Behavioral Patterns

Behavioral patterns are concerned with algorithms and theassignment of responsibilities between objects. Behavioral patternsdescribe not just patterns of objects or classes but also the patternsof communication between them. These patterns characterize complexcontrol flow that's difficult to follow at run-time. They shift yourfocus away from flow of control to let you concentrate just on the wayobjects are interconnected.

Behavioral class patterns use inheritance to distribute behaviorbetween classes. This chapter includes two such patterns. Template Method is the simpler and more common ofthe two. A template method is an abstract definition of an algorithm.It defines the algorithm step by step. Each step invokes either anabstract operation or a primitive operation. A subclass fleshes outthe algorithm by defining the abstract operations. The otherbehavioral class pattern is Interpreter, whichrepresents a grammar as a class hierarchy and implements aninterpreter as an operation on instances of these classes.

Behavioral object patterns use object composition rather thaninheritance. Some describe how a group of peer objects cooperate toperform a task that no single object can carry out by itself. Animportant issue here is how peer objects know about each other. Peerscould maintain explicit references to each other, but that wouldincrease their coupling. In the extreme, every object would knowabout every other. The Mediator pattern avoidsthis by introducing a mediator object between peers. The mediatorprovides the indirection needed for loose coupling.

Chain of Responsibility provides even loosercoupling. It lets you send requests to an object implicitly through achain of candidate objects. Any candidate may fulfill the requestdepending on run-time conditions. The number of candidates isopen-ended, and you can select which candidates participate in thechain at run-time.

The Observer pattern defines and maintains adependency between objects. The classic example of Observer is inSmalltalk Model/View/Controller, where all views of the model are notified whenever themodel's state changes.

Other behavioral object patterns are concerned with encapsulatingbehavior in an object and delegating requests to it. The Strategy pattern encapsulates an algorithm in anobject. Strategy makes it easy to specify and change the algorithm an object uses. The Command pattern encapsulates arequest in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways. The State pattern encapsulates the states of an object so that the object can change its behavior when its state object changes. Visitor encapsulates behavior that wouldotherwise be distributed across classes,and Iterator abstracts the way you access andtraverse objects in an aggregate.

Chain of Responsibility

- **Intent**

Avoid coupling the sender of a request to its receiver by giving morethan one object a chance to handle the request. Chain the receivingobjects and pass the request along the chain until an objecthandles it.

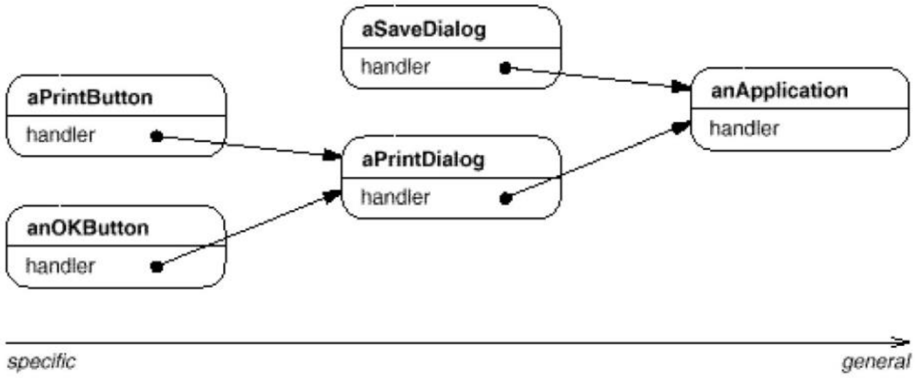
- **Motivation**

Consider a context-sensitive help facility for a graphical userinterface. The user can obtain help information on any part of theinterface just by clicking on it. The help that's provided depends onthe part of the interface that's selected and its context; forexample, a button widget in a dialog box might have different helpinformation than a similar button in the main window. If no specifichelp information exists for that part of the interface, thenthe help system should display a more general help message about theimmediate context the dialog box as a whole, for example.

Hence it's natural to organize help information according to itsgenerality from the most specific to the most general. Furthermore,it's clear that a help request is handled by one of several userinterface objects; which one depends on the context and how specificthe available help is.

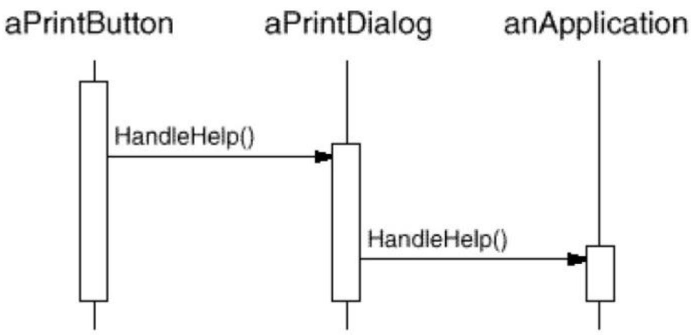
The problem here is that the object that ultimately *provides* thehelp isn't known explicitly to the object (e.g., the button) that *initiates* the help request. What we need is a way to decouple thebutton that initiates the help request from the objects that mightprovide help information. The Chain of Responsibility pattern defineshow that happens.

The idea of this pattern is to decouple senders and receivers bygiving multiple objects a chance to handle a request. The requestgets passed along a chain of objects until one of them handles it.



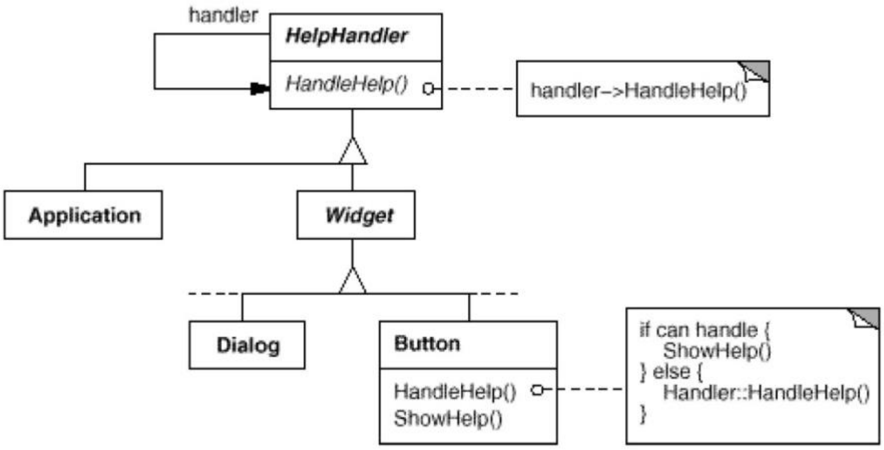
The first object in the chain receives the request and either handlesit or forwards it to the next candidate on the chain, which doeslikewise. The object that made the request has no explicit knowledgeof who will handle it we say the request has an implicitreceiver.

Let's assume the user clicks for help on a button widget marked"Print." The button is contained in an instance of PrintDialog,which knows the application object it belongs to (see preceding object diagram).The following interaction diagram illustrates how the helprequest gets forwarded along the chain:



In this case, neither aPrintButton nor aPrintDialog handles therequest; it stops at anApplication, which can handle it or ignore it.The client that issued the request has no direct reference to theobject that ultimately fulfills it.

To forward the request along the chain, and to ensure receivers remainimplicit, each object on the chain shares a common interface forhandling requests and for accessing its successor on thechain. For example, the help system might define a HelpHandler classwith a corresponding HandleHelp operation. HelpHandler can be theparent class for candidate object classes, or it can be defined as amixin class. Then classes that want to handle help requests can makeHelpHandler a parent:



The Button, Dialog, and Application classes use HelpHandler operationsto handle help requests. HelpHandler's HandleHelp operation forwardsthe request to the successor by default. Subclasses can override thisoperation to provide help under the right circumstances; otherwisethey can use the default implementation to forward the request.

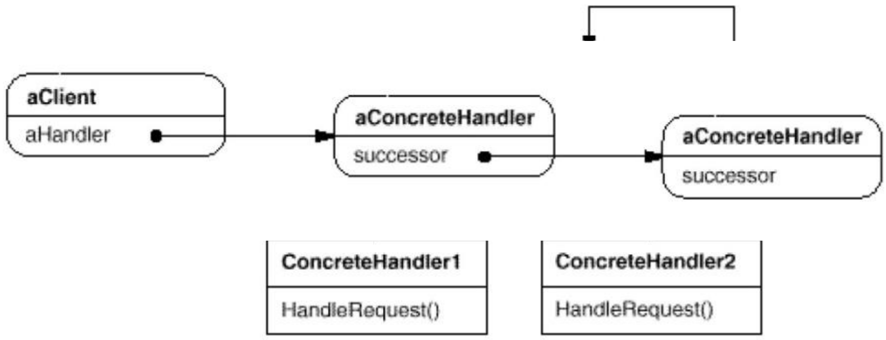
Applicability

Use Chain of Responsibility when

- more than one object may handle a request, and the handler isn't known*apriori*. The handler should be ascertained automatically.
- you want to issue a request to one of several objects withoutspecifying the receiver explicitly.
- the set of objects that can handle a request should be specifieddynamically.

Structure

A typical object structure might look like this:



Participants

Handler (HelpHandler)

- defines an interface for handling requests.
- (optional) implements the successor link.

ConcreteHandler (PrintButton, PrintDialog)

- handles requests it is responsible for.
- can access its successor.
- if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.

Client

- initiates the request to a ConcreteHandler object on the chain.

Collaborations

When a client issues a request, the request propagates along the chainuntil a ConcreteHandler object takes responsibility for handling it.

Consequences

Chain of Responsibility has the following benefits and liabilities:

*Reduced coupling.*The pattern frees an object from knowing which other objecthandles arequest. An object only has to know that a request will be handled"appropriately." Both the receiver and the sender have no explicitknowledge of each other, and an object in the chain doesn't have toknow about the chain's structure.

As a result, Chain of Responsibility can simplify objectinterconnections. Instead of objects maintaining references to allcandidate receivers, they keep a single reference to their successor.

*Added flexibility in assigning responsibilities to objects.*Chain ofResponsibility gives you added flexibility in distributingresponsibilities among objects. You can add or changeresponsibilities for handling a request by adding to or otherwisechanging the chain at run-time. You can combine this with subclassingto specialize handlers statically.

*Receipt isn't guaranteed.*Since a request has no explicit receiver, there'sno *guarantee*it'll be handled the request can fall off the end of the chainwithout ever being handled. A request can also go unhandled when thechain is not configured properly.

Implementation

Here are implementation issues to consider in Chain of Responsibility:

*Implementing the successor chain.*There are two possible ways to implementthe successor chain:

Define new links (usually in the Handler, but ConcreteHandlerscould define them instead).

Use existing links.

Our examples so far define new links, but often you can use existingobject references to form the successor chain. For example, parentreferences in a part-whole hierarchy can define a part's successor. Awidget structure might already have such links. Composite discusses parent references in more detail.

Using existing links works well when the links support the chain youneed. It saves you from defining links explicitly, and it savesspace. But if the structure doesn't reflect the chain ofresponsibility your application requires, then you'll have to define redundant links.

*Connecting successors.*If there are no preexisting references for defininga chain, then you'llhave to introduce them yourself. In that case, the Handler not only defines the interface for the requests but usually maintains thesuccessor as well. That lets the handler provide a defaultimplementation of HandleRequest that forwards the request to thesuccessor (if any). If a ConcreteHandler subclass isn't interestedin the request, it doesn't have to override the forwarding operation,since its default implementation forwards unconditionally.

Here's a HelpHandler base class that maintains a successor link:

Sample Code

The following example illustrates how a chain of responsibility canhandle requests for an on-line help system like the one describedearlier. The help request is an explicit operation. We'll use existingparent references in the widget hierarchy to propagate requestsbetween widgets in the chain, and we'll define a reference in theHandler class to propagate help requests between nonwidgets in thechain.

The HelpHandler class defines the interface for handlinghelp requests. It maintains a help topic (which is empty by default)and keeps a reference to its successor on the chain of help handlers.The key operation is HandleHelp, which subclassesoverride. HasHelp is a convenience operation for checkingwhether there is an associated help topic.

