UNIT – 5 **Behavior Patterns**

Strategy

Intent

Define a family of algorithms, encapsulate each one, and make theminterchangeable.

Strategy lets the algorithm vary independently fromclients that use it.

Also Known As

Policy

Motivation

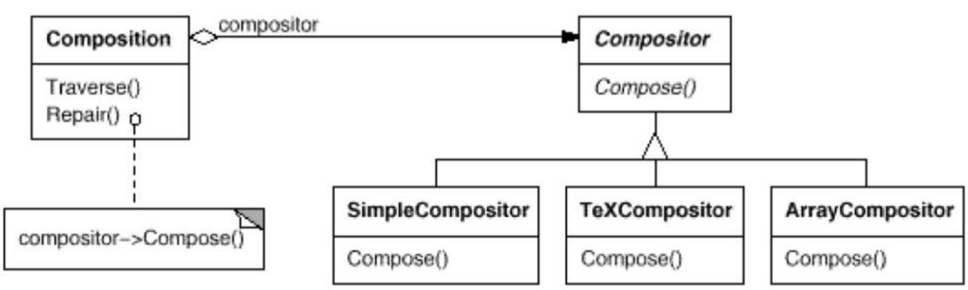
Many algorithms exist for breaking a stream of text into lines.Hard-wiring all such algorithms into the classes that require themisn't desirable for several reasons:

Clients that need linebreaking get more complex if they includethe linebreaking code. That makes clients bigger and harder tomaintain, especially if they support multiple linebreaking algorithms.

Different algorithms will be appropriate at different times. We don'twant to support multiple linebreaking algorithms if we don't use themall.

It's difficult to add new algorithms and vary existing ones whenlinebreaking is an integral part of a client.

We can avoid these problems by defining classes that encapsulatedifferent linebreaking algorithms. An algorithm that's encapsulated inthis way is called a strategy.



Suppose a Composition class is responsible for maintaining andupdating the linebreaks of text displayed in a text viewer.Linebreaking strategies aren't implemented by the class

Composition.Instead, they are implemented separately by subclasses of the abstractCompositor class. Compositor subclasses implement different strategies:

- SimpleCompositorimplements a simple strategy that determines linebreaks one at atime.
- TeXCompositorimplements the TeX algorithm for finding linebreaks. This strategytries to optimize linebreaks globally, that is, one paragraph at atime.
- ArrayCompositorimplements a strategy that selects breaks so that each row has a fixednumber of items. It's useful for breaking a collection of icons intorows, for example.

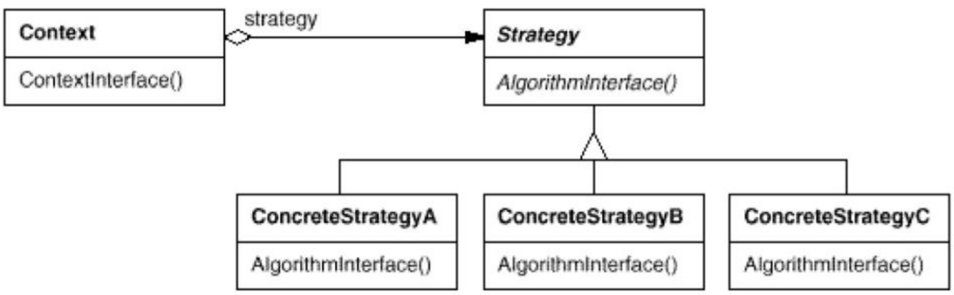
A Composition maintains a reference to a Compositor object. Whenever aComposition reformats its text, it forwards this responsibility to itsCompositor object. The client of Composition specifies whichCompositor should be used by installing the Compositor it desires intothe Composition.

Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategiesprovide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might definealgorithms reflecting different space/time trade-offs.Strategies can be used when these variants are implemented as a classhierarchy of algorithms [HO87].
- an algorithm uses data that clients shouldn't know about. Use theStrategy pattern to avoid exposing complex, algorithm-specific datastructures.
- a class defines many behaviors, and these appear as multipleconditional statements in its operations. Instead of manyconditionals, move related conditional branches into their ownStrategy class.

Structure



Participants

Strategy (Compositor)

declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy (SimpleCompositor, TeXCompositor,ArrayCompositor)

implements the algorithm using the Strategy interface.

Context (Composition)

is configured with a ConcreteStrategy object.

- maintains a reference to a Strategy object.
- may define an interface that lets Strategy access its data.

Collaborations

Strategy and Context interact to implement the chosen algorithm. Acontext may pass all data required by the algorithm to the strategywhen the algorithm is called. Alternatively, the context can passitself as an argument to Strategy operations. That lets the strategycall back on the context as required.

A context forwards requests from its clients to its strategy. Clientsusually create and pass a ConcreteStrategy object to the context;thereafter, clients interact with the context exclusively. There isoften a family of ConcreteStrategy classes for a client to choosefrom.

Consequences

The Strategy pattern has the following benefits and drawbacks:

*Families of related algorithms.*Hierarchies of Strategy classes define afamily of algorithms orbehaviors for contexts to reuse. Inheritance canhelp factor out common functionality of the algorithms.

*An alternative to subclassing.*Inheritance offers another way to supporta variety of algorithms orbehaviors. You can subclass a Context class directly to give itdifferent behaviors. But this hard-wires the behavior into Context.It mixes the algorithm implementation with Context's, making Contextharder to understand, maintain, and extend. And you can't vary thealgorithm dynamically. You wind up with many related classes whoseonly difference is the algorithm or behavior they employ.

Encapsulating the algorithm in separate Strategy classes lets you varythe algorithm independently of its context, making it easier toswitch, understand, and extend.

*Strategies eliminate conditional statements.*The Strategy pattern offersan alternative to conditional statements forselecting desired behavior. When different behaviors are lumped into oneclass, it's hard to avoid using conditional statements to select theright behavior. Encapsulating the behavior in separate Strategy classeseliminates these conditional statements.

