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

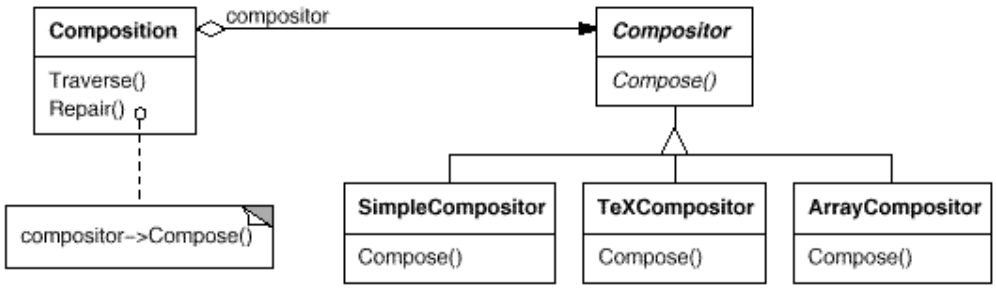
Policy

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:

- **SimpleCompositor**implements a simple strategy that determines linebreaks one at atime.
- **TeXCompositor**implements the TeX algorithm for finding linebreaks. This strategytries to optimize linebreaks globally, that is, one paragraph at atime.
- **ArrayCompositor**implements 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.

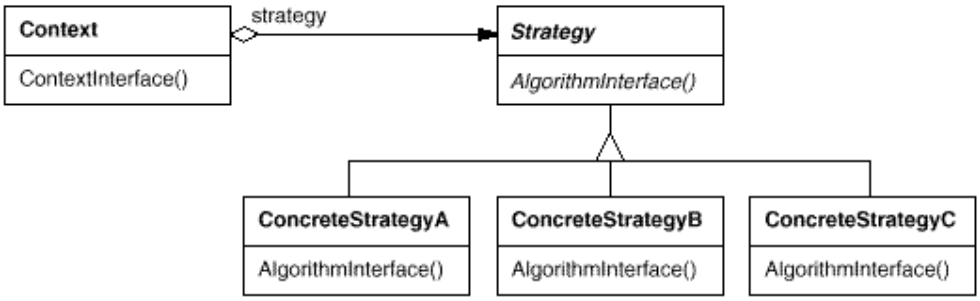
A Composition maintains a reference to a Compositor object. Whenever aComposition reformats its text, it forwards this responsibility to itsCompositor object. The client of Composition specifies whichCompositor should be used by installing the Compositor it desires intothe Composition.

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:

1. *Families of related algorithms.*Hierarchies of Strategy classes define a family of algorithms orbehaviors for contexts to reuse. Inheritance canhelp factor out common functionality of the algorithms.
2. *An alternative to subclassing.*Inheritance offers another way to support a 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.
3. *Strategies eliminate conditional statements.*The Strategy pattern offers an 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.

4. *A choice of implementations.*Strategies can provide different implementations of the *same*behavior. The client can choose among strategies with differenttime and space trade-offs.
5. *Clients must be aware of different Strategies.*The pattern has a potential drawback 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.
6. *Communication overhead between Strategy and Context.*The Strategy interface is 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.
7. *Increased number of objects.*Strategies increase the number of objects in an 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:

1. *Defining the Strategy and Context interfaces.*The Strategy and Context interfaces 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.

2. *Strategies as template parameters.* In C++ templates can be used to configure a 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 as aparameter:

```
template <class AStrategy>
class Context {
        void Operation() { theStrategy.DoAlgorithm(); }
                // ...
private:
        AStrategy theStrategy;
};
```

The class is then configured with a Strategy class when it's instantiated:

```
class MyStrategy {
public:
        void DoAlgorithm();
};

Context<MyStrategy> aContext;
```

With templates, there's no need to define an abstract class that defines the interface to the Strategy. Using Strategy as atemplate parameter also lets you bind a Strategy to itsContext statically, which can increase efficiency.