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;

class HelpHandler { public:
HelpHandler(HelpHandler*    =    0,    Topic    =
NO_HELP_TOPIC); virtual bool HasHelp();
virtual  void  SetHandler(HelpHandler*,  Topic);
virtual void HandleHelp();
private:
HelpHandler* _successor;
```

```
Topic _topic;
};

HelpHandler::HelpHandler (
HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }

bool HelpHandler::HasHelp () { return
_topic != NO_HELP_TOPIC;
}

void HelpHandler::HandleHelp () { if
(_successor != 0) { successor-
>HandleHelp();
}
}
```

Known Uses

Several class libraries use the Chain of Responsibility pattern tohandle user events. They use different names for the Handler class,but the idea is the same: When the user clicks the mouse or presses akey, an event gets generated and passed along the chain.MacApp [App89] and ET++ [WGM88] call it "EventHandler,"Symantec's TCL library [Sym93b] calls it "Bureaucrat," andNeXT's AppKit [Add94] uses the name "Responder."

The Unidraw framework for graphical editors defines Command objectsthat encapsulate requests to Component and ComponentViewobjects [VL90]. Commands are requests in the sensethat a component or component view may interpret a command to performan operation. This corresponds to the "requests as objects"approach described in Implementation. Components and component viewsmay be structured hierarchically. A component or a component view mayforward command interpretation to its parent, which may in turnforward it to its parent, and so on, thereby forming a chain ofresponsibility.

ET++ uses Chain of Responsibility to handle graphical update. Agraphical object calls the InvalidateRect operation whenever it mustupdate a part of its appearance. A graphical object can't handleInvalidateRect by itself, because it doesn't know enough about itscontext. For example, a graphical object can be enclosed in objectslike Scrollers or Zoomers that transform its coordinate system. Thatmeans the object might be scrolled or zoomed so that it's partiallyout of view. Therefore the default implementation of InvalidateRectforwards the request to the enclosing container object. The lastobject in the forwarding chain is a Window instance. By the timeWindow receives the request, the invalidation rectangle is guaranteedto be transformed properly. The Window handles InvalidateRect bynotifying the window system interface and requesting an update.

Related Patterns

Chain of Responsibility is often applied in conjunction with Composite (183). There, a component's parent can act as its successor.

Command

Intent

Encapsulate a request as an object, thereby letting you parameterizeclients with different requests, queue or log requests, and supportundoable operations.

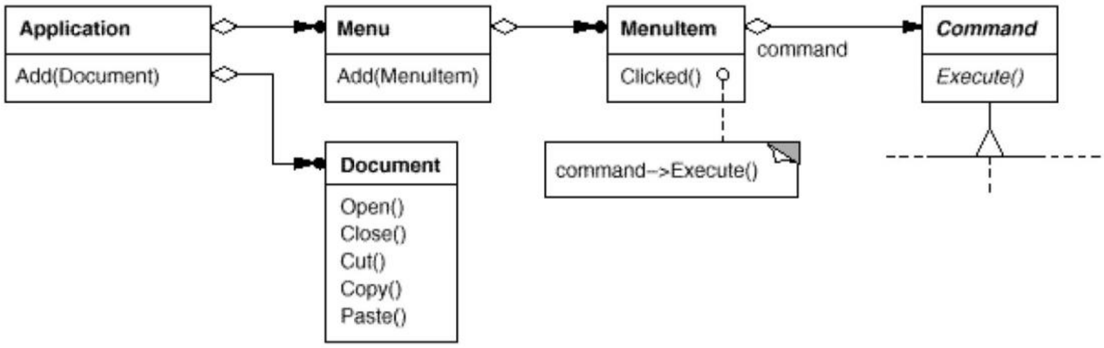
Also Known As

Action, Transaction

Motivation

Sometimes it's necessary to issue requests to objects without knowinganything about the operation being requested or the receiver of therequest. For example, user interface toolkits include objects likebuttons and menus that carry out a request in response to user input.But the toolkit can't implement the request explicitly in the buttonor menu, because only applications that use the toolkit know whatshould be done on which object. As toolkit designers we have no wayof knowing the receiver of the request or the operations that willcarry it out.

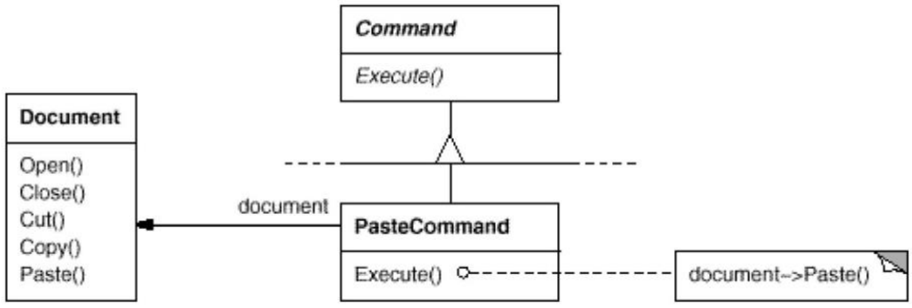
The Command pattern lets toolkit objects make requests of unspecifiedapplication objects by turning the request itself into an object. Thisobject can be stored and passed around like other objects. The key tothis pattern is an abstract Command class, which declares an interfacefor executing operations. In the simplest form this interfaceincludes an abstract Execute operation.Concrete Command subclassesspecify a receiver-action pair by storing the receiver as an instancevariable and by implementing Execute to invoke the request. Thereceiver has the knowledge required to carry out the request.



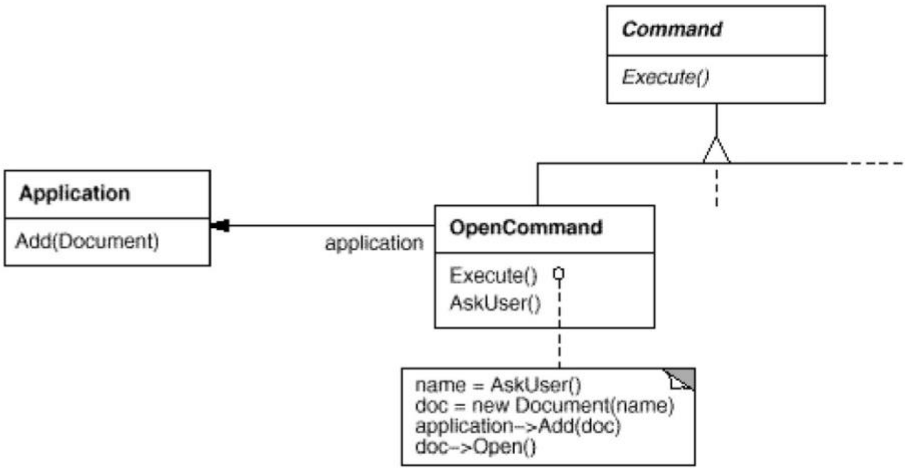
Menus can be implemented easily with Command objects. Each choice ina Menu is an instance of a MenuItem class. An Application class createsthese menus and their menu items along with the rest of the user interface.The Application class also keeps track of Document objects that a user hasopened.

The application configures each MenuItem with an instance of aconcrete Command subclass. When the user selects a MenuItem, theMenuItem calls Execute on its command, and Execute carries out theoperation. MenuItems don't know which subclass of Command they use.Command subclasses store the receiver of the request and invoke one ormore operations on the receiver.

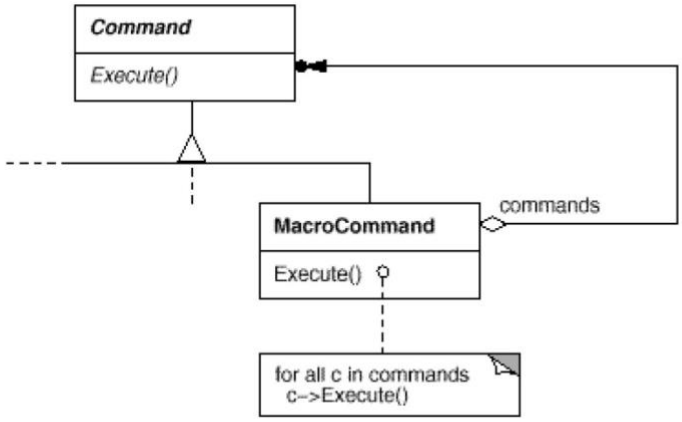
For example, PasteCommand supports pasting text from the clipboardinto a Document. PasteCommand's receiver is the Document object it issupplied upon instantiation. The Execute operation invokes Paste onthe receiving Document.



OpenCommand's Execute operation is different: it prompts the userfor a document name, creates a corresponding Document object, adds thedocument to the receiving application, and opens the document.



Sometimes a MenuItem needs to execute a *sequence* of commands.For example, a MenuItem for centering a page at normal size could beconstructed from a CenterDocumentCommand object and aNormalSizeCommand object. Because it's common to string commandstogether in this way, we can define a MacroCommand class to allow aMenuItem to execute an open-ended number of commands. MacroCommand isa concrete Command subclass that simply executes a sequence ofCommands. MacroCommand has no explicit receiver, because the commandsit sequences define their own receiver.



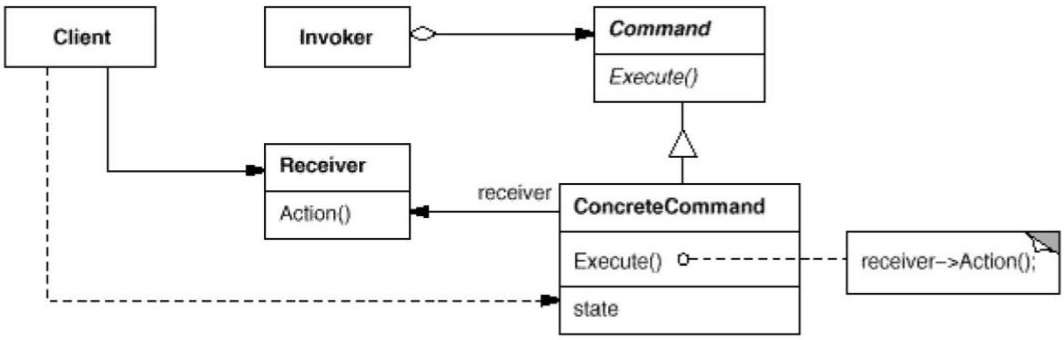
In each of these examples, notice how the Command pattern decouplesthe object that invokes the operation from the one having theknowledge to perform it. This gives us a lot of flexibility indesigning our user interface. An application can provide both a menuand a push button interface to a feature just by making the menu andthe push button share an instance of the same concrete Command subclass.We can replace commands dynamically, which would be useful forimplementing context-sensitive menus. We can also support commandscripting by composing commands into larger ones. All of this ispossible because the object that issues a request only needs to knowhow to issue it; it doesn't need to know how the request will be carried out.

Applicability

Use the Command pattern when you want to parameterize objects by an action to perform, as MenuItem objects did above. You can express such parameterization in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for callbacks. specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there. support undo. The Command's Execute operation can store state for reversing its effects in the command itself. The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute. Executed commands are stored in a history list. Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling Unexecute and Execute, respectively. support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes. Recovering from a crash involves reloading logged commands from disk and reexecuting them with the Execute operation.

structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support transactions. A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions.

Structure



Participants

Command

declares an interface for executing an operation.

ConcreteCommand (PasteCommand, OpenCommand)

o defines a binding between a Receiver object and an action.

implements Execute by invoking the corresponding operation(s) on Receiver.

Client (Application)

creates a ConcreteCommand object and sets its receiver.

Invoker (MenuItem)

asks the command to carry out the request.

Receiver (Document, Application)

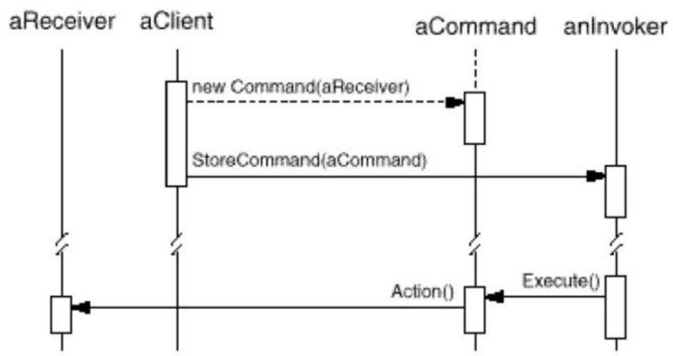
knows how to perform the operations associated with carrying out a request.

Any class may serve as a Receiver.

Collaborations

The client creates a ConcreteCommand object and specifies its receiver.
An Invoker object stores the ConcreteCommand object.
The invoker issues a request by calling Execute on the command. Whencommands are undoable, ConcreteCommand stores state for undoing thecommand prior to invoking Execute.
The ConcreteCommand object invokes operations on its receiver to carryout the request.

The following diagram shows the interactions between these objects.It illustrates how Command decouples the invoker from the receiver(and the request it carries out).



Consequences

The Command pattern has the following consequences:

Command decouples the object that invokes the operation from the onethat knows how to perform it.
Commands are first-class objects. They can be manipulated and extendedlike any other object.
You can assemble commands into a composite command. An example is theMacroCommand class described earlier. In general, composite commandsare an instance of the Composite (183) pattern.
It's easy to add new Commands, because you don't have to changeexisting classes.

Implementation

Consider the following issues when implementing the Command pattern:

- *How intelligent should a command be?*A command can have a wide range ofabilities. At one extreme itmerely defines a binding between a receiver and the actions that carryout the request. At the other extreme it implements everything itselfwithout delegating to a receiver at all. The latter extreme is usefulwhen you want to define commands that are independent of existingclasses, when no suitable receiver exists, or when a command knows itsreceiver implicitly. For example, a command that creates anotherapplication window may be just as capable of creating the window asany other object. Somewhere in between these extremes are commandsthat have enough knowledge to find their receiver dynamically.
- *Supporting undo and redo.*Commands can support undo and redo capabilitiesif they provide a wayto reverse their execution (e.g., an Unexecute or Undo operation). AConcreteCommand class might need to store additional state to do so. Thisstate can include
  - o the Receiver object, which actually carries out operations inresponse to the request,
  - o the arguments to the operation performed on the receiver, and
  - o any original values in the receiver that can changeas a result of handling the request. The receiver must provideoperations that let the command return the receiver to its prior state.

  To support one level of undo, an application needs to store only thecommand that was executed last. For multiple-level undo and redo, theapplication needs a history list of commands that havebeen executed, where the maximum length of the list determines thenumber of undo/redo levels. The history list stores sequences ofcommands that have been executed. Traversing backward through thelist and reverse-executing commands cancels their effect; traversingforward and executing commands reexecutes them.

  An undoable command might have to be copied before it can be placed onthe history list. That's because the command object that carried outthe original request, say, from a MenuItem, will perform otherrequests at later times. Copying is required to distinguish differentinvocations of the same command if its state can vary acrossinvocations.

For example, a DeleteCommand that deletes selected objects must storedifferent sets of objects each time it's executed. Therefore theDeleteCommand object must be copied following execution, and the copyis placed on the history list. If the command's state never changeson execution, then copying is not required only a reference to thecommand need be placed on the history list. Commands that must becopied before being placed on the history list act as prototypes (see Prototype (133)).

*Avoiding error accumulation in the undo process.*Hysteresis can be a problemin ensuring a reliable,semantics-preserving undo/redo mechanism. Errors can accumulate ascommands are executed, unexecuted, and reexecuted repeatedly so thatan application's state eventually diverges from original values. Itmay be necessary therefore to store more information in the command toensure that objects are restored to their original state. The Memento (316) pattern can be applied to give the commandaccess to this information without exposing the internals of otherobjects.

*Using C++ templates.*For commands that (1) aren't undoable and (2) don'trequire arguments,we can use C++ templates to avoid creating a Command subclass forevery kind of action and receiver. We show how to do this in the SampleCode section.

Sample Code

The C++ code shown here sketches the implementation of the Command classesin the Motivation section. We'll define OpenCommand,PasteCommand, and MacroCommand. First theabstract Command class:

```
class Command {
public:
```

```
virtual ~Command();
virtual void Execute() = 0;
protected:
Command();
};
```

OpenCommand opens a document whose name is supplied by theuser. An OpenCommand must be passed anApplication object in its constructor. AskUser is animplementation routine that prompts the user for the name of thedocument to open.