For example, without strategies, the code for breakingtext into lines could look like

```
void Composition::Repair () {
    switch  (_breakingStrategy)  {  case
    SimpleStrategy:
        ComposeWithSimpleCompositor();
        break;          case
    TeXStrategy:
        ComposeWithTeXCompositor();
        break;
        // ...
    }
    // merge results with existing composition, if necessary
}
```

The Strategy pattern eliminates this case statement by delegating thelinebreaking task to a Strategy object:

```
void   Composition::Repair   ()   {   _compositor-
        >Compose();
        // merge results with existing composition, if necessary
}
```

Code containing many conditional statements often indicatesthe need to apply the Strategy pattern.

*A choice of implementations.*Strategies can provide differentimplementations of the *same*behavior. The client can choose among strategies with differenttime and space trade-offs.

*Clients must be aware of different Strategies.*The pattern has a potentialdrawback in that a client must understandhow Strategies differ before it can select the appropriate one.Clients might be exposed to implementation issues. Therefore youshould use the Strategy pattern only when the variation in behavior isrelevant to clients.

*Communication overhead between Strategy and Context.*The Strategy interfaceis shared by all ConcreteStrategy classeswhether the algorithms they implement are trivial or complex. Henceit's likely that some ConcreteStrategies won't use all the informationpassed to them through this interface; simple ConcreteStrategies mayuse none of it! That means there will be times when the contextcreates and initializes parameters that never get used. If this is anissue, then you'll need tighter coupling between Strategy and Context.

*Increased number of objects.*Strategies increase the number of objects inan application. Sometimesyou can reduce this overhead by implementing strategies as statelessobjects that contexts can share. Any residual state is maintained by thecontext, which passes it in each request to the Strategy object. Sharedstrategies should not maintain state across invocations. The Flyweight (218) pattern describes this approach in moredetail.

Implementation

Consider the following implementation issues:

*Defining the Strategy and Context interfaces.*The Strategy and Contextinterfaces must give a ConcreteStrategyefficient access to any data it needs from a context, and vice versa.

One approach is to have Context pass data in parameters to Strategyoperations in other words, take the data to the strategy. This keepsStrategy and Context decoupled. On the other hand, Context mightpass data the Strategy doesn't need.

Another technique has a context pass *itself* as an argument, andthe strategy requests data from the context explicitly.Alternatively, the strategy can store a reference to its context,eliminating the need to pass anything at all. Either way, thestrategy can request exactly what it needs. But now Context mustdefine a more elaborate interface to its data, which couples Strategyand Context more closely.

The needs of the particular algorithm and its data requirements willdetermine

the best technique.

*Strategies as template parameters.*In C++ templates can be used to configurea class with a strategy.This technique is only applicable if (1) the Strategy can be selectedat compile-time, and (2) it does not have to be changed at run-time.In this case, the class to be configured (e.g., Context) isdefined as a template class that has a Strategy class

*Making Strategy objects optional.*The Context class may be simplified ifit's meaningful *not* tohave a Strategy object. Context checks to see if it has a Strategyobject before accessing it. If there is one, then Context uses itnormally. If there isn't a strategy, then Context carries out defaultbehavior. The benefit of this approach is that clients don't have todeal with Strategy objects at all *unless* they don't like thedefault behavior.

Sample Code

We'll give the high-level code for the Motivation example, which isbased on the implementation of Composition and Compositor classes inInterViews [LCI+92].

The Composition class maintains a collection ofComponent instances, which represent text and graphicalelements in a document. A composition arranges component objects intolines using an instance of a Compositor subclass, whichencapsulates a linebreaking strategy. Each component has anassociated natural size, stretchability, and shrinkability. Thestretchability defines how much the component can grow beyond itsnatural size;

shrinkability is how much it can shrink. Thecomposition passes these values to a compositor, which uses them todetermine the best location for linebreaks.

```
class Composition { public:
        Composition(Compositor*);  void
        Repair();
private:
        Compositor* _compositor;
        Component* _components;  // the list of components
        int _componentCount;   // the number of components
        int _lineWidth;          // the Composition's line width
int* _lineBreaks;        // the position of linebreaks
                                     // in components
int _lineCount;          // the number of lines
};
```

When a new layout is required, the composition asks its compositor todetermine where to place linebreaks. The composition passes thecompositor three arrays that define natural sizes, stretchabilities,and shrinkabilities of the components. It also passes the number ofcomponents, how wide the line is, and an array that the compositorfills with the position of each linebreak. The compositor returns thenumber of calculated breaks.

The Compositor interface lets the composition pass thecompositor all the information it needs. This is an example of"taking the data to the strategy":

```
class Compositor {
public:
        virtual int Compose(
                Coord natural[], Coord stretch[], Coord shrink[],
                int componentCount, int lineWidth, int breaks[]
        ) = 0;
protected:
        Compositor();
};
```

Note that Compositor is an abstract class. Concretesubclasses define specific linebreaking strategies.

The composition calls its compositor in its Repairoperation. Repair first initializes arrays with the naturalsize, stretchability, and shrinkability of each component (the detailsof which we omit for brevity). Then it calls on the compositor toobtain the linebreaks and finally lays out the components according tothe breaks (also omitted):

```
void Composition::Repair () { Coord*
        natural; Coord* stretchability;
        Coord*    shrinkability;    int
        componentCount; int* breaks;

        // prepare the arrays with the desired component sizes
// ...

          determine where the breaks are: int
        breakCount;
        breakCount = _compositor->Compose(
                natural,       stretchability,       shrinkability,
                componentCount, _lineWidth, breaks
        );

          lay out components according to breaks
          ...
}
```

Now let's look at the Compositor subclasses.SimpleCompositor examines components a line at a time todetermine where breaks should go:

```
class SimpleCompositor : public Compositor {
public:
        SimpleCompositor();
        virtual int Compose(
```

```
            Coord natural[], Coord stretch[], Coord shrink[],
            int componentCount, int lineWidth, int breaks[]
    );
```

Known Uses

Both ET++ [WGM88] and InterViews use strategies to encapsulatedifferent linebreaking algorithms as we've described.

In the RTL System for compiler code optimization [JML92],strategies define different register allocation schemes(RegisterAllocator) and instruction set scheduling policies(RISCscheduler, CISCscheduler). This provides flexibility in targeting theoptimizer for different machine architectures.