3. *Making Strategy objects optional.* The Context class may be simplified if it'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[]
        );
```

```
        // ...
};
```

TeXCompositor uses a more global strategy. It examines a*paragraph* at a time, taking into account the components' sizeand stretchability. It also tries to give an even "color" to theparagraph by minimizing the whitespace between components.

```
class TeXCompositor : public Compositor {
public:
        TeXCompositor();
        virtual int Compose(
                Coord natural[], Coord stretch[], Coord shrink[],
                int componentCount, int lineWidth, int breaks[]
        );
        // ...
};
```

ArrayCompositor breaks the components into lines at regularintervals.

```
class ArrayCompositor : public Compositor {
public:
        ArrayCompositor(int interval);
        virtual int Compose(
                Coord natural[], Coord stretch[], Coord shrink[],
                int componentCount, int lineWidth, int breaks[]
        );
        // ...
};
```

These classes don't use all the information passed inCompose. SimpleCompositor ignores the stretchabilityof the components, taking only their natural widths into account.TeXCompositor uses all the information passed to it, whereasArrayCompositor ignores everything.

To instantiate Composition, you pass it the compositoryou want to use:

```
        Composition* quick = new Composition(new SimpleCompositor);
        Composition* slick = new Composition(new TeXCompositor);
        Composition* iconic = new Composition(new ArrayCompositor(100));
```

Compositor's interface is carefully designed to support alllayout algorithms that subclasses might implement. You don't want tohave to change this interface with every new subclass, because that willrequire changing existing subclasses. In general, the Strategy andContext interfaces determine how well the pattern achieves its intent.

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.

The Booch components [BV90] use strategies as templatearguments. The Booch collection classes support three different kinds ofmemory allocation strategies: managed (allocation out of a pool),controlled (allocations/deallocations are protected by locks), andunmanaged (the normal memory allocator). These strategies are passed astemplate arguments to a collection class when it's instantiated. Forexample, an UnboundedCollection that uses the unmanaged strategy isinstantiated as UnboundedCollection.

RApp is a system for integrated circuit layout [GA89, AG90].RApp must lay out and route wires that connect subsystems on thecircuit. Routing algorithms in RApp are defined assubclasses of an abstract Router class. Router is a Strategy class.

Borland's ObjectWindows [Bor94] uses strategies in dialogsboxes to ensure that the user enters valid data. For example, numbers mighthave to be in a certain range, and a numeric entry field should acceptonly digits. Validating that a string is correct can require atable look-up.

ObjectWindows uses Validator objects to encapsulate validationstrategies. Validators are examples of Strategy objects. Data entryfields delegate the validation strategy to an optional Validatorobject. The client attaches a validator to a field if validation isrequired (an example of an optional strategy). When the dialog isclosed, the entry fields ask their validators to validate the data.The class library provides validators for common cases, such as

aRangeValidator for numbers. New client-specific validation strategiescan be defined easily by subclassing the Validator class.

Related Patterns

Flyweight (218): Strategy objects often make good flyweights.

Template Method

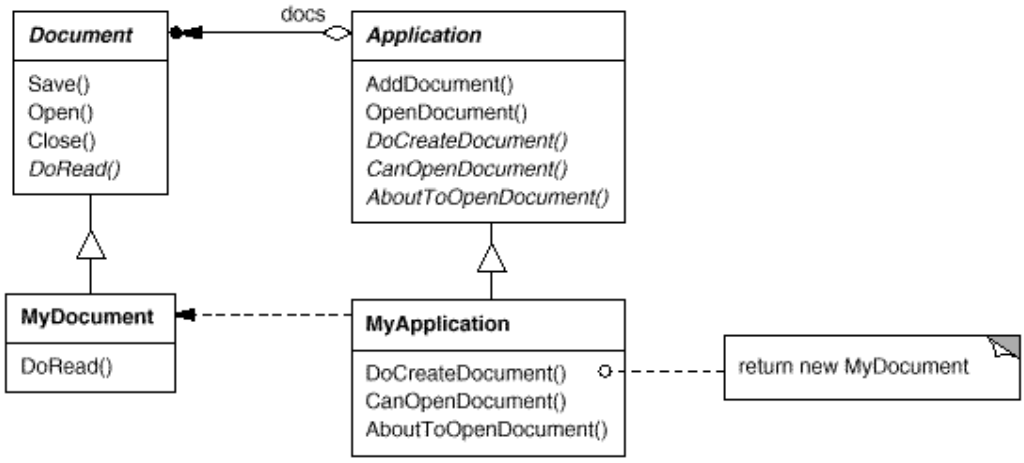
Intent

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.

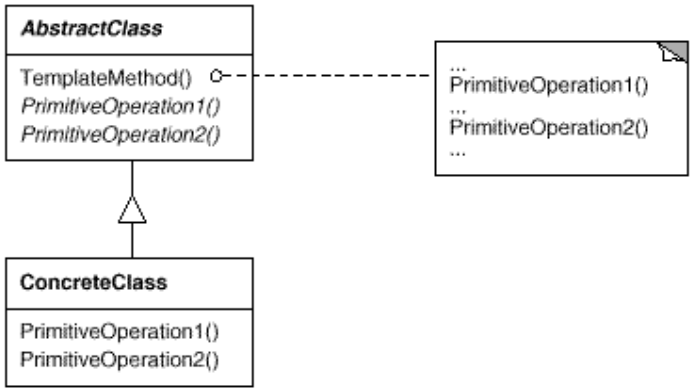
By defining some of the steps of an algorithm using abstractoperations, the template method fixes their ordering, but it letsApplication and Document subclasses vary those steps to suit theirneeds.

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.Then subclasses can then override this hook operation:

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

HookOperation does nothing in ParentClass:

```
void ParentClass::HookOperation () { }
```

Subclasses override HookOperation to extend itsbehavior:

```
void DerivedClass::HookOperation () {
        // derived class extension
}
```

Implementation

Three implementation issues are worth noting:

1. *Using C++ access control.* In C++, the primitive operations that a template method 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.
2. *Minimizing primitive operations.* An important goal in designing template methods 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.
3. *Naming conventions.* You can identify the operations that should be overridden 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 (121) are often called by template methods. In the Motivation example,the factory method DoCreateDocument is called by the template methodOpenDocument.

Strategy (349): Template methods use inheritance to vary part of an algorithm.Strategies use delegation to vary the entire algorithm.

Visitor

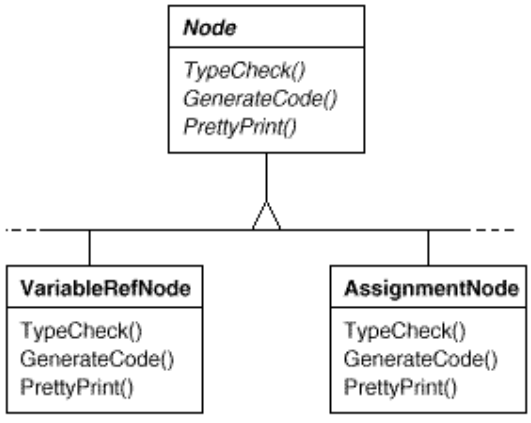
Intent

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.



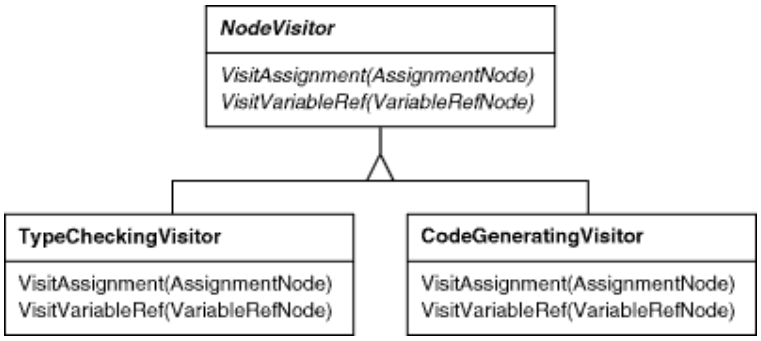
This diagram shows part of the Node class hierarchy. The problem hereis that distributing all these operations across the various nodeclasses leads to a system that's hard to understand, maintain, andchange. It will be confusing to have

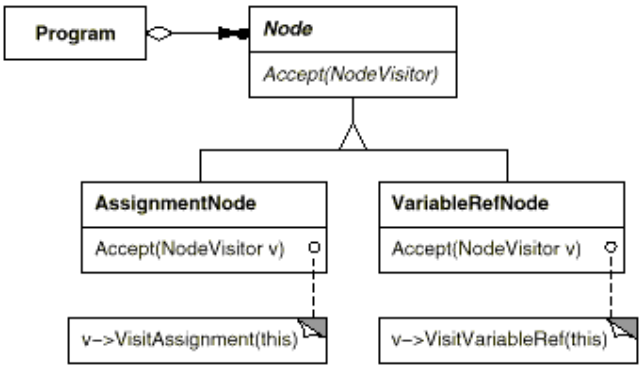