```
class OpenCommand : public Command {
public:
OpenCommand(Application*);
virtual void Execute(); protected:
virtual const char* AskUser(); private:
Application* _application; char*
_response;
};

OpenCommand::OpenCommand (Application* a)
{ _application = a;
}

void OpenCommand::Execute () {
const char* name = AskUser(); if
(name != 0) {
Document* document = new Document(name);
_application->Add(document); document-
>Open();
}
}
```

A PasteCommand must be passed a Document object asits receiver. The receiver is given as a parameter to PasteCommand'sconstructor.

```
class PasteCommand : public Command {
public:
PasteCommand(Document*);
virtual void Execute(); private:
Document* _document; };
```

```
PasteCommand::PasteC  (Document*  doc)                    voi
ommand                {            _document = doc;  }    d
PasteCommand::Execut      _document-
e () {                    >Paste();       }
```

For simple commands that aren't undoable and don't require arguments,we can use a class template to parameterize the command's receiver.We'll define a template subclass SimpleCommand for suchcommands. SimpleCommand is parameterized by theReceiver type and maintains a binding between a receiver objectand an action stored as a pointer to a member function.

```
template<class Receiver>
class SimpleCommand : public Command {
public:
typedef void (Receiver::* Action)();
SimpleCommand(Receiver* r, Action a) :
_receiver(r), _action(a) { }
virtual void Execute(); private:
Action _action; Receiver*
_receiver; };
```

The constructor stores the receiver and the action in the correspondinginstance variables. Execute simply applies the action to thereceiver.

```
template<class Receiver>
```

```
void SimpleCommand&ltReceiver>::Execute () {                    (_receiver-
>*_action)();                                    }
```

To create a command that calls Actionon an instance of class MyClass, a client simply writes

```
MyClass* receiver = new MyClass;
// ...
Command* aCommand =
new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// ...
aCommand->Execute();
```

Keep in mind that this solution only works for simple commands. More complex commands that keep track of not only their receivers but also arguments and/or undo state require a Command subclass.

A MacroCommand manages a sequence of subcommands and providesoperations for adding and removing subcommands. No explicit receiveris required, because the subcommands already define their receiver.

```
class MacroCommand : public Command {
public:
MacroCommand();
virtual ~MacroCommand(); virtual
void Add(Command*); virtual void
Remove(Command*); virtual void
Execute(); private:
List<Command*>* _cmds;
};
```

The key to the MacroCommand is its Execute memberfunction. This traverses all the subcommands and performsExecute on each of them.

```
void MacroCommand::Execute () {
ListIterator<Command*> i(_cmds);
```

```
for (i.First(); !i.IsDone(); i.Next()) { Command*
c = i.CurrentItem(); c->Execute();
}
}
```

Note that should the MacroCommand implement anUnexecute operation, then its subcommands must beunexecuted in *reverse* order relative to Execute'simplementation.

Finally, MacroCommand must provide operations to manage itssubcommands. The MacroCommand is also responsible fordeleting its subcommands.

