing Clone can be difficult when their internals include objects that don't support copying or have circular references.

Implementation

Prototype is particularly useful with static languages like C++, where classes are not objects, and little or no type information is available at run-time. It's less important in languages like Smalltalk or Objective C that provide what amounts to a prototype (i.e., a class object) for creating instances of each class. This pattern is built into prototype-based languages like Self [US87], in which all object creation happens by cloning a prototype.

Consider the following issues when implementing prototypes:

1. *Using a prototype manager.* When the number of prototypes in a system isn't fixed (that is, they can be created and destroyed dynamically), keep a registry of available prototypes. Clients won't manage prototypes themselves but will store and retrieve them from the registry. A client will ask the registry for a prototype before cloning it. We call this registry a **prototype manager**.

   A prototype manager is an associative store that returns the prototype matching a given key. It has operations for registering a prototype under a key and for unregistering it. Clients can change or even browse through the registry at run-time. This lets clients extend and take inventory on the system without writing code.

2. *Implementing the Clone operation.* The hardest part of the Prototype pattern is implementing the Clone operation correctly. It's particularly tricky when object structures contain circular references.

   Most languages provide some support for cloning objects. For example, Smalltalk provides an implementation of copy that's inherited by all subclasses of Object. C++ provides a copy constructor. But these facilities don't solve the "shallow copy versus deep copy" problem [GR83]. That is, does cloning an object in turn clone its instance variables, or do the clone and original just share the variables?

A shallow copy is simple and often sufficient, and that's what Smalltalk provides by default. The default copy constructor in C++ does a member-wise copy, which means pointers will be shared between the copy and the original. But cloning prototypes with complex structures usually requires a deep copy, because the clone and the original must be independent. Therefore you must ensure that the clone's components are clones of the prototype's components. Cloning forces you to decide what if anything will be shared.

If objects in the system provide Save and Load operations, then you can use them to provide a default implementation of Clone simply by saving the object and loading it back immediately. The Save operation saves the object into a memory buffer, and Load creates a duplicate by reconstructing the object from the buffer.

3. *Initializing clones.* While some clients are perfectly happy with the clone as is, others will want to initialize some or all of its internal state to values of their choosing. You generally can't pass these values in the Clone operation, because their number will vary between classes of prototypes. Some prototypes might need multiple initialization parameters; others won't need any. Passing parameters in the Clone operation precludes a uniform cloning interface.

   It might be the case that your prototype classes already define operations for (re)setting key pieces of state. If so, clients may use these operations immediately after cloning. If not, then you may have to introduce an Initialize operation (see the Sample Code section) that takes initialization parameters as arguments and sets the clone's internal state accordingly. Beware of deep-copying Clone operations—the copies may have to be deleted (either explicitly or within Initialize) before you reinitialize them.

Sample Code

We'll define a MazePrototypeFactory subclass of the MazeFactory class. MazePrototypeFactory will be initialized with prototypes of the objects it will create so that we don't have to subclass it just to change the classes of walls or rooms it creates.

MazePrototypeFactory augments the MazeFactory interface with a constructor that takes the prototypes as arguments:

```
class MazePrototypeFactory : public MazeFactory {
public:
```

```
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

The new constructor simply initializes its prototypes:

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
  _prototypeMaze = m;
  _prototypeWall = w;
  _prototypeRoom = r;
  _prototypeDoor = d;
}
```

The member functions for creating walls, rooms, and doors are similar: Each clones a prototype and then initializes it. Here are the definitions of MakeWall and MakeDoor:

```
Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}
```

We can use MazePrototypeFactory to create a prototypical or default maze just by initializing it with prototypes of basic maze components:

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);
```

To change the type of maze, we initialize MazePrototypeFactory with a different set of prototypes. The following call creates a maze with a BombedDoor and a RoomWithABomb:

```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);
```

An object that can be used as a prototype, such as an instance of Wall, must support the Clone operation. It must also have a copy constructor for cloning. It may also need a separate operation for reinitializing internal state. We'll add the Initialize operation to Door to let clients initialize the clone's rooms.

Compare the following definition of Door to the one on page 96:

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;

    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
};

Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}
```

```
void Door::Initialize (Room* r1, Room* r2) {
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}
```

The BombedWall subclass must override Clone and implement a corresponding copy constructor.

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();
private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

Although BombedWall::Clone returns a Wall*, its implementation returns a pointer to a new instance of a subclass, that is, a BombedWall*. We define Clone like this in the base class to ensure that clients that clone the prototype don't have to know about their concrete subclasses. Clients should never need to downcast the return value of Clone to the desired type.