type-checking code mixed withpretty-printing code or flow analysis code. Moreover, adding a newoperation usually requires recompiling all of these classes. It would bebetter if each new operation could be added separately, and the nodeclasses were independent of the operations that apply to them.

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.

To make visitors work for more than just type-checking, we need anabstract parent class NodeVisitor for all visitors of an abstract syntaxtree. NodeVisitor must declare an operation for each node class. Anapplication that needs to compute program metrics will define newsubclasses of NodeVisitor and will no longer need to add application-specific code to the node classes. The Visitor pattern encapsulates the operations for each compilation phase in a Visitor associated with that phase.





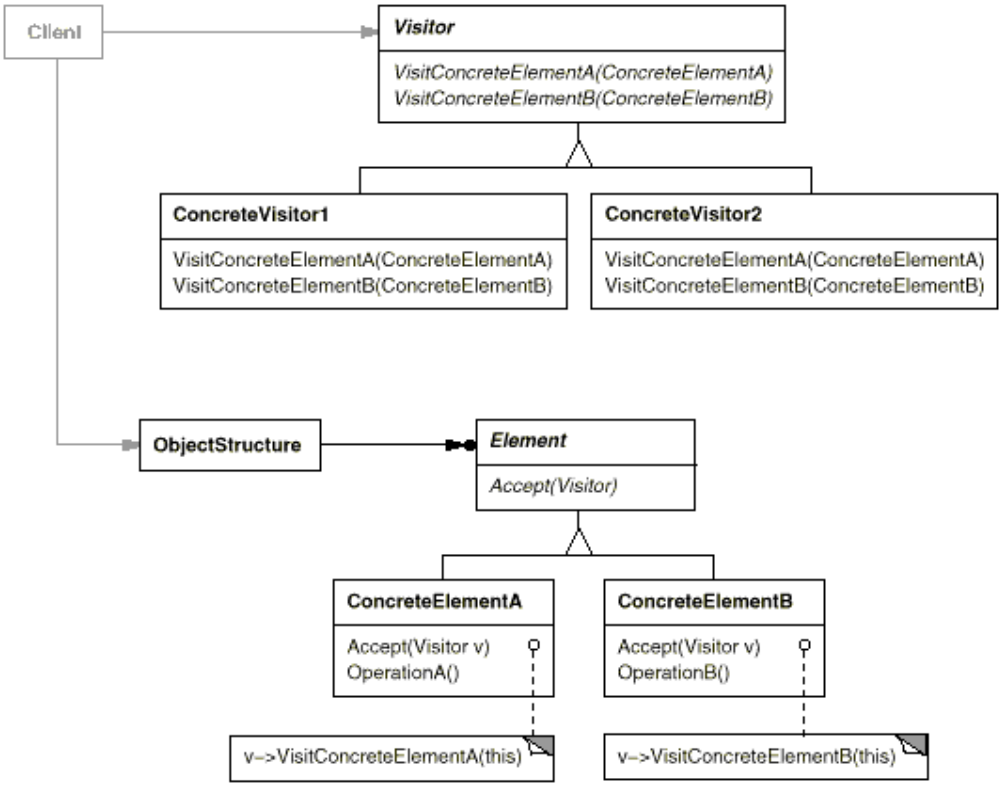
With the Visitor pattern, you define two class hierarchies: one for theelements being operated on (the Node hierarchy) and one for the visitorsthat define operations on the elements (the NodeVisitor hierarchy). Youcreate a new operation by adding a new subclass to the visitor classhierarchy. As long 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

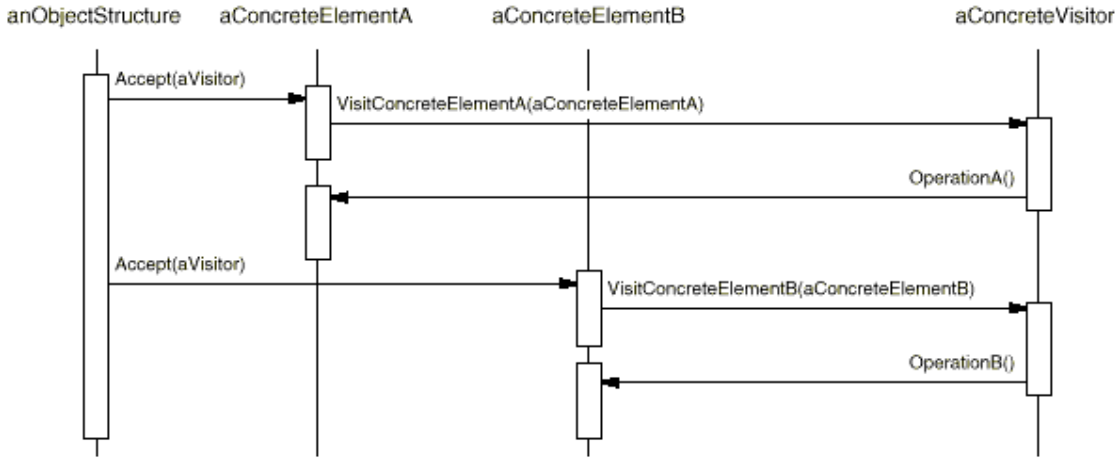
- **Visitor** (NodeVisitor)
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then 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:

1. *Visitor makes adding new operations easy.*Visitors make it easy to add operations 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.

2. *A visitor gathers related operations and separates unrelated ones.* Related behavior 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.
3. *Adding new ConcreteElement classes is hard.* The Visitor pattern makes it hard 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.

   So the key consideration in applying the Visitor pattern is whether youare mostly likely to change the algorithm applied over an objectstructure or the classes of objects that make up the structure. TheVisitor class hierarchy can be difficult to maintain when newConcreteElement classes are added frequently. In such cases, it'sprobably easier just to define operations on the classes that make upthe structure. If the Element class hierarchy is stable, but you arecontinually adding operations or changing algorithms, then the Visitorpattern will help you manage the changes.

4. *Visiting across class hierarchies.* An iterator (see Iterator (289)) can visit the objects in astructure as it traverses them by calling their operations. But an iteratorcan't work across object structures with different types of elements. Forexample, the Iterator interface defined on page 295 can access only objects of type Item:

```
template <class Item>
class Iterator {
        // ...
        Item CurrentItem() const;
};
```

   This implies that all elements the iterator can visit have a common parentclass Item.

   Visitor does not have this restriction. It can visit objects thatdon't have a common parent class. You can add any type of object to aVisitor interface. For example, in

```
class Visitor {
public:
        // ...
        void VisitMyType(MyType*);
```

```
        void VisitYourType(YourType*);
};
```

MyType and YourType do not have to be related throughinheritance at all.

5. *Accumulating state.*Visitors can accumulate state as they visit each element in the objectstructure. Without a visitor, this state would be passed as extraarguments to the operations that perform the traversal, or theymight appear as global variables.
6. *Breaking encapsulation.*Visitor's approach assumes that the ConcreteElement interface is powerfulenough to let visitors do their job. As a result, the pattern oftenforces you to provide public operations that access an element'sinternal state, which may compromise its encapsulation.

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:

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

programming languages support it directly (CLOS, forexample). Languages like C++ and Smalltalk support**single-dispatch**.

In single-dispatch languages, two criteria determine which operationwill fulfill a request: the name of the request and the type ofreceiver. For example, the operation that a GenerateCode request willcall depends on the type of node object you ask. In C++, callingGenerateCode on an instance of VariableRefNode willcall VariableRefNode::GenerateCode (which generates code for avariable reference). Calling GenerateCode on anAssignmentNode will callAssignmentNode::GenerateCode (which will generate code for anassignment). The operation that gets executed depends both on the kindof request and the type of the receiver.

"Double-dispatch" simply means the operation that gets executeddepends on the kind of request and the types of *two* receivers.Accept is a double-dispatch operation. Its meaning dependson two types: the Visitor's and the Element's. Double-dispatchinglets visitors request different operations on each class ofelement.[11]

This is the key to the Visitor pattern: The operation that getsexecuted depends on both the type of Visitor and the type of Elementit visits. Instead of binding operations statically into the Elementinterface, you can consolidate the operations in a Visitor and useAccept to do the binding at run-time. Extending the Elementinterface amounts to defining one new Visitor subclass rather than many newElement subclasses.

2. *Who is responsible for traversing the object structure?*A visitor must visit each element of the object structure. The questionis, how does it get there? We can put responsibility for traversal inany of three places: in the object structure, in the visitor, or in aseparate iterator object (see Iterator (289)).

   Often the object structure is responsible for iteration. A collectionwill simply iterate over its elements, calling the Accept operation oneach. A composite will commonly traverse itself by having each Acceptoperation traverse the element's children and call Accept on each ofthem recursively.

   Another solution is to use an iterator to visit the elements. In C++,you could use either an internal or external iterator, depending on whatis available and what is most efficient. In Smalltalk, you usually usean internal iterator using do: and a block. Since internaliterators are implemented by the object structure, using an internaliterator is a lot like making the object structure responsible foriteration. The main difference is that an internal iterator will notcause double-dispatching—it will call an operation on the *visitor* with an *element*

as an argument as opposed to calling anoperation on the *element* with the *visitor* as an argument.But it's easy to use the Visitor pattern with an internal iterator ifthe operation on the visitor simply calls the operation on the elementwithout recursing.

You could even put the traversal algorithm in the visitor, although you'llend up duplicating the traversal code in each ConcreteVisitor for eachaggregate ConcreteElement. The main reason to put the traversal strategyin the visitor is to implement a particularly complex traversal, onethat depends on the results of the operations on the object structure.We'll give an example of such a case in the Sample Code.

Sample Code

Because visitors are usually associated with composites, we'll use theEquipment classes defined in the Sample Code of Composite (183) 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 (183). 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:

```
void Chassis::Accept (EquipmentVisitor& visitor) {
        for (
                ListIterator i(_parts);
                !i.IsDone();
                i.Next()
        ) {
                i.CurrentItem()->Accept(visitor);
        }
        visitor.VisitChassis(this);
}
```

Subclasses of EquipmentVisitor define particular algorithmsover the equipment structure. The PricingVisitor computes thecost of the equipment structure. It

computes the net price of all simpleequipment (e.g., floppies) and the discount price of all compositeequipment (e.g., chassis and buses).