```
void MacroCommand::Add (Command* c) { _cmds->Append(c); }
void MacroCommand::Remove (Command* c) { _cmds->Remove(c); }
```

Known Uses

Perhaps the first example of the Command pattern appears in a paper byLieberman [Lie85]. MacApp [App89] popularizedthe notion of commands for implementing undoable operations.ET++ [WGM88], InterViews [LCI+92], andUnidraw [VL90] also define classes that follow theCommand pattern. InterViews defines an Action abstract class thatprovides command functionality. It also defines an ActionCallbacktemplate, parameterized by action method, that can instantiate commandsubclasses automatically.

The THINK class library [Sym93b] also uses commands to supportundoable actions. Commands in THINK are called "Tasks." Taskobjects are passed along a Chain of Responsibility (251) for consumption.

Unidraw's command objects are unique in that they can behave likemessages. A Unidraw command may be sent to another object forinterpretation, and the result of the interpration varies with thereceiving object. Moreover, the receiver may delegate theinterpretation to another object, typically the receiver's parent in alarger structure as in a Chain of Responsibility. The receiver of aUnidraw command is thus computed rather than stored. Unidraw'sinterpretation mechanism depends on run-time type information.

Coplien describes how to implement functors, objects thatare functions, in C++ [Cop92]. He achieves a degree oftransparency in their use by overloading the function call operator(operator()). The Command pattern is different; its focusis on maintaining a *binding between* a receiver and a function(i.e., action), not just maintaining a function.

Related Patterns

A Composite can be used to implement MacroCommands.

A Memento can keep state the command requires to undo its effect.

A command that must be copied before being placed on the historylist acts as a Prototype

Interpreter

Intent

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

Motivation

If a particular kind of problem occurs often enough, then it might beworthwhile to express instances of the problem as sentences in asimple language. Then you can build an interpreter that solves theproblem by interpreting these sentences.

For example, searching for strings that match a pattern is a commonproblem. Regular expressions are a standard language for specifyingpatterns of strings. Rather than building custom algorithms to matcheach pattern against strings, search algorithms could interpret aregular expression that specifies a set of strings to match.

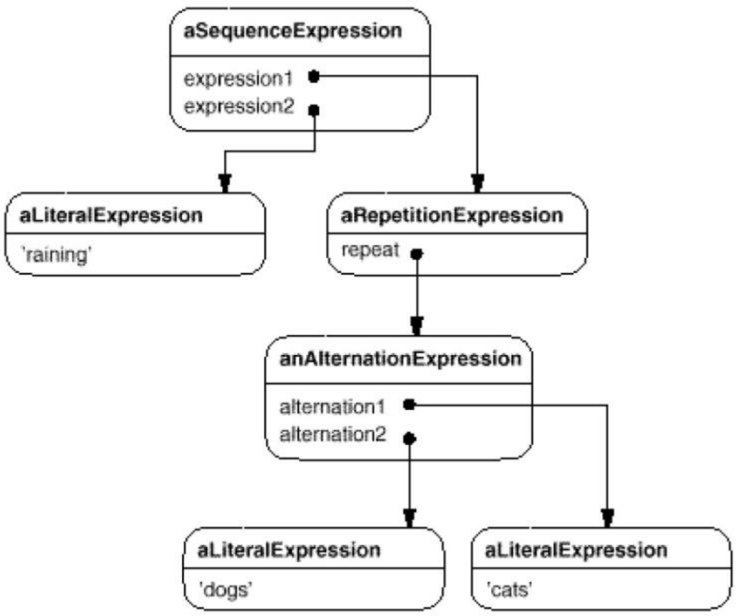
The Interpreter pattern describes how to define a grammar for simplelanguages, represent sentences in the language, and interpret thesesentences. In this example, the pattern describes how to define agrammar for regular expressions, represent a particular regularexpression, and how to interpret that regular expression.

Suppose the following grammar defines the regular expressions:

expression ::= literal | alternation | sequence | repetition | '('
expression ')'
alternation ::= expression '|' expression sequence
::= expression '&' expression repetition ::=
expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

The symbol expression is the start symbol, and literalis a terminal symbol defining simple words.

The Interpreter pattern uses a class to represent each grammar rule.Symbols on the right-hand side of the rule are instance variables ofthese classes. The grammar above is represented by five classes: anabstract class RegularExpression and its four subclassesLiteralExpression, AlternationExpression, SequenceExpression, andRepetitionExpression. The last three classes define variables thathold subexpressions. Every regular expression defined by this grammar is represented by anabstract syntax tree made up of instances of these classes. Forexample, the abstract syntax tree



represents the regular expression

raining& (dogs | cats) *

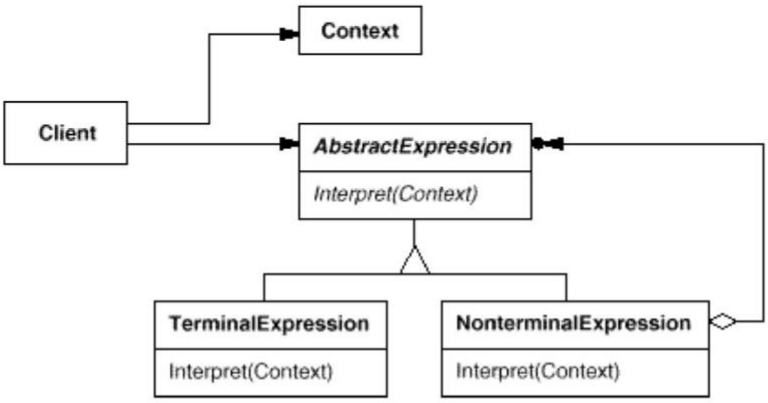
We can create an interpreter for these regular expressions by definingthe Interpret operation on each subclass of RegularExpression.Interpret takes as an argument

the context in which to interpret theexpression. The context contains the input string and information onhow much of it has been matched so far. Each subclass ofRegularExpression implements Interpret to match the next part of theinput string based on the current context. For example, LiteralExpression will check if the input matches the literal itdefines,AlternationExpression will check if the input matches any of itsalternatives, RepetitionExpression will check if the input has multiple copies ofexpression it repeats, and so on.

Applicability

Use the Interpreter pattern when there is a language to interpret, andyou can represent statements in the language as abstract syntax trees.The Interpreter pattern works best when the grammar is simple. For complex grammars, the class hierarchy forthe grammar becomes large and unmanageable. Tools such as parsergenerators are a better alternative in such cases. They can interpretexpressions without building abstract syntax trees, which can savespace and possibly time. efficiency is not a critical concern. The most efficient interpretersare usually *not* implemented by interpreting parse trees directlybut by first translating them into another form. For example, regularexpressions are often transformed into state machines. But even then,the *translator* can be implemented by the Interpreter pattern, sothe pattern is still applicable.

Structure



Participants

AbstractExpression (RegularExpression)

declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

TerminalExpression (LiteralExpression)

implements an Interpret operation associated with terminal symbols in the grammar.

an instance is required for every terminal symbol in a sentence.

NonterminalExpression (AlternationExpression,RepetitionExpression, SequenceExpressions)

one such class is required for every rule $R ::= R_1 R_2 \ldots R_n$ in the grammar.

maintains instance variables of type AbstractExpression for each of the symbols R_1 through R_n.

implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R_1 through R_n.

Context

contains information that's global to the interpreter.

Client

builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.

o invokes the Interpret operation.

Collaborations

The client builds (or is given) the sentence as an abstract syntaxtree of NonterminalExpression and TerminalExpression instances. Thenthe client initializes the context and invokes the Interpretoperation.

Each NonterminalExpression node defines Interpret in terms ofInterpret on each subexpression. The Interpret operation of eachTerminalExpression defines the base case in the recursion.

The Interpret operations at each node use the context tostore and access the state of the interpreter.

Consequences

The Interpreter pattern has the following benefits and liabilities:

- *It's easy to change and extend the grammar.*Because the pattern uses classesto represent grammar rules, you canuse inheritance to change or extend the grammar. Existing expressionscan be modified incrementally, and new expressions can be defined asvariations on old ones.
- *Implementing the grammar is easy, too.*Classes defining nodes in the abstractsyntax tree have similarimplementations. These classes are easy to write, and often theirgeneration can be automated with a compiler or parser generator.
- *Complex grammars are hard to maintain.*The Interpreter pattern defines atleast one class for every rulein the grammar (grammar rules defined using BNF may require multipleclasses). Hence grammars containing many rules can be hard tomanage and maintain. Other design patterns can be applied tomitigate the problem (see Implementation).But when the grammar is very complex, other techniques such asparser or compiler generators are more appropriate.
- *Adding new ways to interpret expressions.*The Interpreter pattern makes iteasier to evaluate an expression in anew way. For example, you can support pretty printing ortype-checking an expression by defining a new operation on theexpression classes. If you keep creating new ways of interpreting anexpression, then consider using the Visitor pattern to avoid changing the grammar classes.

Implementation

The Interpreter and Composite patterns share many implementation issues. The following issuesare specific to Interpreter:

- *Creating the abstract syntax tree.*The Interpreter pattern doesn't explainhow to *create* anabstract syntax tree. In other words, it doesn't address parsing.The abstract syntax tree can be created by a table-driven parser, by ahand-crafted (usually recursive descent) parser, or directly by theclient.
- *Defining the Interpret operation.*You don't have to define the Interpretoperation in the expressionclasses. If it's common to create a new interpreter, then it's betterto use the Visitor pattern to put Interpret in aseparate "visitor" object. For example, a grammar for a programminglanguage will have many operations on abstract syntax

trees, such asas type-checking, optimization, code generation, and so on. It will bemore likely to use a visitor to avoid defining these operations onevery grammar class.

*Sharing terminal symbols with the Flyweight pattern.*Grammars whosesentences contain many occurrences of a terminal symbolmight benefit from sharing a single copy of that symbol. Grammars forcomputer programs are good examples each program variable willappear in many places throughout the code. In the Motivation example,a sentence can have the terminal symbol dog (modeled by theLiteralExpression class) appearing many times.

Terminal nodes generally don't store information about their positionin the abstract syntax tree. Parent nodes pass them whatever contextthey need during interpretation. Hence there is a distinction betweenshared (intrinsic) state and passed-in (extrinsic) state, and the Flyweight pattern applies.

For example, each instance of LiteralExpression for dogreceives a context containing the substring matched so far. And everysuch LiteralExpression does the same thing in its Interpretoperation it checks whether the next part of the input contains adog no matter where the instance appears in the tree.

Sample Code

Here are two examples. The first is a complete example in Smalltalkfor checking whether a sequence matches a regular expression. Thesecond is a C++ program for evaluating Boolean expressions.

The regular expression matcher tests whether a string is in thelanguage defined by the regular expression. The regular expression isdefined by the following grammar:

expression ::= literal | alternation | sequence | repetition |
'(' expression ')'
alternation ::= expression '|' expression sequence
::= expression '&' expression repetition ::=
expression 'repeat'

literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

This grammar is a slight modification of the Motivation example. Wechanged the concrete syntax of regular expressions a little, becausesymbol "*" can't be a postfix operation in Smalltalk. Sowe use repeat instead. For example, the regular expression

(('dog ' | 'cat ') repeat & 'weather')

matches the input string "dog dog cat weather".

Known Uses

The Interpreter pattern is widely used in compilers implemented withobject-oriented languages, as the Smalltalk compilers are. SPECTalkuses the pattern to interpret descriptions of input fileformats [Sza92]. The QOCA constraint-solving toolkituses it to evaluate constraints [HHMV92].

Considered in its most general form (i.e., an operation distributedover a class hierarchy based on the Composite pattern), nearly everyuse of the Composite pattern will also contain the Interpreterpattern. But the Interpreter pattern should be reserved for thosecases in which you want to think of the class hierarchy as defining alanguage.

Related Patterns

Composite :The abstract syntax tree is an instance of the Composite pattern.

Flyweight shows how to share terminal symbols within the abstract syntaxtree.

Iterator :The interpreter can use an Iterator to traverse the structure.

Visitor canbe used to maintain the behavior in each node in the abstract syntaxtree in one class.

Iterator

Intent

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

Also Known As

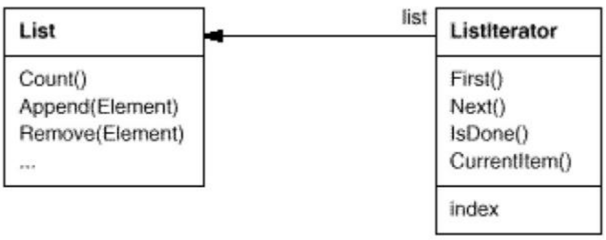
Cursor

Motivation

An aggregate object such as a list should give you a way to access itselements without exposing its internal structure. Moreover, you mightwant to traverse the list in different ways, depending on what youwant to accomplish. But you probably don't want to bloat the Listinterface with operations for different traversals, even if you couldanticipate the ones you will need. You might also need to have more thanone traversal pending on the same list.

The Iterator pattern lets you do all this. The key idea in thispattern is to take the responsibility for access and traversal out ofthe list object and put it into an iterator object. TheIterator class defines an interface for accessing the list's elements.An iterator object is responsible for keeping track of the currentelement; that is, it knows which elements have been traversed already.

For example, a List class would call for a ListIterator with thefollowing relationship between them:



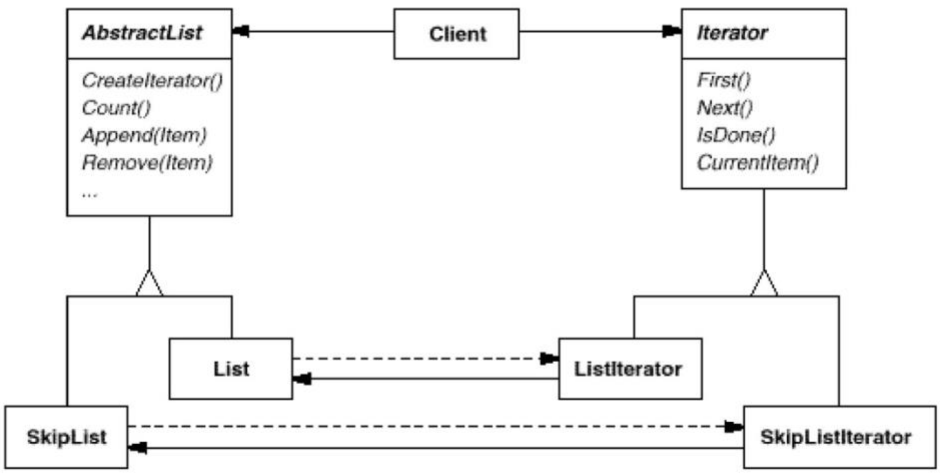
Before you can instantiate ListIterator, you must supply the List totraverse. Once you have the ListIterator instance, you can access thelist's elements sequentially. The CurrentItem operation returns thecurrent element in the list, First initializes the current element tothe first element, Next advances the current element to the nextelement, and IsDone tests whether we've advanced beyond the lastelement that is, we're finished with the traversal.

Separating the traversal mechanism from the List object lets us defineiterators for different traversal policies without enumerating them inthe List interface. For example, FilteringListIterator might provideaccess only to those elements that satisfy specific filteringconstraints.

Notice that the iterator and the list are coupled, and the client mustknow that it is a *list* that's traversed as opposed to some otheraggregate structure. Hence the client commits to a particularaggregate structure. It would be better if we could change the aggregateclass without changing client code. We can do this by generalizingthe iterator concept to support polymorphic iteration.

As an example, let's assume that we also have a SkipListimplementation of a list. A skiplist [Pug90] is aprobabilistic data structure with characteristics similar to balancedtrees. We want to be able to write code that works for both List andSkipList objects.

We define an AbstractList class that provides a common interfacefor manipulating lists. Similarly, we need an abstract Iteratorclass that defines a common iteration interface. Then we can defineconcrete Iterator subclasses for the different list implementations.As a result, the iteration mechanism becomes independent of concreteaggregate classes.



The remaining problem is how to create the iterator. Since we want towrite code that's independent of the concrete List subclasses, wecannot simply instantiate a specific class. Instead, we make the listobjects responsible for creating their
corresponding iterator. Thisrequires an operation like CreateIterator through which clientsrequest an iterator object.

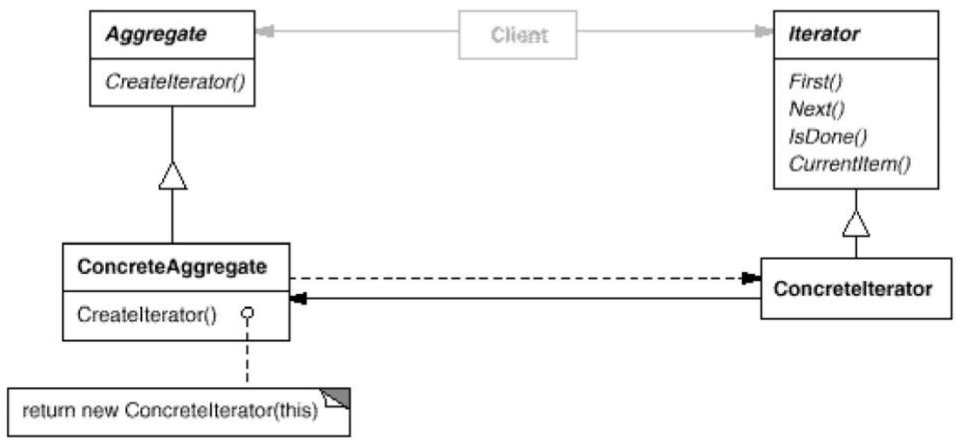
CreateIterator is an example of a factory method (see Factory Method (121)). We use it here to let a client aska list object for the appropriate iterator. The Factory Methodapproach give rise to two class hierarchies, one for lists and anotherfor iterators. The CreateIterator factory method "connects" the twohierarchies.

Applicability

Use the Iterator pattern

- to access an aggregate object's contents without exposing its internalrepresentation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregatestructures (that is, to support polymorphic iteration).

Structure



Participants

Iterator

defines an interface for accessing and traversing elements.

ConcreteIterator

implements the Iterator interface.
keeps track of the current position in the traversal of the aggregate.

Aggregate

defines an interface for creating an Iterator object.

ConcreteAggregate

implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Collaborations

A ConcreteIterator keeps track of the current object in theaggregate and can compute the succeeding object in thetraversal.

Consequences

The Iterator pattern has three important consequences:

*It supports variations in the traversal of an aggregate.*Complex aggregatesmay be traversed in many ways. For example, codegeneration and semantic checking involve traversing parse trees. Codegeneration may traverse the parse tree inorder or preorder.Iterators make it easy to change the traversal algorithm: Just replacethe iterator instance with a different one. You can also defineIterator subclasses to support new traversals.

*Iterators simplify the Aggregate interface.*Iterator's traversal interfaceobviates the need for a similarinterface in Aggregate, thereby simplifying the aggregate's interface.

*More than one traversal can be pending on an aggregate.*An iterator keepstrack of its own traversal state. Therefore you canhave more than one traversal in progress at once.

Implementation

Iterator has many implementation variants and alternatives. Someimportant ones follow. The trade-offs often depend on thecontrol structures your language provides. Some languages (CLU [LG86], for example) even support this pattern directly.

*Who controls the iteration?*A fundamental issue is deciding which partycontrols the iteration,the iterator or the client that uses the iterator. When the clientcontrols the iteration, the iterator is called an externaliterator, and when the iterator controls it, the iterator is aninternal iterator.[2]Clients that use anexternal iterator must advance the traversal and request the nextelement explicitly from the iterator. In contrast, the client handsan internal iterator an operation to perform, and the iterator appliesthat operation to every element in the aggregate.

External iterators are more flexible than internal iterators. It'seasy to compare two collections for equality with an externaliterator, for example, but it's practically impossible with internaliterators. Internal iterators are especially weak in a language likeC++ that does not provide anonymous functions, closures, orcontinuations like Smalltalk and CLOS. But on the other hand,internal iterators are easier to use, because they define the iterationlogic for you.

*Who defines the traversal algorithm?*The iterator is not the only place wherethe traversal algorithm canbe defined. The aggregate might define the traversal algorithm anduse the iterator to store just the state of the iteration. We callthis kind of iterator a cursor, since it merely points tothe current position in the aggregate. A client will invoke the Nextoperation on the aggregate with the cursor as an argument, and theNext operation will change the state of thecursor.[3]

If the iterator is responsible for the traversal algorithm, then it'seasy to use different iteration algorithms on the same aggregate, andit can also be easier to reuse the same algorithm on differentaggregates. On the other hand, the traversal algorithm might need toaccess the private variables of the aggregate. If so, putting thetraversal algorithm in the iterator violates the encapsulation of theaggregate.

▼ Sample Code

We'll look at the implementation of a simple List class, which is partof our foundation library (Appendix C) .We'll show two Iterator implementations, one for traversing the List infront-to-back order, and another for traversing back-to-front (thefoundation library supports only the first one). Then we show how touse these iterators and how to avoid committing to a particularimplementation. After that, we change the design to make sureiterators get deleted properly. The last example illustrates aninternal iterator and compares it to its external counterpart.

*List and Iterator interfaces.*First let's look at the part of the Listinterface that's relevant toimplementing iterators. Refer to (Appendix C). for the full interface.

```
template<class Item> class
List {
public:
List(long size =
DEFAULT_LIST_CAPACITY); long Count()
const;
Item&Get(long index) const;
// ...
};
```

The List class provides a reasonably efficient way tosupport iteration through its public interface. It's sufficient toimplement both traversals. So there's no need to give iteratorsprivileged access to the underlying data structure; that is, theiterator classes are not friends of List. To enabletransparent use of the different traversals we define an abstractIterator class, which defines the iterator interface.

```
template<class Item> class
Iterator { public:
virtual void First() = 0; virtual void
Next() = 0; virtual bool IsDone() const =
0;
```

```
virtual   Item   CurrentItem()   const   =   0;
protected:
Iterator();
};
```

2. *Iterator subclass implementations.*ListIterator is a subclass of Iterator.