In Smalltalk, you can reuse the standard copy method inherited from Object to clone any MapSite. You can use MazeFactory to produce the prototypes you'll need; for example, you can create a room by supplying the name #room. The MazeFactory has a dictionary that maps names to prototypes. Its make: method looks like this:

```
make: partName
    ^ (partCatalog at: partName) copy
```

Given appropriate methods for initializing the MazeFactory with prototypes, you could create a simple maze with the following code:

```
CreateMaze
    on: (MazeFactory new
        with: Door new named: #door;
        with: Wall new named: #wall;
        with: Room new named: #room;
        yourself)
```

where the definition of the on: class method for CreateMaze would be

```
on: aFactory
    | room1 room2 |
    room1 := (aFactory make: #room) location: 1@1.
    room2 := (aFactory make: #room) location: 2@1.
    door := (aFactory make: #door) from: room1 to: room2.

    room1
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: door;
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: (aFactory make: #wall).
    room2
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: (aFactory make: #wall);
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: door.
    ^ Maze new
        addRoom: room1;
        addRoom: room2;
        yourself
```

Known Uses

Perhaps the first example of the Prototype pattern was in Ivan Sutherland's Sketchpad system [Sut63]. The first widely known application of the pattern in an object-oriented language was in ThingLab, where users could form a composite

object and then promote it to a prototype by installing it in a library of reusable objects [Bor81]. Goldberg and Robson mention prototypes as a pattern [GR83], but Coplien [Cop92] gives a much more complete description. He describes idioms related to the Prototype pattern for C++ and gives many examples and variations.

Etgdb is a debugger front-end based on ET++ that provides a point-and-click interface to different line-oriented debuggers. Each debugger has a corresponding DebuggerAdaptor subclass. For example, GdbAdaptor adapts etgdb to the command syntax of GNU gdb, while SunDbxAdaptor adapts etgdb to Sun's dbx debugger. Etgdb does not have a set of DebuggerAdaptor classes hard-coded into it. Instead, it reads the name of the adaptor to use from an environment variable, looks for a prototype with the specified name in a global table, and then clones the prototype. New debuggers can be added to etgdb by linking it with the DebuggerAdaptor that works for that debugger.

The "interaction technique library" in Mode Composer stores prototypes of objects that support various interaction techniques [Sha90]. Any interaction technique created by the Mode Composer can be used as a prototype by placing it in this library. The Prototype pattern lets Mode Composer support an unlimited set of interaction techniques.

The music editor example discussed earlier is based on the Unidraw drawing framework [VL90].

Related Patterns

Prototype and Abstract Factory (99) are competing patterns in some ways, as we discuss at the end of this chapter. They can also be used together, however. An Abstract Factory might store a set of prototypes from which to clone and return product objects.

Designs that make heavy use of the Composite (183) and Decorator (196) patterns often can benefit from Prototype as well.

---

[1]Such applications reflect the Composite (183) and Decorator (196) patterns.

Singleton

Intent

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

Motivation

It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

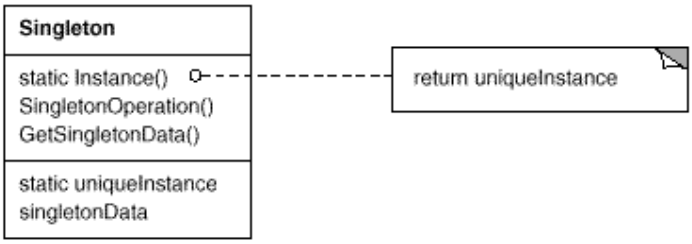
A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

Applicability

Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure



Participants

- **Singleton**
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
  - may be responsible for creating its own unique instance.

Collaborations

- Clients access a Singleton instance solely through Singleton's Instance operation.

Consequences

The Singleton pattern has several benefits:

1. *Controlled access to sole instance.* Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
2. *Reduced name space.* The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
3. *Permits refinement of operations and representation.* The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.
4. *Permits a variable number of instances.* The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to control the number of instances that the

application uses. Only the operation that grants access to the Singleton instance needs to change.

5. *More flexible than class operations.* Another way to package a singleton's functionality is to use class operations (that is, static member functions in C++ or class methods in Smalltalk). But both of these language techniques make it hard to change a design to allow more than one instance of a class. Moreover, static member functions in C++ are never virtual, so subclasses can't override them polymorphically.