The ET++SwapsManager calculation engine framework computes prices fordifferent financial instruments [EG92]. Its keyabstractions are Instrument and YieldCurve. Different instruments areimplemented as subclasses of Instrument. YieldCurve calculatesdiscount factors, which determine the present value of future cashflows. Both of these classes delegate some behavior to Strategyobjects. The framework provides a family of ConcreteStrategy classesfor generating cash flows, valuing swaps, and calculating discountfactors. You can create new calculation engines by configuringInstrument and YieldCurve with the different ConcreteStrategy objects.This approach supports mixing and matching existing Strategyimplementations as well as defining new ones.

Related Patterns

Flyweight : Strategy objects often make good flyweights.

Template Method

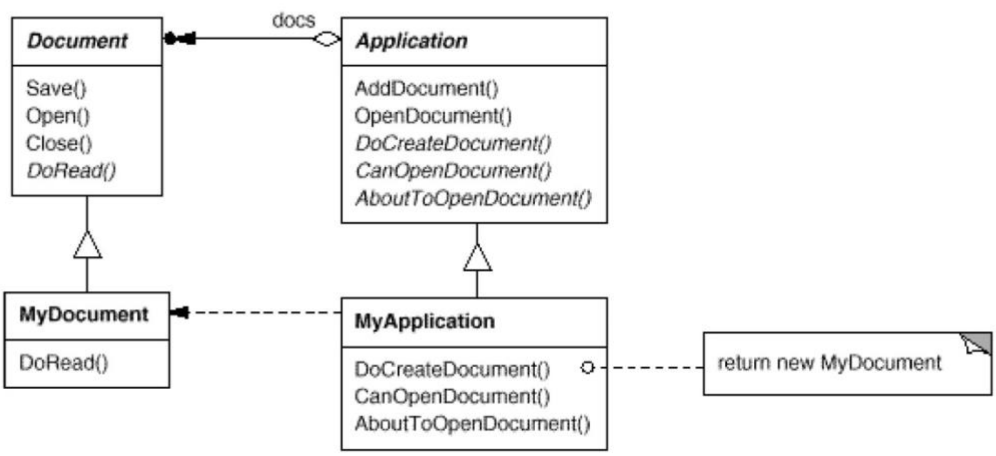
Intent

Define the skeleton of an algorithm in an operation, deferring somesteps to subclasses. Template Method lets subclasses redefinecertain steps of an algorithm without changing the algorithm'sstructure.

Motivation

Consider an application framework that provides Application andDocument classes. The Application class is responsible for openingexisting documents stored in an external format, such as a file. ADocument object represents the information in a document once it'sread from the file.

Applications built with the framework can subclass Application andDocument to suit specific needs. For example, a drawing applicationdefines DrawApplication and DrawDocument subclasses; a spreadsheetapplication defines SpreadsheetApplication and SpreadsheetDocumentsubclasses.



The abstract Application class defines the algorithm for opening andreading a document in its OpenDocument operation:

```
void Application::OpenDocument (const char* name) {
	if (!CanOpenDocument(name)) {
		// cannot handle this document
		return;}
```

```
Document* doc = DoCreateDocument(); if
(doc) {
    _docs->AddDocument(doc);
AboutToOpenDocument(doc);
doc->Open();        doc-
>DoRead();
    }
}
```

OpenDocument defines each step for opening a document. It checks ifthe document can be opened, creates the application-specific Documentobject, adds it to its set of documents, and reads the Document from afile.

We call OpenDocument a template method. A template methoddefines an algorithm in terms of abstract operations that subclassesoverride to provide concrete behavior. Application subclasses definethe steps of the algorithm that check if the document can be opened(CanOpenDocument) and that create the Document (DoCreateDocument).Document classes define the step that reads the document (DoRead).The template method also defines an operation that lets Applicationsubclasses know when the document is about to be opened(AboutToOpenDocument), in case they care.

By defining some of the steps of an algorithm using abstractoperations, the template method fixes their ordering, but it letsApplication and Document subclasses vary those steps to suit theirneeds.

Applicability

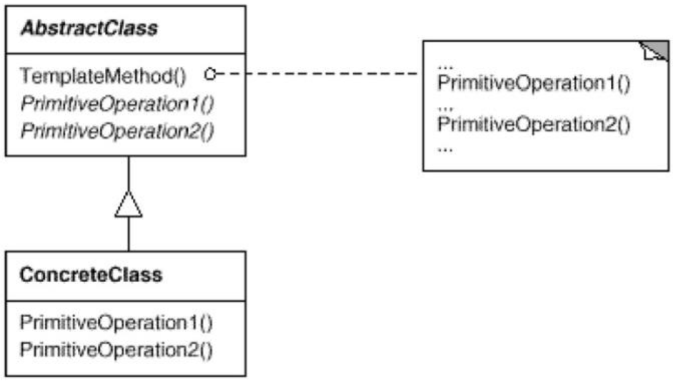
The Template Method pattern should be used

- to implement the invariant parts of an algorithm once and leave it upto subclasses to implement the behavior that can vary.
- when common behavior among subclasses should be factored and localizedin a common class to avoid code duplication. This is a good example of"refactoring to generalize" as described by Opdyke andJohnson [OJ93]. You first identify

thedifferences in the existing code and then separate the differencesinto new operations. Finally, you replace the differing code with atemplate method that calls one of these new operations.

to control subclasses extensions. You can define a template methodthat calls "hook" operations (see Consequences) at specific points,thereby permitting extensions only at those points.

Structure



Participants

AbstractClass (Application)

defines abstract primitive operations that concretesubclasses define to implement steps of an algorithm.

implements a template method defining the skeleton of an algorithm.The template method calls primitive operations as well as operationsdefined in AbstractClass or those of other objects.

ConcreteClass (MyApplication)

implements the primitive operations to carry out subclass-

specificsteps of the algorithm.

Collaborations

ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences

Template methods are a fundamental technique for code reuse. They areparticularly important in class libraries, because they are the meansfor factoring out common behavior in library classes.