```
template<class Item>
class ListIterator : public Iterator<Item> { public:
ListIterator(const  List<Item>*  aList);  virtual
void First();
virtual   void   Next();   virtual   bool
IsDone() const;
virtual Item CurrentItem() const; private:
const   List<Item>*   _list;   long
_current;
};
```

The implementation of ListIterator is straightforward. Itstores the List along with an index _current intothe list:

```
template<class Item>
ListIterator<Item>::ListIterator ( const List<Item>* aList )
        : _list(aList), _current(0) {     }
```

Known Uses

Iterators are common in object-oriented systems. Most collectionclass libraries offer iterators in one form or another.

Here's an example from the Booch components [Boo94], apopular collection class library. It provides both a fixed size(bounded) and dynamically growing (unbounded) implementation of aqueue. The queue interface is defined by an abstract Queue class. Tosupport polymorphic iteration over the different queueimplementations, the queue iterator is implemented in the terms of theabstract Queue class interface. This variation has the advantage thatyou don't need a factory method to ask the queue implementations fortheir appropriate iterator. However, it requires the interface of theabstract Queue class to be powerful enough to implement the iteratorefficiently.

Iterators don't have to be defined as explicitly in Smalltalk. Thestandard collection classes (Bag, Set, Dictionary, OrderedCollection,String, etc.) define an internal iterator method do:, whichtakes a block (i.e., closure) as an argument. Each element in thecollection is bound to the local variable in the block; then the blockis executed. Smalltalk also includes a set of Stream classes thatsupport an iterator-like interface. ReadStream is essentially anIterator, and it can act as an external iterator for all thesequential collections. There are no standard external iterators fornonsequential collections such as Set and Dictionary.

Polymorphic iterators and the cleanup Proxy described earlier areprovided by the ET++ container classes [WGM88]. The Unidrawgraphical editing framework classes use cursor-basediterators [VL90].

ObjectWindows 2.0 [Bor94] provides a class hierarchy ofiterators for containers. You can iterate over different containertypes in the same way. The ObjectWindow iteration syntax relies onoverloading the postincrement operator ++ to advance theiteration.

Related Patterns

Composite :Iterators are often applied to recursive structures such asComposites.

Factory Method :Polymorphic iterators rely on factory methods to instantiate theappropriate Iterator subclass.

Memento isoften used in conjunction with the Iterator pattern. An iteratorcan use a memento to capture the state of an iteration. The iteratorstores the memento internally.

Mediator

Intent

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

Motivation

Object-oriented design encourages the distribution of behavioramong objects. Such distribution can result in an object structurewith many connections between objects; in the worst case, every objectends up knowing about every other.

Though partitioning a system into many objects generally enhancesreusability, proliferating interconnections tend to reduce it again.Lots of interconnections make it less likely that an object can workwithout the support of others the system acts as though it weremonolithic. Moreover, it can be difficult to change the system'sbehavior in any significant way, since behavior is distributed amongmany objects. As a result, you may be forced to define many subclassesto customize the system's behavior.

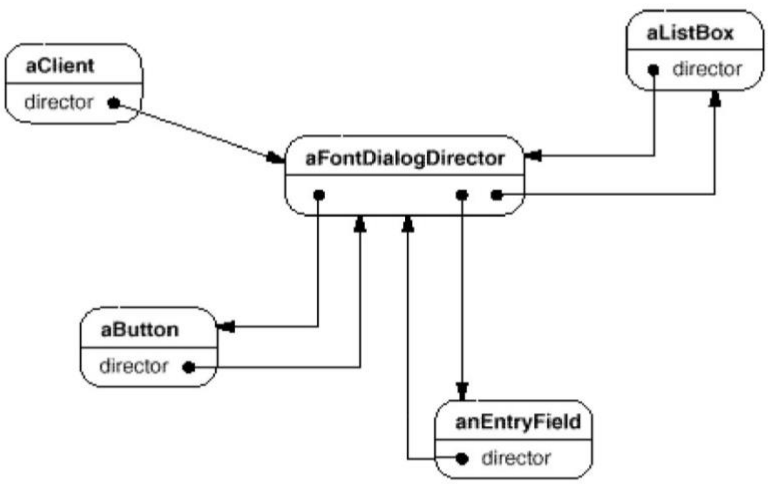
As an example, consider the implementation of dialog boxes in agraphical user interface. A dialog box uses a window to present acollection of widgets such as buttons, menus, and entry fields, asshown here:

Often there are dependencies between the widgets in the dialog. Forexample, a button gets disabled when a certain entry field is empty.Selecting an entry in a list of choices called a list boxmight change the contents of an entry field. Conversely, typing textinto the entry field might automatically select one or morecorresponding entries in the list box. Once text appears in the entryfield, other buttons may become enabled that let the user do somethingwith the text, such as changing or deleting the thing to which it refers.

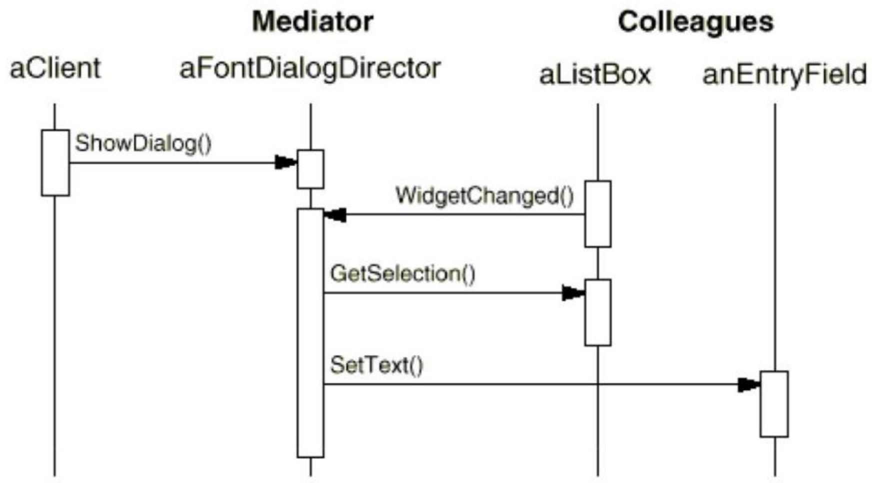
Different dialog boxes will have different dependencies betweenwidgets. So even though dialogs display the same kinds of widgets,they can't simply reuse stock widget classes; they have to becustomized to reflect dialog-specific dependencies. Customizing themindividually by subclassing will be tedious, since many classes areinvolved.

You can avoid these problems by encapsulating collective behavior in aseparate mediator object. A mediator is responsible forcontrolling and coordinating the interactions of a group of objects.The mediator serves as an intermediary that keeps objects in the groupfrom referring to each other explicitly. The objects only know themediator, thereby reducing the number of interconnections.

For example, FontDialogDirector can be the mediatorbetween the widgets in a dialog box. A FontDialogDirector object knowsthe widgets in a dialog and coordinates their interaction. It acts asa hub of communication for widgets:



The following interaction diagram illustrates how the objects cooperate tohandle a change in a list box's selection:

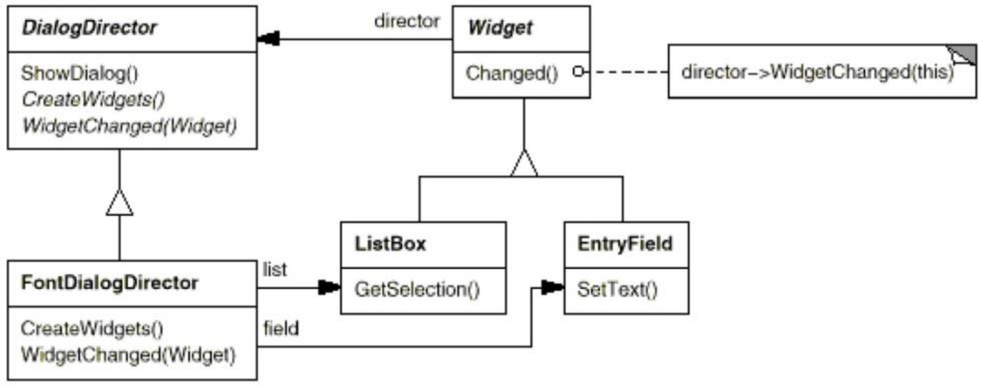


Here's the succession of events by which a list box's selection passesto an entry field:

The list box tells its director that it's changed.
The director gets the selection from the list box.
The director passes the selection to the entry field.
Now that the entry field contains some text, the directorenables button(s) for initiating an action (e.g., "demibold," "oblique").

Note how the director mediates between the list box and the entry field.Widgets communicate with each other only indirectly, through thedirector. They don't have to know about each other; all they know is thedirector. Furthermore, because the behavior is localized in one class,it can be changed or replaced by extending or replacing that class.

Here's how the FontDialogDirector abstraction can be integrated into aclass library:



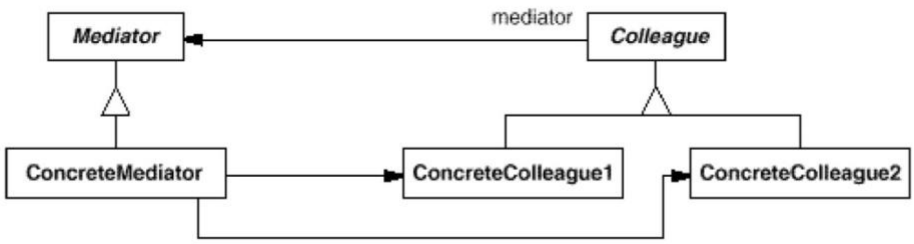
DialogDirector is an abstract class that defines the overall behavior ofa dialog. Clients call the ShowDialog operation to display the dialog onthe screen. CreateWidgets is an abstract operation for creating thewidgets of a dialog. WidgetChanged is another abstract operation;widgets call it to inform their director that they have changed.DialogDirector subclasses override CreateWidgets to create the properwidgets, and they override WidgetChanged to handle the changes.

Applicability

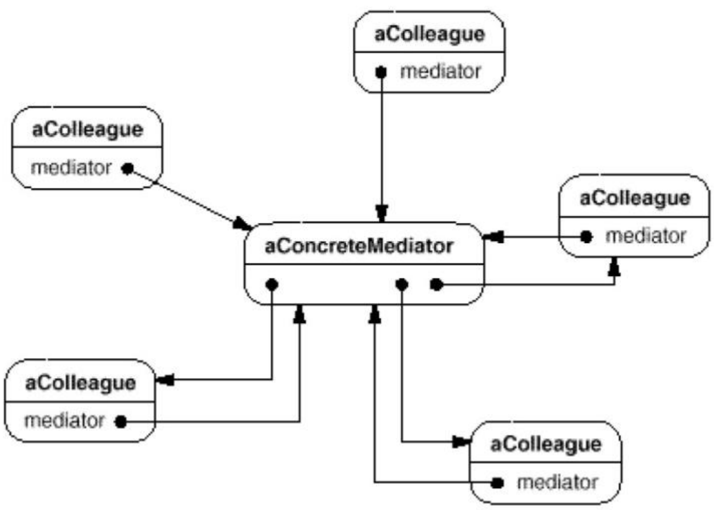
Use the Mediator pattern when

- a set of objects communicate in well-defined but complex ways. Theresulting interdependencies are unstructured and difficult tounderstand.
- reusing an object is difficult because it refers to and communicateswith many other objects.
- a behavior that's distributed between several classes should becustomizable without a lot of subclassing.

Structure



A typical object structure might look like this:



Participants

Mediator (DialogDirector)

defines an interface for communicating with Colleague objects.

ConcreteMediator (FontDialogDirector)

implements cooperative behavior by coordinating Colleague objects.

knows and maintains its colleagues.

Colleague classes (ListBox, EntryField)

each Colleague class knows its Mediator object.

each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

Collaborations

Colleagues send and receive requests from a Mediator object. Themediator implements the cooperative behavior by routing requestsbetween the appropriate colleague(s).

Consequences

The Mediator pattern has the following benefits and drawbacks:

*It limits subclassing.*A mediator localizes behavior that otherwise wouldbe distributed amongseveral objects. Changing this behavior requires subclassing Mediatoronly; Colleague classes can be reused as is.

*It decouples colleagues.*A mediator promotes loose coupling betweencolleagues. You can varyand reuse Colleague and Mediator classes independently.

*It simplifies object protocols.*A mediator replaces many-to-manyinteractions with one-to-manyinteractions between the mediator and its colleagues. One-to-manyrelationships are easier to understand, maintain, and extend.

*It abstracts how objects cooperate.*Making mediation an independent conceptand encapsulating it in anobject lets you focus on how objects interact apart from theirindividual behavior. That can help clarify how objects interact in asystem.

*It centralizes control.*The Mediator pattern trades complexity ofinteraction for complexity inthe mediator. Because a mediator encapsulates protocols, it can becomemore complex than any individual colleague. This can make the mediatoritself a monolith that's hard to maintain.

Implementation

The following implementation issues are relevant to the Mediatorpattern:

Omitting the abstract Mediator class. There's no need to define an abstractMediator class when colleagueswork with only one mediator. The abstract coupling that theMediator class provides lets colleagues work with different Mediatorsubclasses, and vice versa.

Colleague-Mediator communication. Colleagues have to communicate withtheir mediator when an event ofinterest occurs. One approach is to implement the Mediator as anObserver using the Observer pattern. Colleagueclasses act as Subjects, sending notifications to the mediatorwhenever they change state. The mediator responds by propagating theeffects of the change to other colleagues.

Another approach defines a specialized notification interface inMediator that lets colleagues be more direct in their communication.Smalltalk/V for Windows uses a form of delegation: When communicatingwith the mediator, a colleague passes itself as an argument, allowingthe mediator to identify the sender. The Sample Code uses thisapproach, and the Smalltalk/V implementation is discussed further inthe Known Uses.

Sample Code

We'll use a DialogDirector to implement the font dialog box shown inthe Motivation. The abstract class DialogDirector definesthe interface for directors.