Implementation

Here are implementation issues to consider when using the Singleton pattern:

1. *Ensuring a unique instance.* The Singleton pattern makes the sole instance a normal instance of a class, but that class is written so that only one instance can ever be created. A common way to do this is to hide the operation that creates the instance behind a class operation (that is, either a static member function or a class method) that guarantees only one instance is created. This operation has access to the variable that holds the unique instance, and it ensures the variable is initialized with the unique instance before returning its value. This approach ensures that a singleton is created and initialized before its first use.

   You can define the class operation in C++ with a static member function Instance of the Singleton class. Singleton also defines a static member variable _instance that contains a pointer to its unique instance.

   The Singleton class is declared as

```
class Singleton {
public:
   static Singleton* Instance();
protected:
   Singleton();
private:
   static Singleton* _instance;
};
```

   The corresponding implementation is

```
Singleton* Singleton::_instance = 0;
```

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Clients access the singleton exclusively through the Instance member function. The variable _instance is initialized to 0, and the static member function Instance returns its value, initializing it with the unique instance if it is 0. Instance uses lazy initialization; the value it returns isn't created and stored until it's first accessed.

Notice that the constructor is protected. A client that tries to instantiate Singleton directly will get an error at compile-time. This ensures that only one instance can ever get created.

Moreover, since the _instance is a pointer to a Singleton object, the Instance member function can assign a pointer to a subclass of Singleton to this variable. We'll give an example of this in the Sample Code.

There's another thing to note about the C++ implementation. It isn't enough to define the singleton as a global or static object and then rely on automatic initialization. There are three reasons for this:

1. We can't guarantee that only one instance of a static object will ever be declared.
2. We might not have enough information to instantiate every singleton at static initialization time. A singleton might require values that are computed later in the program's execution.
3. C++ doesn't define the order in which constructors for global objects are called across translation units [ES90]. This means that no dependencies can exist between singletons; if any do, then errors are inevitable.

An added (albeit small) liability of the global/static object approach is that it forces all singletons to be created whether they are used or not. Using a static member function avoids all of these problems.

In Smalltalk, the function that returns the unique instance is implemented as a class method on the Singleton class. To ensure that only one instance is created, override the new operation. The resulting Singleton class might have the following two class methods, where SoleInstance is a class variable that is not used anywhere else:

```
new
    self error: 'cannot create new object'

default
    SoleInstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

2. *Subclassing the Singleton class.* The main issue is not so much defining the subclass but installing its unique instance so that clients will be able to use it. In essence, the variable that refers to the singleton instance must get initialized with an instance of the subclass. The simplest technique is to determine which singleton you want to use in the Singleton's Instance operation. An example in the Sample Code shows how to implement this technique with environment variables.

   Another way to choose the subclass of Singleton is to take the implementation of Instance out of the parent class (e.g., MazeFactory) and put it in the subclass. That lets a C++ programmer decide the class of singleton at link-time (e.g., by linking in an object file containing a different implementation) but keeps it hidden from the clients of the singleton.

   The link approach fixes the choice of singleton class at link-time, which makes it hard to choose the singleton class at run-time. Using conditional statements to determine the subclass is more flexible, but it hard-wires the set of possible Singleton classes. Neither approach is flexible enough in all cases.

   A more flexible approach uses a **registry of singletons**. Instead of having Instance define the set of possible Singleton classes, the Singleton classes can register their singleton instance by name in a well-known registry.

   The registry maps between string names and singletons. When Instance needs a singleton, it consults the registry, asking for the singleton by name. The registry looks up the corresponding singleton (if it exists) and returns it. This approach frees Instance from knowing all possible Singleton classes or instances. All it requires is a common interface for all Singleton classes that includes operations for the registry:

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
```

```
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

Register registers the Singleton instance under the given name. To keep the registry simple, we'll have it store a list of NameSingletonPair objects. Each NameSingletonPair maps a name to a singleton. The Lookup operation finds a singleton given its name. We'll assume that an environment variable specifies the name of the singleton desired.

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup

        _instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}
```

Where do Singleton classes register themselves? One possibility is in their constructor. For example, a MySingleton subclass could do the following:

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

Of course, the constructor won't get called unless someone instantiates the class, which echoes the problem the Singleton pattern is trying to solve! We can get around this problem in C++ by defining a static instance of MySingleton. For example, we can define

```
static MySingleton theSingleton;
```

in the file that contains MySingleton's implementation.