Template methods lead to an inverted control structure that'ssometimes referred to as "the Hollywood principle," that is, "Don'tcall us, we'll call you" [Swe85]. This refers tohow a parent class calls the operations of a subclass and not theother way around.

Template methods call the following kinds of operations:

- concrete operations (either on the ConcreteClass or onclient classes);
- concrete AbstractClass operations (i.e., operations that aregenerally useful to subclasses);
- primitive operations (i.e., abstract operations);
- factory methods (see Factory Method (121)); and
- hook operations, which provide default behavior thatsubclasses can extend if necessary. A hook operation often doesnothing by default.

It's important for template methods to specify which operations arehooks (*may* be overridden) and which are abstract operations(*must* be overridden). To reuse an abstract class effectively,subclass writers must understand which operations are designed foroverriding.

A subclass can *extend* a parent class operation's behavior byoverriding the operation and calling the parent operation explicitly:

```
void DerivedClass::Operation () {
        // DerivedClass extended behavior
        ParentClass::Operation();
}
```

Unfortunately, it's easy to forget to call the inherited operation.We can transform such an operation into a template method to givethe parent control over how subclasses extend it. The idea is tocall a hook operation from a template method in the parent class.

Implementation

Three implementation issues are worth noting:

*Using C++ access control.*In C++, the primitive operations that a templatemethod calls can bedeclared protected members. This ensures that they are only called bythe template method. Primitive operations that *must* be overridden aredeclared pure virtual. The template method itself should not beoverridden; therefore you can make the template method a nonvirtualmember function.

*Minimizing primitive operations.*An important goal in designing templatemethods is to minimize thenumber of primitive operations that a subclass must override to fleshout the algorithm. The more operations that need overriding, the moretedious things get for clients.

*Naming conventions.*You can identify the operations that should beoverridden by adding aprefix to their names. For example, the MacApp framework for Macintoshapplications [App89] prefixes template method names with "Do-":"DoCreateDocument", "DoRead", and so forth.

Sample Code

The following C++ example shows how a parent class can enforce aninvariant for its subclasses. The example comes from NeXT'sAppKit [Add94]. Consider a class View that supportsdrawing on the screen. View enforces the invariant that itssubclasses can draw into a view only after it becomes the "focus,"which requires certain drawing state (for example, colors and fonts) tobe set up properly.

We can use a Display template method to set up this state.View defines two concrete operations,SetFocus and ResetFocus, that set up and clean upthe drawing state, respectively. View's DoDisplayhook operation performs the actual drawing. Display callsSetFocus before DoDisplay to set up the drawingstate; Display calls ResetFocus afterwards torelease the drawing state.

```
void View::Display () {
        SetFocus();
        DoDisplay();
        ResetFocus();
}
```

To maintain the invariant, the View's clients always callDisplay, and View subclasses always overrideDoDisplay.

DoDisplay does nothing in View:

```
void View::DoDisplay () { }
```

Subclasses override it to add their specific drawing behavior:

```
void MyView::DoDisplay () {
        // render the view's contents
}
```

Known Uses

Template methods are so fundamental that they can be found in almostevery abstract class. Wirfs-Brock et al. [WBWW90, WBJ90] provide a good overview anddiscussion of template methods.

Related Patterns

Factory Methods are often called by template methods. In the Motivation example,the factory method DoCreateDocument is called by the template methodOpenDocument.

Strategy : Template methods use inheritance to vary part of an algorithm.Strategies use delegation to vary the entire algorithm.

Visitor

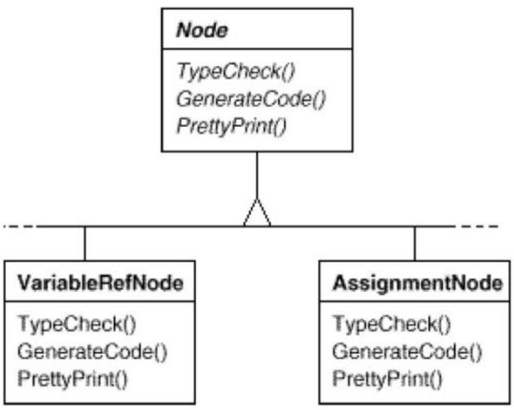
Intent

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

Motivation

Consider a compiler that represents programs as abstract syntax trees.It will need to perform operations on abstract syntax trees for "staticsemantic" analyses like checking that all variables are defined. Itwill also need to generate code. So it might define operations fortype-checking, code optimization, flow analysis, checking for variablesbeing assigned values before they're used, and so on.
Moreover, we coulduse the abstract syntax trees for pretty-printing, programrestructuring, code instrumentation, and computing various metrics of aprogram.

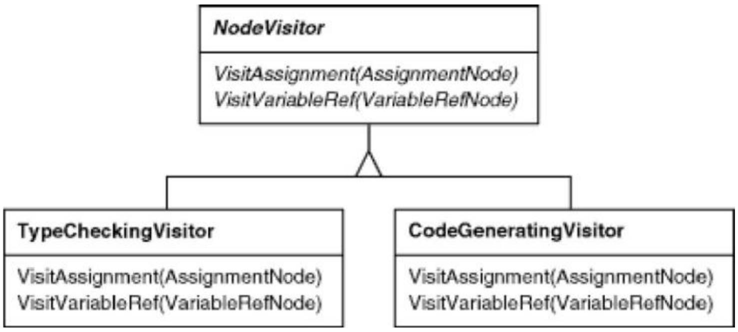
Most of these operations will need to treat nodes that representassignment statements differently from nodes that represent variables orarithmetic expressions. Hence there will be one class for assignmentstatements, another for variable accesses, another for arithmeticexpressions, and so on. The set of node classes depends on the languagebeing compiled, of course, but it doesn't change much for a givenlanguage.



This diagram shows part of the Node class hierarchy. The problem hereis that distributing all these operations across the various nodeclasses leads to a system that's hard to understand, maintain, andchange. It will be confusing to have type-checking code mixed withpretty-printing code or flow analysis code. Moreover, adding a newoperation usually requires recompiling all of these classes. It would bebetter if each new operation could be added separately, and the nodeclasses were independent of the operations that apply to them.