```
class DialogDirector { public:
virtual  ~DialogDirector();  virtual
void ShowDialog();
virtual  void  WidgetChanged(Widget*)  =  0;
protected:
DialogDirector();
virtual void CreateWidgets() = 0; };
```

Widget is the abstract base class for widgets. Awidget knows its director.

```
class Widget {
public:
Widget(DialogDirector*);
virtual void Changed();
virtual void HandleMouse(MouseEvent& event);
// ...
private:
DialogDirector* _director;
};
```

Changed calls the director's WidgetChangedoperation. Widgets call WidgetChanged on their director toinform it of a significant event.

```
void Widget::Changed ()
{        _director->WidgetChanged(this);       }
```

Subclasses of DialogDirector overrideWidgetChanged to affect the appropriate widgets. The widgetpasses a reference to itself as an argument to WidgetChangedto let the director identify the widget that changed.DialogDirector subclasses redefine theCreateWidgets pure virtual to construct the widgets in thedialog.

The ListBox, EntryField, and Button aresubclasses of Widget for specialized user interfaceelements. ListBox provides a GetSelectionoperation to get the current selection, and EntryField'sSetText operation puts new text into the field.

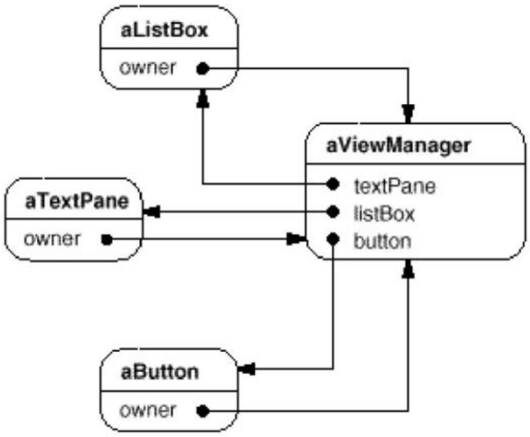
```
class ListBox : public Widget { public:
ListBox(DialogDirector*);
virtual const char* GetSelection();
virtual void SetList(List<char*>* listItems); virtual
void HandleMouse(MouseEvent& event); // ...
};
```

Known Uses

Both ET++ [WGM88] and the THINK C class library [Sym93b] usedirector-like objects in dialogs as mediators between widgets.

The application architecture of Smalltalk/V for Windows is based on amediator structure [LaL94]. In that environment, anapplication consists of a Window containing a set of panes. Thelibrary contains several predefined Pane objects; examples includeTextPane, ListBox, Button, and so on.These panes can be used without subclassing. An application developeronly subclasses from ViewManager, a class that's responsible for doinginter-pane coordination. ViewManager is the Mediator, and each paneonly knows its view manager, which is considered the "owner" of thepane. Panes don't refer to each other directly.

The following object diagram shows a snapshot of an application atrun-time:



Smalltalk/V uses an event mechanism for Pane-ViewManagercommunication. A pane generates an event when it wants to getinformation from the mediator or when it wants to inform the mediatorthat something significant happened. An event defines a symbol (e.g.,#select) that identifies the event. To handle the event, theview manager registers a method selector with the pane. This selectoris the event's handler; it will be invoked whenever the event occurs.

The following code excerpt shows how a ListPane object gets created insidea ViewManager subclass and how ViewManager registers an event handlerfor the #select event:

Related Patterns

Facade differsfrom Mediator in that it abstracts a subsystem of objects to providea more convenient interface. Its protocol is unidirectional; thatis, Facade objects make requests of the subsystem classes but notvice versa. In contrast, Mediator enables cooperative behaviorthat colleague objects don't or can't provide, and the protocol ismultidirectional.

Colleagues can communicate with the mediator using the Observer pattern.

Memento

Intent

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

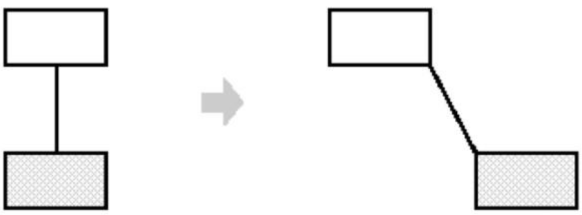
Also Known As

Token

Motivation

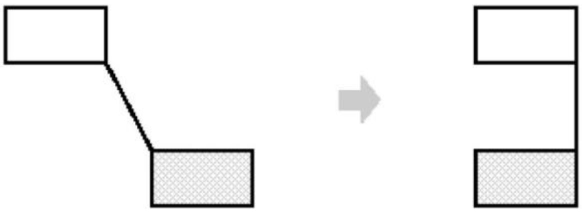
Sometimes it's necessary to record the internal state of an object.This is required when implementing checkpoints and undo mechanismsthat let users back out of tentative operations or recover fromerrors. You must save state information somewhere so that you canrestore objects to their previous states. But objects normallyencapsulate some or all of their state, making it inaccessible toother objects and impossible to save externally. Exposing this statewould violate encapsulation, which can compromise the application'sreliability and extensibility.

Consider for example a graphical editor that supports connectivitybetween objects. A user can connect two rectangles with a line, andthe rectangles stay connected when the user moves either of them. Theeditor ensures that the line stretches to maintain the connection.



A well-known way to maintain connectivity relationships betweenobjects is with a constraint-solving system. We can encapsulate thisfunctionality in a ConstraintSolver object.ConstraintSolver records connections as they are made and generatesmathematical equations that describe them. It solves these equationswhenever the user makes a connection or otherwise modifies thediagram. ConstraintSolver uses the results of its calculations torearrange the graphics so that they maintain the proper connections.

Supporting undo in this application isn't as easy as it may seem. Anobvious way to undo a move operation is to store the original distancemoved and move the object back an equivalent distance. However, thisdoes not guarantee all objects will appear where they did before.Suppose there is some slack in the connection. In that case, simplymoving the rectangle back to its original location won't necessarilyachieve the desired effect.



In general, the ConstraintSolver's public interface might beinsufficient to allow precise reversal of its effects on otherobjects. The undo mechanism must work more closely withConstraintSolver to reestablish previous state, but we should alsoavoid exposing the ConstraintSolver's internals to the undo mechanism.

We can solve this problem with the Memento pattern. A memento is an object that stores a snapshot of theinternal state of another object the memento's originator. The undo mechanism will request a mementofrom the originator when it needs to checkpoint the originator'sstate. The originator initializes the memento with information thatcharacterizes its current state. Only the originator can store andretrieve information from the memento the memento is "opaque" toother objects.

In the graphical editor example just discussed, the ConstraintSolver can actas an originator. The following sequence of events characterizes theundo process:

The editor requests a memento from the ConstraintSolver as aside-effect of the move operation.

The ConstraintSolver creates and returns a memento, an instance of aclass SolverState in this case. A SolverState memento contains datastructures that describe the current state of the ConstraintSolver'sinternal equations and variables.

Later when the user undoes the move operation, the editor gives theSolverState back to the ConstraintSolver.

Based on the information in the SolverState, the ConstraintSolverchanges its internal structures to return its equations and variablesto their exact previous state.

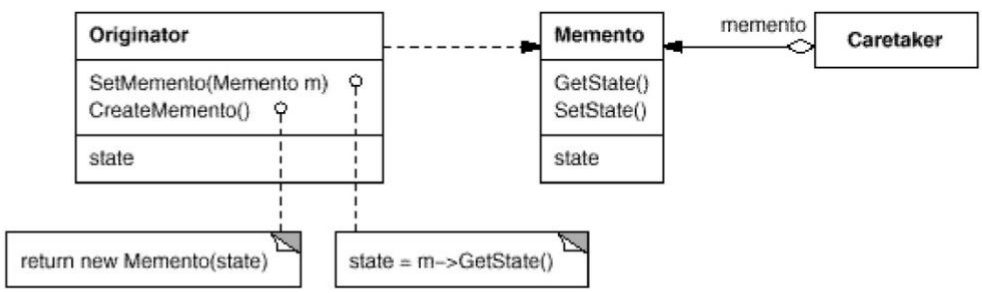
This arrangement lets the ConstraintSolver entrust other objects withthe information it needs to revert to a previous state withoutexposing its internal structure and representations.

Applicability

Use the Memento pattern when

- a snapshot of (some portion of) an object's state must be saved sothat it can be restored to that state later, *and*
- a direct interface to obtaining the state would exposeimplementation details and break the object's encapsulation.

Structure



Participants

Memento (SolverState)

stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.

protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a *narrow* interface to the Memento it can only pass the memento to other objects. Originator, in contrast, sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

Originator (ConstraintSolver)

creates a memento containing a snapshot of its current internal state.

uses the memento to restore its internal state.

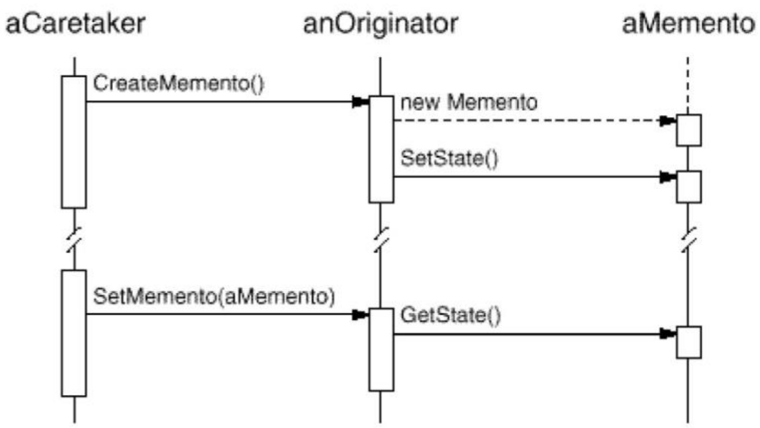
Caretaker (undo mechanism)

is responsible for the memento's safekeeping.

- never operates on or examines the contents of a memento.

Collaborations

A caretaker requests a memento from an originator, holds it for atime, and passes it back to the originator, as the followinginteraction diagram illustrates:



Sometimes the caretaker won't pass the memento back to the originator,because the originator might never need to revert to an earlier state.

Mementos are passive. Only the originator that created a memento willassign or retrieve its state.

Consequences

The Memento pattern has several consequences:

*Preserving encapsulation boundaries.*Memento avoids exposing informationthat only an originator shouldmanage but that must be stored nevertheless outside the originator.The pattern shields other objects from potentially complex Originatorinternals, thereby preserving encapsulation boundaries.

*It simplifies Originator.*In other encapsulation-preserving designs,Originator keeps theversions of internal state that clients have requested. That puts allthe storage management burden on Originator. Having clientsmanage the state they ask for simplifies Originator and keepsclients from having to notify originators when they're done.

*Using mementos might be expensive.*Mementos might incur considerableoverhead if Originator must copylarge amounts of information to store in the memento or if clientscreate and return mementos to the originator often enough. Unlessencapsulating and restoring Originator state is cheap, the patternmight not be appropriate. See the discussion of incrementality in theImplementation section.

*Defining narrow and wide interfaces.*It may be difficult in some languagesto ensure that only theoriginator can access the memento's state. *Hidden costs in caring for mementos.*A caretaker is responsible for deletingthe mementos it cares for.However, the caretaker has no idea how much state is in the memento.Hence an otherwise lightweight caretaker might incur large storagecosts when it stores mementos.

Implementation

Here are two issues to consider when implementing the Memento pattern:

*Language support.*Mementos have two interfaces: a wide one for originatorsand a narrowone for other objects. Ideally the implementation language willsupport two levels of static protection. C++ lets you do this bymaking the Originator a friend of Memento and making Memento's wideinterface private. Only the narrow interface should be declaredpublic. For example:

```
class State;
```

```
class Originator { public:

Memento* CreateMemento();
void SetMemento(const Memento*);
    ...
private: State* _state;

    internal data structures
    ...
};

class Memento { public:
    narrow  public  interface  virtual
~Memento(); private:
    private members accessible only to Originator friend
class Originator;
Memento();
void   SetState(State*);   State*
GetState();
private: State* _state;
    ...
};
```

Sample Code

The C++ code given here illustrates the ConstraintSolver example discussed earlier. Weuse MoveCommand objects dothe translation of a graphical object from one position to another.The graphical editor calls the command's Execute operationto move a graphical object and Unexecute to undo the move.The command stores its target, the distance moved, and an instance ofConstraintSolverMemento, a memento containing state from theconstraint solver.

```
class Graphic;
// base class for graphical objects in the graphical editor
```

```
class MoveCommand {
public:
MoveCommand(Graphic* target, const Point& delta);
void Execute();
void Unexecute();
private:
ConstraintSolverMemento* _state;
Point _delta;
Graphic* _target; };
```

The connection constraints are established by the classConstraintSolver. Its key member function isSolve, which solves the constraints registered withthe AddConstraint operation. To support undo,ConstraintSolver's state can be externalized withCreateMemento into a ConstraintSolverMementoinstance. The constraint solver can be returned to a previousstate by calling SetMemento. ConstraintSolveris a Singleton (144).