No longer is the Singleton class responsible for creating the singleton. Instead, its primary responsibility is to make the singleton object of

choice accessible in the system. The static object approach still has a potential drawback—namely that instances of all possible Singleton subclasses must be created, or else they won't get registered.

Sample Code

Suppose we define a MazeFactory class for building mazes as described on page 92. MazeFactory defines an interface for building different parts of a maze. Subclasses can redefine the operations to return instances of specialized product classes, like BombedWall objects instead of plain Wall objects.

What's relevant here is that the Maze application needs only one instance of a maze factory, and that instance should be available to code that builds any part of the maze. This is where the Singleton pattern comes in. By making the MazeFactory a singleton, we make the maze object globally accessible without resorting to global variables.

For simplicity, let's assume we'll never subclass MazeFactory. (We'll consider the alternative in a moment.) We make it a Singleton class in C++ by adding a static Instance operation and a static _instance member to hold the one and only instance. We must also protect the constructor to prevent accidental instantiation, which might lead to more than one instance.

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

The corresponding implementation is

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
```

```
    return _instance;
}
```

Now let's consider what happens when there are subclasses of MazeFactory, and the application must decide which one to use. We'll select the kind of maze through an environment variable and add code that instantiates the proper MazeFactory subclass based on the environment variable's value. The Instance operation is a good place to put this code, because it already instantiates MazeFactory:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ... other possible subclasses

        } else {       // default
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

Note that Instance must be modified whenever you define a new subclass of MazeFactory. That might not be a problem in this application, but it might be for abstract factories defined in a framework.

A possible solution would be to use the registry approach described in the Implementation section. Dynamic linking could be useful here as well—it would keep the application from having to load all the subclasses that are not used.

Known Uses

An example of the Singleton pattern in Smalltalk-80 [Par90] is the set of changes to the code, which is ChangeSet current. A more subtle example is the relationship between classes and their metaclasses. A metaclass is the class of a class, and each metaclass has one instance. Metaclasses do not have names (except indirectly

through their sole instance), but they keep track of their sole instance and will not normally create another.

The InterViews user interface toolkit [LCI+92] uses the Singleton pattern to access the unique instance of its Session and WidgetKit classes, among others. Session defines the application's main event dispatch loop, stores the user's database of stylistic preferences, and manages connections to one or more physical displays. WidgetKit is an Abstract Factory (99) for defining the look and feel of user interface widgets. The WidgetKit::instance() operation determines the particular WidgetKit subclass that's instantiated based on an environment variable that Session defines. A similar operation on Session determines whether monochrome or color displays are supported and configures the singleton Session instance accordingly.

Related Patterns

Many patterns can be implemented using the Singleton pattern. See Abstract Factory (99), Builder (110), and Prototype (133).

Discussion of Creational Patterns

There are two common ways to parameterize a system by the classes ofobjects it creates. One way is to subclass the class that creates theobjects; this corresponds to using the Factory Method (121) pattern. The main drawback of thisapproach is that it can require creating a new subclass just to changethe class of the product. Such changes can cascade. For example,when the product creator is itself created by a factory method, thenyou have to override its creator as well.

The other way to parameterize a system relies more on objectcomposition: Define an object that's responsible for knowing the classof the product objects, and make it a parameter of the system. Thisis a key aspect of the Abstract Factory (99),Builder (110), and Prototype (133) patterns. All three involve creating a new "factory object" whoseresponsibility is to create product objects. Abstract Factory has thefactory object producing objects of several classes. Builder has thefactory object building a complex product incrementally using acorrespondingly complex protocol. Prototype has the factory objectbuilding a product by copying a prototype object. In this case, thefactory object and the prototype are the same object, because theprototype is responsible for returning the product.

Consider the drawing editor framework described in the Prototypepattern. There are several ways to parameterize a GraphicTool by theclass of product:

- By applying the Factory Method pattern, a subclass of GraphicTool will becreated for each subclass of Graphic in the palette. GraphicTool willhave a NewGraphic operation that each GraphicTool subclass willredefine.
- By applying the Abstract Factory pattern, there will be a class hierarchyof GraphicsFactories, one for each Graphic subclass. Each factorycreates just one product in this case: CircleFactory will createCircles, LineFactory will create Lines, and so on. A GraphicTool willbe parameterized with a factory for creating the appropriate kind ofGraphics.