We can have both by packaging related operations from each class in aseparate object, called a visitor, and passing it toelements of the abstract syntax tree as it's traversed. When an element"accepts" the visitor, it sends a request to the visitor that encodesthe element's class. It also includes the element as an argument. Thevisitor will then execute the operation for that element theoperation that used to be in the class of the element.

For example, a compiler that didn't use visitors might type-check aprocedure by calling the TypeCheck operation on its abstract syntaxtree. Each of the nodes would implement TypeCheck by calling TypeCheckon its components (see the preceding class diagram). If the compilertype-checked a procedure using visitors, then it would create aTypeCheckingVisitor object and call the Accept operation on theabstract syntax tree with that object as an argument. Each of thenodes would implement Accept by calling back on the visitor: anassignment node calls VisitAssignment operation on the visitor, whilea variable reference calls VisitVariableReference. What used to be theTypeCheck operation in class AssignmentNode is now the VisitAssignmentoperation on TypeCheckingVisitor.



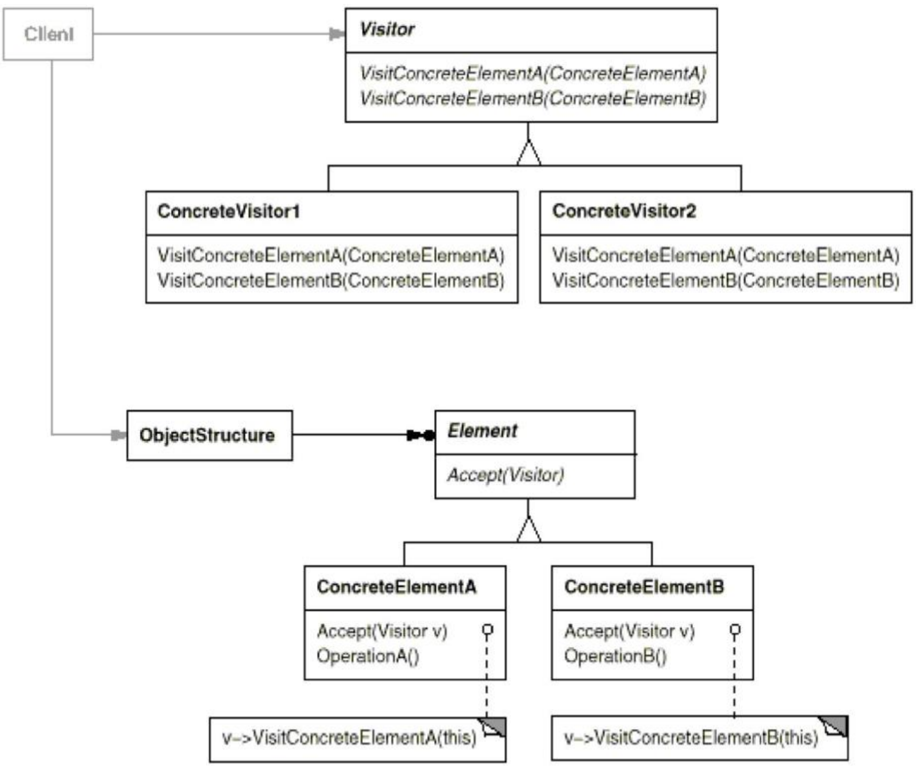
With the Visitor pattern, you define two class hierarchies: one for theelements being operated on (the Node hierarchy) and one for the visitorsthat define operations on the elements (the NodeVisitor hierarchy). Youcreate a new operation by adding a new subclass to the visitor classhierarchy. As long as the grammar that the compiler accepts doesn'tchange (that is, we don't have to add new Node subclasses), we can addnew functionality simply by defining new NodeVisitor subclasses.

Applicability

Use the Visitor pattern when

- an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- many distinct and unrelated operations need to be performed on objectsin an object structure, and you want to avoid "polluting" theirclasses with these operations. Visitor lets you keep related operationstogether by defining them in one class. When the object structure isshared by many applications, use Visitor to put operations in just thoseapplications that need them.
- the classes defining the object structure rarely change, but you oftenwant to define new operations over the structure. Changing the objectstructure classes requires redefining the interface to all visitors,which is potentially costly. If the object structure classes changeoften, then it's probably better to define the operations in those classes.

Structure



Participants

- Visitor (NodeVisitor)

  declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.

ConcreteVisitor (TypeCheckingVisitor)

  implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

Element (Node)

  defines an Accept operation that takes a visitor as an argument.

ConcreteElement (AssignmentNode,VariableRefNode)

  implements an Accept operation that takes a visitor as an argument.

ObjectStructure (Program)

  can enumerate its elements.

  may provide a high-level interface to allow the visitor to visit its elements.

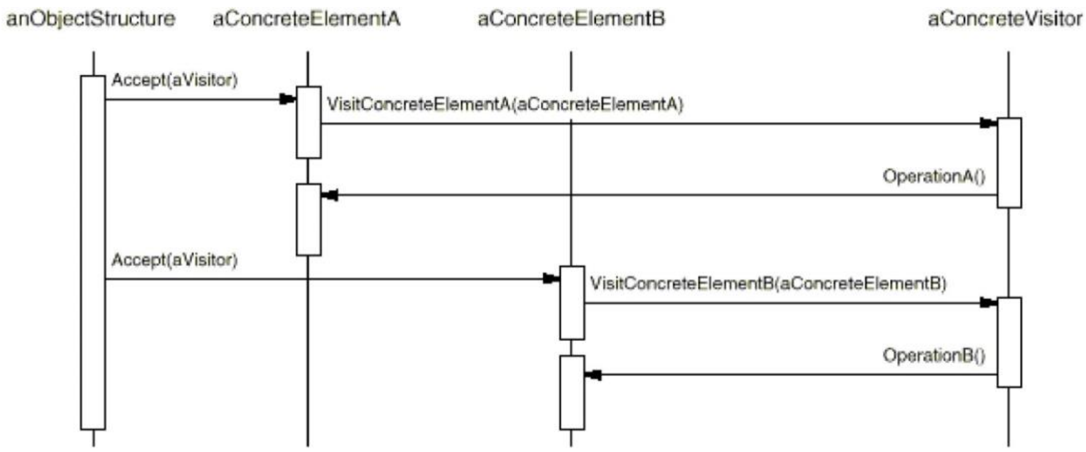
  may either be a composite (see Composite (183)) or a collection such as a list or a set.