```
class ConstraintSolver { public:
static ConstraintSolver* Instance(); void
Solve();
void AddConstraint(
Graphic* startConnection, Graphic* endConnection );
void RemoveConstraint(
Graphic* startConnection, Graphic* endConnection );
ConstraintSolverMemento* CreateMemento();
void SetMemento(ConstraintSolverMemento*);
private:
    nontrivial state and operations for enforcing
    connectivity semantics };

class ConstraintSolverMemento {
public:
```

```
virtual ~ConstraintSolverMemento();
private:
friend class ConstraintSolver;
ConstraintSolverMemento();
// private constraint solver state };
```

Given these interfaces, we can implement MoveCommand membersExecute and Unexecute as follows:

```
void MoveCommand::Execute () {
ConstraintSolver* solver = ConstraintSolver::Instance(); _state
= solver->CreateMemento();
// create a memento _target-
>Move(_delta); solver-
>Solve();
}

void MoveCommand::Unexecute () {
ConstraintSolver* solver = ConstraintSolver::Instance();
_target->Move(-_delta);
solver->SetMemento(_state); //
restore solver state solver-
>Solve();
}
```

Execute acquires a ConstraintSolverMemento mementobefore it moves the graphic. Unexecute moves the graphicback, sets the constraint solver's state to the previous state, andfinally tells the constraint solver to solve the constraints.

Known Uses

The preceding sample code is based on Unidraw's support for connectivitythrough its CSolver class [VL90].

Collections in Dylan [App92] provide an iteration interface thatreflects the Memento pattern. Dylan's collections have the notion of a"state" object, which is a memento that represents the state of theiteration. Each collection can represent the current state of theiteration in any way it chooses; the representation is completelyhidden from clients. The Dylan iteration approach might be translatedto C++ as follows:

```
template<class Item> class
Collection { public:
Collection();
IterationState* CreateInitialState(); void
Next(IterationState*);
bool IsDone(const IterationState*) const; Item
CurrentItem(const IterationState*) const;
IterationState* Copy(const IterationState*) const; void
Append(const Item&);
void Remove(const Item&);
// ...
};
```

CreateInitialState returns an initializedIterationState object for the collection. Next advancesthe state object to the next position in the iteration; it effectivelyincrements the iteration index. IsDone returnstrue if Next has advanced beyond the last elementin the collection. CurrentItem dereferences the stateobject and returns the element in the collection to which it refers.Copy returns a copy of the given state object. This isuseful for marking a point in an iteration.

Given a class ItemType, we can iterate over a collection ofits instances as follows:

```
class ItemType { public:
```

```
void Process();
// ...
};

Collection<ItemType*> aCollection;
IterationState* state;
state = aCollection.CreateInitialState(); while
(!aCollection.IsDone(state)) {
aCollection.CurrentItem(state)->Process();
aCollection.Next(state);
}
delete state;
```

The memento-based iteration interface has two interesting benefits:

- More than one state can work on the same collection. (The sameis true of the Iterator (289) pattern.)
- It doesn't require breaking a collection's encapsulationto support iteration. The memento is only interpreted by thecollection itself; no one else has access to it. Other approaches toiteration require breaking encapsulation by making iterator classesfriends of their collection classes. The situation is reversed in thememento-based implementation: Collection is a friend of theIteratorState.

The QOCA constraint-solving toolkit stores incremental information inmementos [HHMV92]. Clients can obtain a memento that characterizesthe current solution to a system of constraints. The memento containsonly those constraint variables that have changed since the lastsolution.Usually only a small subset of the solver's variableschanges for each new solution.

Related Patterns

Command : Commands can use mementos to maintainstate for undoable operations.

Iterator : Mementoscan be used for iteration as described earlier.

Note that our example deletes the state object at the end of the iteration. But delete won't get called if ProcessItem throws an exception, thus creating garbage. This is a problem in C++ but not in Dylan, which has garbage collection.

Observer

Intent

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

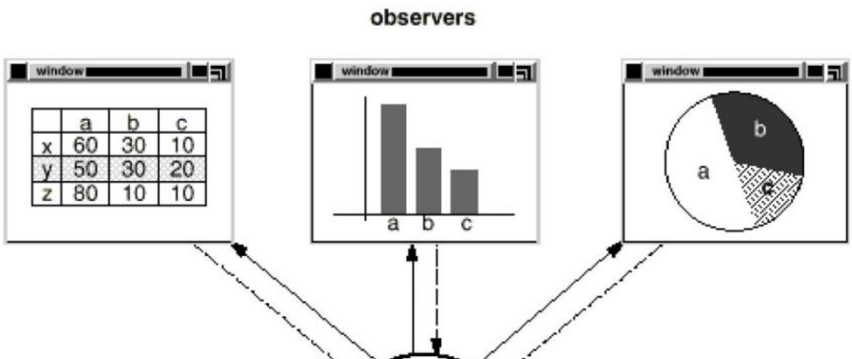
Also Known As

Dependents, Publish-Subscribe

Motivation

A common side-effect of partitioning a system into a collection ofcooperating classes is the need to maintain consistency betweenrelated objects. You don't want to achieve consistency by making theclasses tightly coupled, because that reduces their reusability.

For example, many graphical user interface toolkits separate thepresentational aspects of the user interface from the underlyingapplication data [KP88, LVC89, P+88, WGM88].Classes defining application data and presentations can be reusedindependently. They can work together, too. Both a spreadsheet objectand bar chart object can depict information in the same application dataobject using different presentations. The spreadsheet and the bar chartdon't know about each other, thereby letting you reuse only the one youneed. But they *behave* as though they do. When the user changes theinformation in the spreadsheet, the bar chart reflects the changesimmediately, and vice versa.



This behavior implies that the spreadsheet and bar chart are dependenton the data object and therefore should be notified of any change inits state. And there's no reason to limit the number of dependentobjects to two; there may be any number of different user interfacesto the same data.

The Observer pattern describes how to establish these relationships.The key objects in this pattern are subject andobserver. A subject may have any number of dependentobservers. All observers are notified whenever the subject undergoesa change in state. In response, each observer will query the subjectto synchronize its state with the subject's state.

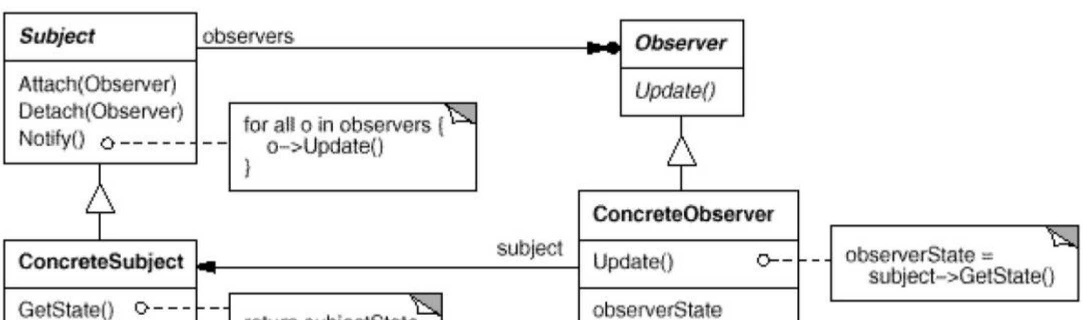
This kind of interaction is also known aspublish-subscribe. The subject is the publisher ofnotifications. It sends out these notifications without having to knowwho its observers are. Any number of observers can subscribe toreceive notifications.

Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other.Encapsulating these aspects in separate objects lets you vary andreuse them independently.
- When a change to one object requires changing others, and youdon't know how many objects need to be changed.
- When an object should be able to notify other objects without makingassumptions about who these objects are. In other words, you don'twant these objects tightly coupled.

Structure



Participants

Subject

knows its observers. Any number of Observer objects may observe a subject.

provides an interface for attaching and detaching Observer objects.

Observer

defines an updating interface for objects that should be notified of changes in a subject.

ConcreteSubject

stores state of interest to ConcreteObserver objects.

sends a notification to its observers when its state changes.

ConcreteObserver

maintains a reference to a ConcreteSubject object.

- stores state that should stay consistent with the subject's.

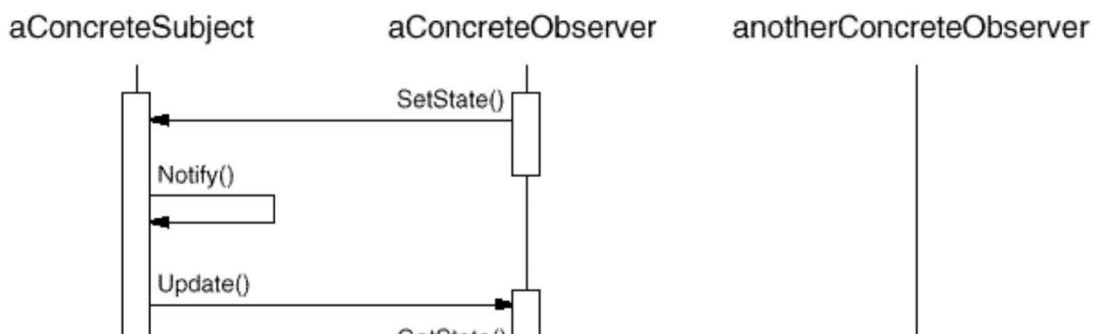
implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

ConcreteSubject notifies its observers whenever a changeoccurs that could make its observers' state inconsistent with its own.

After being informed of a change in the concrete subject, aConcreteObserver object may query the subject for information.ConcreteObserver uses this information to reconcile its state with thatof the subject.

The following interaction diagram illustrates the collaborationsbetween a subject and two observers:



Note how the Observer object that initiates the change requestpostpones its update until it gets a notification from the subject.Notify is not always called by the subject. It can be called by anobserver or by another kind of object entirely. The Implementationsection discusses some common variations.

Consequences

The Observer pattern lets you vary subjects and observersindependently. You can reuse subjects without reusing theirobservers, and vice versa. It lets you add observers withoutmodifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include thefollowing:

*Abstract coupling between Subject and Observer.*All a subject knows is thatit has a list of observers, eachconforming to the simple interface of the abstract Observer class.The subject doesn't know the concrete class of any observer. Thus thecoupling between subjects and observers is abstract and minimal.

Because Subject and Observer aren't tightly coupled, they can belong todifferent layers of abstraction in a system. A lower-level subjectcan communicate and inform a higher-level observer, thereby keeping thesystem's layering intact. If Subject and Observer are lumpedtogether, then the resulting object must either span two layers (andviolate the
layering), or it must be forced to live in one layer orthe other (which might compromise the layering abstraction).

*Support for broadcast communication.*Unlike an ordinary request, thenotification that a subject sendsneedn't specify its receiver. The notification is broadcastautomatically to all interested objects that subscribed to it. Thesubject doesn't care how many interested objects exist; its onlyresponsibility is to notify its observers. This gives you the freedomto add and remove observers at any time. It's up to the observer tohandle or ignore a notification.

*Unexpected updates.*Because observers have no knowledge of each other'spresence, they canbe blind to the ultimate cost of changing the subject. A seeminglyinnocuous operation on the subject may cause a cascade of updates toobservers and their dependent objects. Moreover, dependency criteriathat aren't well-defined or maintained usually lead to spuriousupdates, which can be hard to track down.

This problem is aggravated by the fact that the simple update protocolprovides no details on *what* changed in the subject. Withoutadditional protocol to help observers discover what changed, they maybe forced to work hard to deduce the changes.

Implementation

Several issues related to the implementation of the dependencymechanism are discussed in this section.

*Mapping subjects to their observers.*The simplest way for a subject to keeptrack of the observers itshould notify is to store references to them explicitly in thesubject. However, such storage may be too expensive when there aremany subjects and few observers. One solution is to trade space fortime by using an associative look-up (e.g., a hash table) to maintainthe subject-to-observer mapping. Thus a subject with no observersdoes not incur storage overhead. On the other hand, this approachincreases the cost of accessing the observers.

*Observing more than one subject.*It might make sense in some situations foran observer to depend onmore than one subject. For example, a spreadsheet may depend on morethan one data source. It's necessary to extend the Update interfacein such cases to let the observer know *which* subject is sendingthe notification. The subject can simply pass itself as a parameterin the Update operation, thereby letting the observer know whichsubject to examine.

*Who triggers the update?*The subject and its observers rely on thenotification mechanism tostay consistent. But what object actually calls Notify to trigger theupdate? Here are two options:

Have state-setting operations on Subject call Notify after theychange the subject's state. The advantage of this approach is thatclients don't have to remember to call Notify on the subject. Thedisadvantage is that several consecutive operations will causeseveral consecutive updates, which may be inefficient.

Make clients responsible for calling Notify at the right time.The advantage here is that the client can wait to trigger the updateuntil after a series of state changes has been made, therebyavoiding needless intermediate updates. The disadvantage is thatclients have an added responsibility to trigger the update. Thatmakes errors more likely, since clients might forget to call Notify.

};

The following code creates an AnalogClock and aDigitalClock that always show the same time:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Whenever the timer ticks, the two clocks will be updatedand will redisplay themselves appropriately.

Known Uses

The first and perhaps best-known example of the Observer pattern appearsin Smalltalk Model/View/Controller (MVC), the user interface framework in the Smalltalkenvironment [KP88]. MVC's Model class plays the role ofSubject, while View is the base class for observers. Smalltalk,ET++ [WGM88], and the THINK class library [Sym93b] provide ageneral dependency mechanism by putting Subject and Observer interfacesin the parent class for all other classes in the system.

Other user interface toolkits that employ this pattern areInterViews [LVC89], the AndrewToolkit [P+88], and Unidraw [VL90]. InterViewsdefines Observer and Observable (for subjects) classes explicitly.Andrew calls them "view" and "data object," respectively. Unidrawsplits graphical editor objects into View (for observers) and Subjectparts.

Related Patterns

Mediator : Byencapsulating complex update semantics, the ChangeManager acts asmediator between subjects and observers.

Singleton :The ChangeManager may use the Singleton pattern to make it uniqueand globally accessible.

State

Intent

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

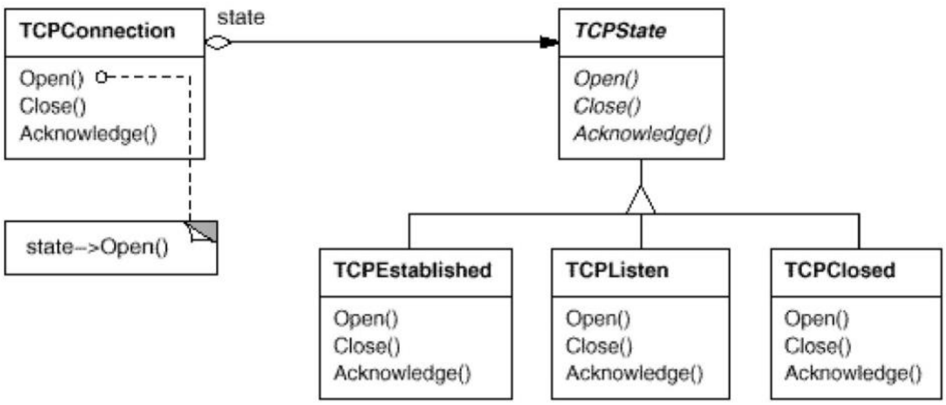
Also Known As

Objects for States

Motivation

Consider a class TCPConnection that represents a network connection.A TCPConnection object can be in one of several different states:Established, Listening, Closed. When a TCPConnection object receivesrequests from other objects, it responds differently depending on itscurrent state. For example, the effect of an Open request depends onwhether the connection is in its Closed state or its Establishedstate. The State pattern describes how TCPConnection can exhibitdifferent behavior in each state.

The key idea in this pattern is to introduce an abstract class calledTCPState to represent the states of the network connection. TheTCPState class declares an interface common to all classes thatrepresent different operational states. Subclasses of TCPStateimplement state-specific behavior. For example, the classesTCPEstablished and TCPClosed implement behavior particular to theEstablished and Closed states of TCPConnection.



The class TCPConnection maintains a state object (an instance of asubclass of TCPState) that represents the current state of the TCPconnection. The class TCPConnection delegates all state-specificrequests to this state object. TCPConnection uses its TCPStatesubclass instance to perform operations particular to the state of theconnection.

Whenever the connection changes state, the TCPConnection objectchanges the state object it uses. When the connection goes fromestablished to closed, for example, TCPConnection will replace itsTCPEstablished instance with a TCPClosed instance.

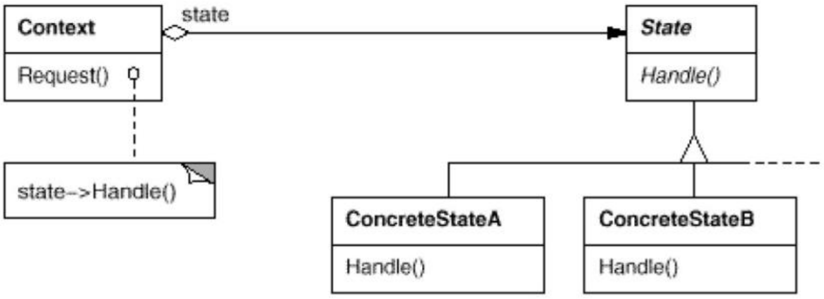
Applicability

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change itsbehavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend onthe object's state. This state is usually represented by one or moreenumerated constants. Often, several

operations will contain thissame conditional structure. The State pattern puts each branch of theconditional in a separate class. This lets you treat the object'sstate as an object in its own right that can vary independently fromother objects.

Structure



Participants

Context (TCPConnection)

- o defines the interface of interest to clients.

  maintains an instance of a ConcreteState subclass that defines thecurrent state.

State (TCPState)

  defines an interface for encapsulating the behavior associated with aparticular state of the Context.

ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed)

  each subclass implements a behavior associated with a state ofthe Context.

Collaborations

Context delegates state-specific requests to the current ConcreteStateobject.

A context may pass itself as an argument to the State objecthandling the request. This lets the State object accessthe context if necessary.

Context is the primary interface for clients. Clients can configure acontext with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.

Either Context or the ConcreteState subclasses can decide which statesucceeds another and under what circumstances.

Consequences

The State pattern has the following consequences:

*It localizes state-specific behavior and partitionsbehavior for different states.*The State pattern puts all behavior associated with a particularstateinto one object. Because all state-specific code lives in a Statesubclass, new states and transitions can be added easily by definingnew subclasses.

An alternative is to use data values to define internal states andhave Context operations check the data explicitly. But then we'dhave look-alike conditional or case statements scattered throughoutContext's implementation. Adding a new state could requirechanging several operations, which complicates maintenance.

The State pattern avoids this problem but might introduce another,because the pattern distributes behavior for different states acrossseveral State subclasses. This increases the number of classes and isless compact than a single class. But such distribution is actuallygood if there are many states, which would otherwise necessitate largeconditional statements.

Like long procedures, large conditional statements are undesirable.They're monolithic and tend to make the code less explicit, whichin turn makes them difficult to modify and extend. The State patternoffers a better way to structure state-specific code. The logic thatdetermines the state transitions doesn't reside in monolithicif or switch statements but instead is partitionedbetween the State subclasses. Encapsulating each state transition andaction in a class elevates the idea of an execution state to fullobject status. That imposes structure on the code and makes itsintent clearer.

*It makes state transitions explicit.*When an object defines its current statesolely in terms of internaldata values, its state transitions have no explicit representation;they only show up as assignments to some variables. Introducingseparate objects for different states makes the transitions moreexplicit. Also, State objects can protect the Context frominconsistent internal states, because state transitions are atomicfrom the

Context's perspective they happen by rebinding *one*variable (the Context's State object variable), notseveral [dCLF93].

*State objects can be shared.*If State objects have no instance variables thatis, the state theyrepresent is encoded entirely in their type then contexts can sharea State object. When states are shared in this way, they areessentially flyweights with nointrinsic state, only behavior.

Implementation

The State pattern raises a variety of implementation issues:

*Who defines the state transitions?*The State pattern does not specify whichparticipant defines thecriteria for state transitions. If the criteria are fixed, then theycan be implemented entirely in the Context. It is generally moreflexible and appropriate, however, to let the State subclassesthemselves specify their successor state and when to make thetransition. This requires adding an interface to the Context thatlets State objects set the Context's current state explicitly.

Decentralizing the transition logic in this way makes it easy tomodify or extend the logic by defining new State subclasses. Adisadvantage of decentralization is that one State subclass will haveknowledge of at least one other, which introduces implementationdependencies between subclasses.

*A table-based alternative.*In *C++ Programming Style* [Car92],Cargilldescribes another way to impose structure on state-driven code: Heuses tables to map inputs to state transitions. For each state, atable maps every possible input to a succeeding state. In effect,this approach converts conditional code (and virtual functions, in thecase of the State pattern) into a table look-up.

The main advantage of tables is their regularity: You can change thetransition criteria by modifying data instead of changing programcode. There are some disadvantages, however:

A table look-up is often less efficient than a (virtual)function call.

Putting transition logic into a uniform, tabular format makes thetransition criteria less explicit and therefore harder to understand.
It's usually difficult to add actions to accompany the statetransitions. The table-driven approach captures the states and theirtransitions, but it must be augmented to perform arbitrary computationon each transition.

The key difference between table-driven state machines and the Statepattern can be summed up like this: The State pattern modelsstate-specific behavior, whereas the table-driven approach focuses ondefining state transitions.

*Creating and destroying State objects.*A common implementation trade-offworth considering is whether(1) to create State objects only when they are needed and destroy themthereafter versus (2) creating them ahead of time and neverdestroying them.

The first choice is preferable when the states that will be enteredaren't known at run-time, *and* contexts change stateinfrequently. This approach avoids creating objects that won't beused, which is important if the State objects store a lot ofinformation. The second approach is better when state changes occurrapidly, in which case you want to avoid destroying states, becausethey may be needed again shortly. Instantiation costs are paid onceup-front, and there are no destruction costs at all.
This approachmight be inconvenient, though, because the Context must keepreferences to all states that might be entered.

*Using dynamic inheritance.*Changing the behavior for a particular requestcould be accomplishedby changing the object's class at run-time, but this is not possiblein most object-oriented programming languages. Exceptions includeSelf [US87] and other delegation-based languages thatprovide such a mechanism and hence support the State pattern directly.Objects in Self can delegate operations to other objects to achieve aform of dynamic inheritance. Changing the delegation target atrun-time effectively changes the inheritance structure. Thismechanism lets objects change their behavior and amounts to changingtheir class.

Sample Code

The following example gives the C++ code for the TCP connectionexample described in the Motivation section. This example is asimplified version of the TCP protocol; it doesn't describe thecomplete protocol or all the states of TCPconnections.[8]

First, we define the class TCPConnection, which provides aninterface for transmitting data and handles requests to change state.