- By applying the Prototype pattern, each subclass of Graphics willimplement the Clone operation, and a GraphicTool will be parameterizedwith a prototype of the Graphic it creates.

Which pattern is best depends on many factors. In our drawing editorframework, the Factory Method pattern is easiest to use at first.It's easy to define a new subclass of GraphicTool, and the instancesof GraphicTool are created only when the palette is defined. The maindisadvantage here is that GraphicTool subclasses proliferate, and noneof them does very much.

Abstract Factory doesn't offer much of an improvement, because itrequires an equally large GraphicsFactory class hierarchy. AbstractFactory would be preferable to Factory Method only if there werealready a GraphicsFactory class

hierarchy—either because thecompiler provides it automatically (as in Smalltalk or Objective C) orbecause it's needed in another part of the system.

Overall, the Prototype pattern is probably the best for the drawingeditor framework, because it only requires implementing a Cloneoperation on each Graphics class. That reduces the number of classes,and Clone can be used for purposes other than pure instantiation (e.g.,a Duplicate menu operation).

Factory Method makes a design more customizable and only a little morecomplicated. Other design patterns require new classes, whereasFactory Method only requires a new operation. People often useFactory Method as the standard way to create objects, but it isn'tnecessary when the class that's instantiated never changes or wheninstantiation takes place in an operation that subclasses can easilyoverride, such as an initialization operation.

Designs that use Abstract Factory, Prototype, or Builder are even moreflexible than those that use Factory Method, but they're also morecomplex. Often, designs start out using Factory Method and evolvetoward the other creational patterns as the designer discovers wheremore flexibility is needed. Knowing many design patterns gives youmore choices when trading off one design criterion against another.

4. Structural Patterns

Structural patterns are concerned with how classes and objects arecomposed to form larger structures.Structural *class* patterns use inheritance to compose interfacesor implementations. As a simple example, consider how multipleinheritance mixes two or more classes into one. The result is a classthat combines the properties of its parent classes. This pattern isparticularly useful for making independently developed class librarieswork together. Another example is the class form of the Adapter (157) pattern. In general, an adapter makes oneinterface (the adaptee's) conform to another, thereby providing auniform abstraction of different interfaces. A class adapteraccomplishes this by inheriting privately from an adaptee class. Theadapter then expresses its interface in terms of the adaptee's.

Rather than composing interfaces or implementations, structural *object* patterns describe ways to compose objects to realize newfunctionality. The added flexibility of object composition comes fromthe ability to change the composition at run-time, which is impossiblewith static class composition.

Composite (183) is an example of a structural objectpattern. It describes how to build a class hierarchy made up ofclasses for two kinds of objects: primitive and composite. Thecomposite objects let you compose primitive and other compositeobjects into arbitrarily complex structures. In the Proxy (233) pattern, a proxy acts as a convenientsurrogate or placeholder for another object. A proxy can be used inmany ways. It can act as a local representative for an object in aremote address space. It can represent a large object that should beloaded on demand. It might protect access to a sensitive object.Proxies provide a level of indirection to specific properties ofobjects. Hence they can restrict, enhance, or alter these properties.

The Flyweight (218) pattern defines a structure forsharing objects. Objects are shared for at least two reasons:efficiency and consistency. Flyweight focuses on sharing for spaceefficiency. Applications that use lots of objects must pay carefulattention to the cost of each object. Substantial savings can be hadby sharing objects instead of replicating them. But objects can beshared only if they don't define context-dependent state. Flyweightobjects have no such state. Any additional information they need toperform their task is passed to them when needed. With nocontext-dependent state, Flyweight objects may be shared freely.

Whereas Flyweight shows how to make lots of little objects, Facade (208) shows how to make a single object representan entire subsystem. A facade is a representative for a set ofobjects. The facade carries out its responsibilities by forwardingmessages to the objects it represents. The Bridge (171) pattern

separates an object's abstraction from its implementation sothat you can vary them independently.

Decorator (196) describes how to add responsibilitiesto objects dynamically. Decorator is a structural pattern thatcomposes objects recursively to allow an open-ended number ofadditional responsibilities. For example, a Decorator objectcontaining a user interface component can add a decoration like aborder or shadow to the component, or it can add functionality likescrolling and zooming. We can add two decorations simply by nestingone Decorator object within another, and so on for additionaldecorations. To accomplish this, each Decorator object must conformto the interface of its component and must forward messages to it.The Decorator can do its job (such as drawing a border around thecomponent) either before or after forwarding a message.

Many structural patterns are related to some degree. We'll discussthese relationships at the end of the chapter.