Collaborations

A client that uses the Visitor pattern must create a ConcreteVisitorobject and then traverse the object structure, visiting each elementwith the visitor.

When an element is visited, it calls the Visitor operation thatcorresponds to its class. The element supplies itself as an argumentto this operation to let the visitor access its state, if necessary.

The following interaction diagram illustrates the collaborationsbetween an object structure, a visitor, and two elements:



Consequences

Some of the benefits and liabilities of the Visitor pattern are as follows:

*Visitor makes adding new operations easy.*Visitors make it easy to addoperations that depend on the components ofcomplex objects. You can define a new operation over an object structuresimply by adding a new visitor. In contrast, if you spread functionalityover many classes, then you must change each class to define a newoperation.

*A visitor gathers related operations and separates unrelated ones.*Relatedbehavior isn't spread over the classes defining the objectstructure; it's localized in a visitor. Unrelated sets of behavior arepartitioned in their own visitor subclasses. That simplifies both theclasses defining the elements and the algorithms defined in thevisitors. Any algorithm-specific data structures can be hidden in thevisitor.

*Adding new ConcreteElement classes is hard.*The Visitor pattern makes ithard to add new subclasses of Element. Eachnew ConcreteElement gives rise to a new abstract operation on Visitor anda corresponding implementation in every ConcreteVisitor class. Sometimes adefault implementation can be provided in Visitor that can be inheritedby most of the ConcreteVisitors, but this is the exception rather thanthe rule.

};

Implementation

Each object structure will have an associated Visitor class. Thisabstract visitor class declares a VisitConcreteElement operation foreach class of ConcreteElement defining the object structure. EachVisit operation on the Visitor declares its argument to be aparticular ConcreteElement, allowing the Visitor to access theinterface of the ConcreteElement directly. ConcreteVisitor classesoverride each Visit operation to implement visitor-specific behaviorfor the corresponding ConcreteElement class.

The Visitor class would be declared like this in C++:

```
class Visitor { public:
        virtual void VisitElementA(ElementA*);

        virtual void VisitElementB(ElementB*);

        // and so on for other concrete elements
protected:
        Visitor();
};
```

Each class of ConcreteElement implements an Accept operationthat calls the matching Visit... operation on the visitorfor that ConcreteElement. Thus the operation that ends up gettingcalled depends on both the class of the element and the class of thevisitor.[10]

The concrete elements are declared as

```
class Element {
public:
        virtual ~Element();
        virtual void Accept(Visitor&) = 0;
protected:
        Element();
};

class ElementA : public Element { public:
```

```
        ElementA();
        virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};

class ElementB : public Element { public:
        ElementB();
        virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};
```

A CompositeElement class might implement Acceptlike this:

```
class CompositeElement : public Element {
public:
        virtual void Accept(Visitor&);
private:
        List<Element*>* _children;
};

void CompositeElement::Accept (Visitor& v) {

        ListIterator<Element*> i(_children);

        for (i.First(); !i.IsDone(); i.Next()) {
                i.CurrentItem()->Accept(v);
        }
        v.VisitCompositeElement(this);
}
```

Here are two other implementation issues that arise when you apply theVisitor pattern:

*Double dispatch.*Effectively, the Visitor pattern lets you add operationsto classeswithout changing them. Visitor achieves this by using a techniquecalled double-dispatch. It's a well-known technique. Infact, some

Sample Code

Because visitors are usually associated with composites, we'll use theEquipment classes defined in the Sample Code of Composite to illustrate the Visitor pattern. Wewill use Visitor to define operations for computing theinventory of materials and the total cost for a piece of equipment.The Equipment classes are so simple that using Visitorisn't really necessary, but they make it easy to see what'sinvolved in implementing the pattern.

Here again is the Equipment class from Composite .We've augmented it with anAccept operation to let it work with a visitor.

```
class Equipment { public:
        virtual ~Equipment();

        const char* Name() { return _name; }

        virtual    Watt    Power();    virtual
        Currency NetPrice();
        virtual Currency DiscountPrice();

        virtual    void    Accept(EquipmentVisitor&);
protected:
        Equipment(const char*);
private:
        const char* _name;
};
```

The Equipment operations return the attributes of a piece ofequipment, such as its power consumption and cost. Subclasses redefinethese operations appropriately for specific types of equipment (e.g.,a chassis, drives, and planar boards).

The abstract class for all visitors of equipment has a virtualfunction for each subclass of equipment, as shown next. All of thevirtual functions do nothing by default.

```
class EquipmentVisitor { public:
```

```
	virtual ~EquipmentVisitor();

	virtual void VisitFloppyDisk(FloppyDisk*);
	virtual void VisitCard(Card*);
	virtual void VisitChassis(Chassis*); virtual

	void VisitBus(Bus*);

	// and so on for other concrete subclasses of Equipment
protected:
	EquipmentVisitor();
};
```

Equipment subclasses define Accept inbasically the same way: It calls theEquipmentVisitor operation that corresponds to the classthat received the Accept request, like this:

```
void FloppyDisk::Accept (EquipmentVisitor& visitor) {
	visitor.VisitFloppyDisk(this);
}
```

Equipment that contains other equipment (in particular, subclasses ofCompositeEquipment in the Composite pattern) implementsAccept by iterating over its children and callingAccept on each of them. Then it calls theVisit operation as usual.For example, Chassis::Accept could traverseall the parts in the chassis as follows:

Known Uses

The Smalltalk-80 compiler has a Visitor class called ProgramNodeEnumerator.It's used primarily for algorithms that analyze source code.It isn't used for code generation or pretty-printing, although it could be.