```
class PricingVisitor : public EquipmentVisitor {
public:
          PricingVisitor();

          Currency& GetTotalPrice();

          virtual void VisitFloppyDisk(FloppyDisk*);
          virtual void VisitCard(Card*);
          virtual void VisitChassis(Chassis*);
          virtual void VisitBus(Bus*);
          // ...
private:
          Currency _total;
};

void PricingVisitor::VisitFloppyDisk (FloppyDisk* e) {
          _total += e->NetPrice();
}

void PricingVisitor::VisitChassis (Chassis* e) {
           _total += e->DiscountPrice();
}
```

PricingVisitor will compute the total cost of all nodes in theequipment structure. Note that PricingVisitor chooses theappropriate pricing policy for a class of equipment by dispatching tothe corresponding member function. What's more, we can change thepricing policy of an equipment structure just by changing thePricingVisitor class.

We can define a visitor for computing inventory like this:

```
class InventoryVisitor : public EquipmentVisitor {
public:
          InventoryVisitor();

          Inventory& GetInventory();

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

```
        // ...
private:
        Inventory _inventory;
};
```

The InventoryVisitor accumulates the totals for each type ofequipment in the object structure. InventoryVisitor uses anInventory class that defines an interface for adding equipment(which we won't bother defining here).

```
void InventoryVisitor::VisitFloppyDisk (FloppyDisk* e) {
        _inventory.Accumulate(e);
}

void InventoryVisitor::VisitChassis (Chassis* e) {
        _inventory.Accumulate(e);
}
```

Here's how we can use an InventoryVisitor on anequipment structure:

```
Equipment* component;
InventoryVisitor visitor;

component->Accept(visitor);
cout << "Inventory "
        << component->Name()
        << visitor.GetInventory();
```

Now we'll show how to implement the Smalltalk example from the Interpreter pattern (see page 279) with theVisitor pattern. Like the previous example, this one is so small thatVisitor probably won't buy us much, but it provides a good illustration ofhow to use the pattern. Further, it illustrates a situation in whichiteration is the visitor's responsibility.

The object structure (regular expressions) is made of four classes,and all of them have an accept: method that takes thevisitor as an argument. In class SequenceExpression, theaccept: method is

```
accept: aVisitor
        ^ aVisitor visitSequence: self
```

In class RepeatExpression, the accept: methodsends the visitRepeat: message.In class AlternationExpression, it sends thevisitAlternation: message.In class LiteralExpression, it sends thevisitLiteral: message.

The four classes also must have accessing functions that the visitorcan use. For SequenceExpression these areexpression1 and expression2; forAlternationExpression these are alternative1and alternative2; forRepeatExpression it is repetition; and forLiteralExpression these are components.

The ConcreteVisitor class is REMatchingVisitor. Itis responsible for the traversal because its traversal algorithmis irregular. The biggest irregularity is that aRepeatExpression will repeatedly traverse its component.The class REMatchingVisitor has an instance variableinputState. Its methods are essentially the same asthe match: methods of the expression classes in theInterpreter pattern except theyreplace the argument named inputState with theexpression node being matched. However, theystill return the set of streams that the expression would matchto identify the current state.

```
visitSequence: sequenceExp
        inputState := sequenceExp expression1 accept: self.
        ^ sequenceExp expression2 accept: self.

visitRepeat: repeatExp
        | finalState |
        finalState := inputState copy.
        [inputState isEmpty]
                whileFalse:
                        [inputState := repeatExp repetition accept: self.
                        finalState addAll: inputState].
        ^ finalState

visitAlternation: alternateExp
        | finalState originalState |
        originalState := inputState.
        finalState := alternateExp alternative1 accept: self.
        inputState := originalState.
        finalState addAll: (alternateExp alternative2 accept: self).
        ^ finalState

visitLiteral: literalExp
        | finalState tStream |
        finalState := Set new.
        inputState
                do:
                        [:stream | tStream := stream copy.
                                (tStream nextAvailable:
                                        literalExp components size
```

```
                                    ) = literalExp components
                                            ifTrue: [finalState add: tStream]
                            ].
        ^ finalState
```

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

Interpreter (274):Visitor may be applied to do the interpretation.