```
class TCPOctetStream; class

TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen(); void

    PassiveOpen();     void

    Close();

    void Send();
    void Acknowledge(); void

    Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

TCPConnection keeps an instance of the TCPStateclass in the _state member variable. The classTCPState duplicates the state-changing interface ofTCPConnection. Each TCPState operation takes aTCPConnection instance as a parameter, lettingTCPState access data from TCPConnection andchange the connection's state.

```
class TCPState { public:
        virtual void Transmit(TCPConnection*, TCPOctetStream*);
        virtual void ActiveOpen(TCPConnection*);
        virtual   void   PassiveOpen(TCPConnection*);
        virtual   void   Close(TCPConnection*);   virtual
        void Synchronize(TCPConnection*);
        virtual void Acknowledge(TCPConnection*);
        virtual void Send(TCPConnection*);
protected:
        void ChangeState(TCPConnection*, TCPState*);
};
```

TCPConnection delegates all state-specific requests to itsTCPState instance _state.TCPConnection also provides an operation for changing thisvariable to a new TCPState. The constructor forTCPConnection initializes the object to theTCPClosed state (defined later).

```
TCPConnection::TCPConnection () {
        _state = TCPClosed::Instance();
}

void  TCPConnection::ChangeState  (TCPState*  s)  {
        _state = s;
}

void   TCPConnection::ActiveOpen   ()   {
        _state->ActiveOpen(this);
}

void   TCPConnection::PassiveOpen   ()   {
        _state->PassiveOpen(this);
}
```

```
void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

Known Uses

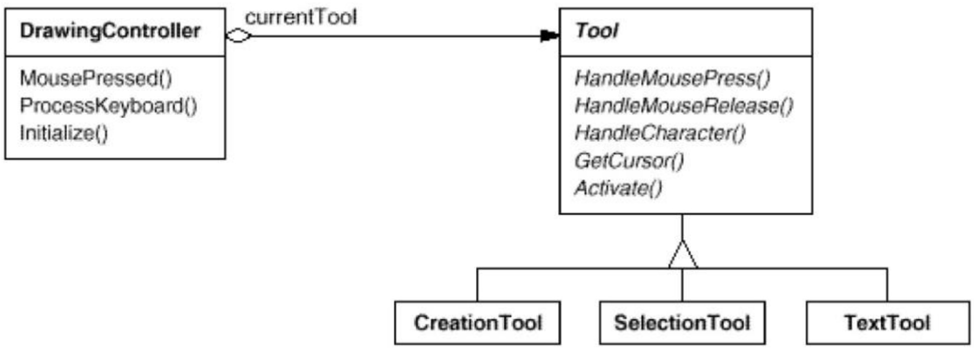
Johnson and Zweig [JZ91] characterize theState pattern and its application to TCP connection protocols.

Most popular interactive drawing programs provide "tools" forperforming operations by direct manipulation. For example, aline-drawing tool lets a user click and drag to create a new line. Aselection tool lets the user select shapes. There's usually a paletteof such tools to choose from. The user thinks of this activity aspicking up a tool and wielding it, but in reality the editor'sbehavior changes with the current tool: When a drawing tool is activewe create shapes;
when the selection tool is active we select shapes;and so forth. We can use the State pattern to change the editor'sbehavior depending on the current tool.

We can define an abstract Tool class from which to define subclassesthat implement tool-specific behavior. The drawing editor maintains acurrent Tool object and delegates requests to it. It replaces thisobject when the user chooses a new tool, causing the behavior of thedrawing editor to change accordingly.

This technique is used in both the HotDraw [Joh92] and Unidraw [VL90] drawing editor frameworks. It allows clients to define new kinds of tools easily. In HotDraw, the

DrawingController class forwards the requests to the current Tool object. In Unidraw, the corresponding classes are Viewer and Tool. The following class diagram sketches the Tool and DrawingController interfaces:



Coplien's Envelope-Letter idiom [Cop92] is related toState. Envelope-Letter is a technique for changing an object's class atrun-time. The State pattern is more specific, focusing on how to dealwith an object whose behavior depends on its state.

Related Patterns

The Flyweight pattern explains when and how State objects can be shared.

State objects are often Singletons .