IRIS Inventor [Str93]is a toolkit for developing 3-D graphics applications. Inventorrepresents a three-dimensional scene as a hierarchy of nodes, eachrepresenting either a geometric object or an attribute of one.Operations like rendering a scene or mapping an input event requiretraversing this hierarchy in different ways. Inventor does thisusing visitors called "actions." There are different visitors forrendering, event handling, searching, filing, and determiningbounding boxes.

To make adding new nodes easier, Inventor implements adouble-dispatch scheme for C++. The scheme relies on run-time typeinformation and a two-dimensional table in which rows representvisitors and columns represent node classes. The cells store apointer to the function bound to the visitor and node class.

Mark Linton coined the term "Visitor" in the X Consortium'sFresco Application Toolkit specification [LP93].

Related Patterns

Composite:Visitors can be used to apply an operation over an object structuredefined by the Composite pattern.

Interpreter :Visitor may be applied to do the interpretation.

Discussion of Behavioral Patterns

Encapsulating Variation

Encapsulating variation is a theme of many behavioral patterns. Whenan aspect of a program changes frequently, these patterns define anobject that encapsulates that aspect. Then other parts of the programcan collaborate with the object whenever they depend on that aspect.The patterns usually define an abstract class that describes theencapsulating object, and the pattern derives its name from thatobject.[12]For example,

a Strategy object encapsulates an algorithm (Strategy (349)),
a State object encapsulates a state-dependent behavior (State (338)),
a Mediator object encapsulates the protocol betweenobjects (Mediator (305)), and
an Iterator object encapsulates the way you access and traverse thecomponents of an aggregate object (Iterator (289)).

These patterns describe aspects of a program that are likely tochange. Most patterns have two kinds of objects: the new object(s)that encapsulate the aspect, and the existing object(s) that use thenew ones. Usually the functionality of new objects would be anintegral part of the existing objects were it not for the pattern. Forexample, code for a Strategy would probably be wired into thestrategy's Context, and code for a State would be implemented directlyin the state's Context.

But not all object behavioral patterns partition functionality likethis. For example, Chain of Responsibility dealswith an arbitrary number of objects (i.e., a chain), all of which mayalready exist in the system.

Chain of Responsibility illustrates another difference in behavioralpatterns: Not all define static communication relationships betweenclasses. Chain of Responsibility prescribes communication between anopen-ended number of objects. Other patterns involve objects that arepassed around as arguments.

Objects as Arguments

Several patterns introduce an object that's *always* usedas an argument. One of these is Visitor . A Visitor object is theargument to a polymorphic Accept operation on the objects it visits.The visitor is never considered a part of those objects, even thoughthe conventional alternative to the pattern is to distribute Visitorcode across the object structure classes.
Other patterns define objects that act as magic tokens to be passedaround and invoked at a later time. Both Command and Memento) fall into this category. In Command,the token represents a request; in Memento, it represents the internalstate of an object at a particular time.In both cases, the token can have a complex internal representation,but the client is never aware of it. But even here there aredifferences. Polymorphism is important in the Command pattern,because executing the Command object is a polymorphic operation. Incontrast, the Memento interface is so narrow that a memento can onlybe passed as a value. So it's likely to present no polymorphicoperations at all to its clients.

Should Communication be Encapsulated or Distributed?

Mediator and Observer arecompeting patterns. The difference between them is that Observerdistributes communication by introducing Observer and Subject objects,whereas a Mediator object encapsulates the communication between otherobjects.

In the Observer pattern, there is no single object that encapsulates aconstraint. Instead, the Observer and the Subject must cooperate tomaintain the constraint. Communication patterns are determined by theway observers and subjects are interconnected: a single subjectusually has many observers, and sometimes the observer of one subjectis a subject of another observer. The Mediator pattern centralizesrather than distributes. It places the responsibility for maintaininga constraint squarely in the mediator.

We've found it easier to make reusable Observers and Subjects than tomake reusable Mediators. The Observer pattern promotes partitioningand loose coupling between Observer and Subject, and that leads tofiner-grained classes that are more apt to be reused.

On the other hand, it's easier to understand the flow of communicationin Mediator than in Observer. Observers and subjects are usuallyconnected shortly after they're created, and it's hard to see how theyare connected later in the program. If you know the Observer pattern,then you understand that the way observers and subjects are connectedis important, and you also know what connections to look for.However, the indirection that Observer introduces will still make asystem harder to understand.

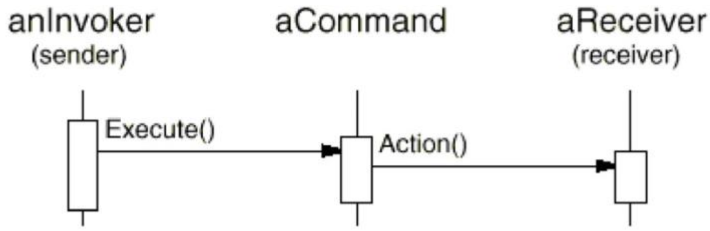
Observers in Smalltalk can be parameterized with messages to accessthe Subject state, and so they are even more reusable than they are inC++. This makes Observer more attractive than Mediator in Smalltalk.Thus a Smalltalk programmer will often use Observer where a C++programmer would use Mediator.

Decoupling Senders and Receivers

When collaborating objects refer to each other directly, they becomedependent on each other, and that can have an adverse impact on thelayering and reusability of a system.

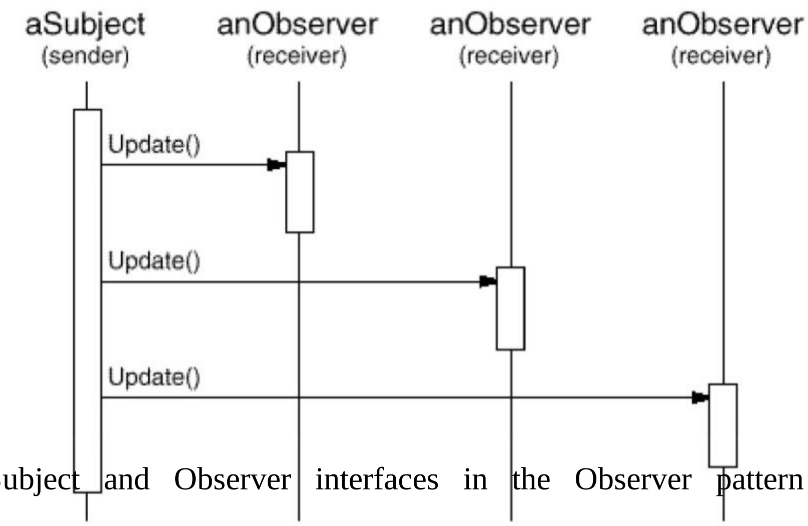
Command, Observer, Mediator,and Chain of Responsibility address how you can decouple senders andreceivers, but with different trade-offs.

The Command pattern supports decoupling by using a Command object todefine the binding between a sender and receiver:



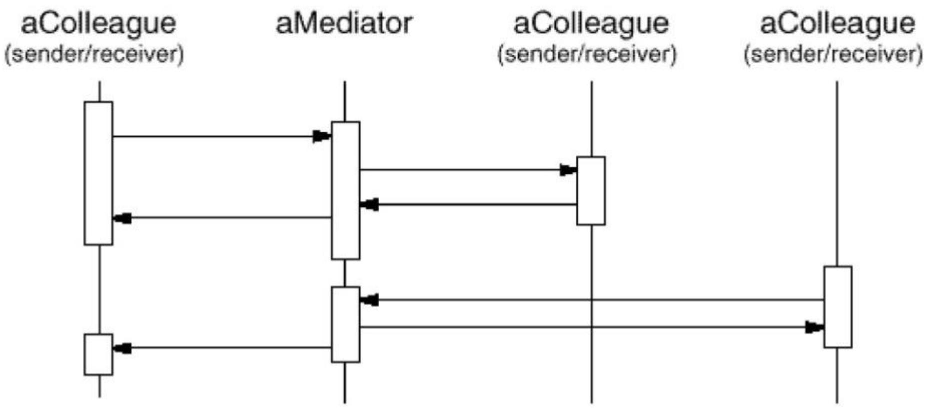
The Command object provides a simple interface for issuing the request(that is, the Execute operation). Defining the sender-receiverconnection in a separate object lets the sender work with differentreceivers. It keeps the sender decoupled from the receivers, makingsenders easy to reuse.

The Observer pattern decouples senders (subjects) from receivers(observers) by defining an interface for signaling changes insubjects. Observer defines a looser sender-receiver binding thanCommand, since a subject may have multiple observers, and their numbercan vary at run-time.



The Subject and Observer interfaces in the Observer pattern aredesigned for communicating changes. Therefore the Observer pattern isbest for decoupling objects when there are data dependencies betweenthem.

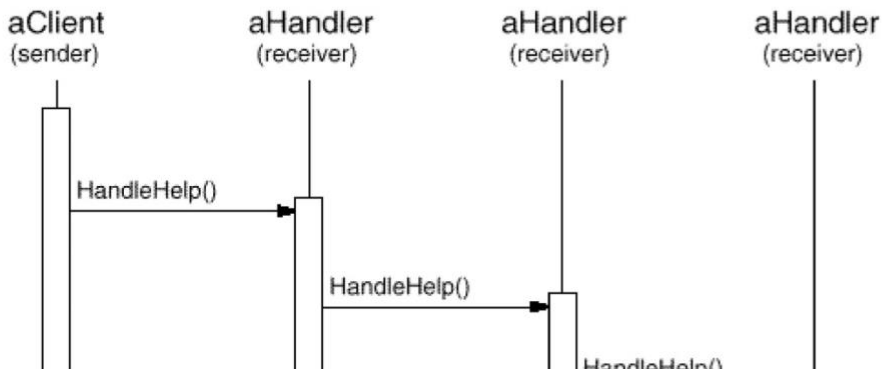
The Mediator pattern decouples objects by having them refer to eachother indirectly through a Mediator object.



A Mediator object routes requests between Colleague objects andcentralizes their communication. Consequently, colleagues can onlytalk to each other through the Mediator interface. Because thisinterface is fixed, the Mediator might have to implement its owndispatching scheme for added flexibility. Requests can be encoded andarguments packed in such a way that colleagues can request anopen-ended set of operations.

The Mediator pattern can reduce subclassing in a system, because itcentralizes communication behavior in one class instead ofdistributing it among subclasses. However, *ad hoc* dispatching schemesoften decrease type safety.

Finally, the Chain of Responsibility pattern decouples the sender fromthe receiver by passing the request along a chain of potentialreceivers:



Since the interface between senders and receivers is fixed, Chain ofResponsibility may also require a custom dispatching scheme. Hence ithas the same type-safety drawbacks as Mediator. Chain ofResponsibility is a good way to decouple the sender and the receiverif the chain is already part of the system's structure, and one ofseveral objects may be in a position to handle the request. Moreover,the pattern offers added flexibility in that the chain can be changed orextended easily.

Summary

With few exceptions, behavioral design patterns complement andreinforce each other. A class in a chain of responsibility, forexample, will probably include at least one application of Template Method.The template method can useprimitive operations to determine whether the object should handle therequest and to choose the object to forward to. The chain can use theCommand pattern to represent requests as objects. Interpreter can use the State pattern todefine parsing contexts. An iterator can traverse an aggregate, and avisitor can apply an operation to each element in the aggregate.

Behavioral patterns work well with other patterns, too. For example, asystem that uses the Composite pattern might usea visitor to perform operations on components of thecomposition. It could use Chain of Responsibility to let componentsaccess global properties through their parent. It could also use Decorator to override these properties on partsof the composition. It could use the Observer pattern to tie oneobject structure to another and the State pattern to let a component change its behavior as its state changes. The composition itself might be created using the approach in Builder , and it might be treated as a Prototype by some other part of the system.

Well-designed object-oriented systems are just like this they have multiple patterns embedded in them but not because their designers necessarily thought in these terms. Composition at the *pattern* level rather than the class or object levels lets us achieve the same synergy with greater ease.