---

[10]We could use function overloading to give these operationsthe same simple name, like Visit, since the operations arealready differentiated by the parameter they're passed. There arepros and cons to such overloading. On the one hand, it reinforces thefact that each operation involves the same analysis, albeit on adifferent argument. On the other hand, that might make what's goingon at the

call site less obvious to someone reading the code. Itreally boils down to whether you believe function overloading is goodor not.

[11]If we can have *double*-dispatch, then why not*triple* or *quadruple*, or any other number? Actually,double-dispatch is just a special case of **multipledispatch**, in which the operation is chosen based on any number oftypes. (CLOS actually supports multiple dispatch.) Languages thatsupport double- or multiple dispatch lessen the need for theVisitor pattern.

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 (251) 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 (366). 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 (263) and Memento (316) 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 (305) and Observer (326) 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.

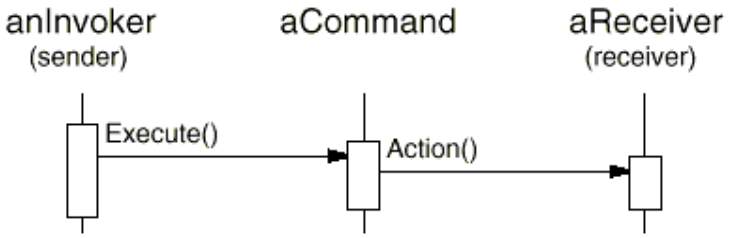
On the other hand, it's easier to understand the flow of communicationin Mediator than in Observer. Observers and subjects are usuallyconnected shortly after they're created, and it's hard to see how theyare connected later in the program. If you know the Observer pattern,then you understand that the 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

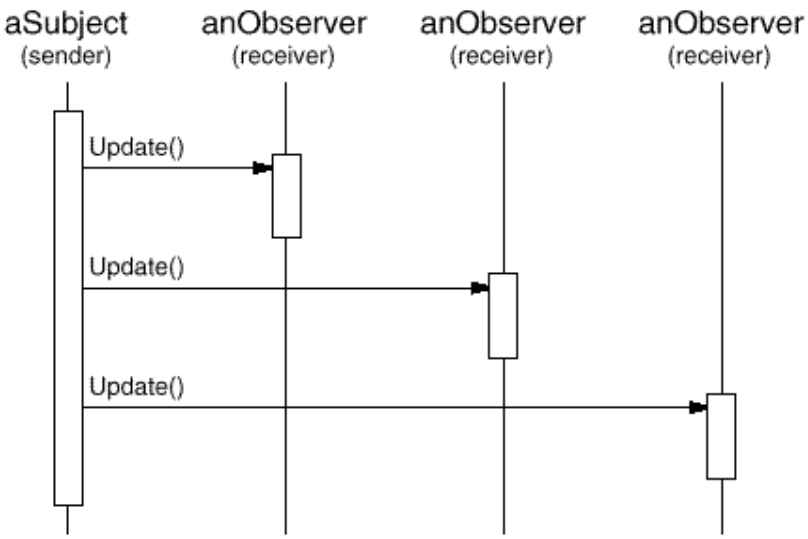
When collaborating objects refer to each other directly, they becomedependent on each other, and that can have an adverse impact on thelayering and reusability of a system. Command, Observer, Mediator,and Chain of Responsibility address how you can decouple senders andreceivers, but with different trade-offs.

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



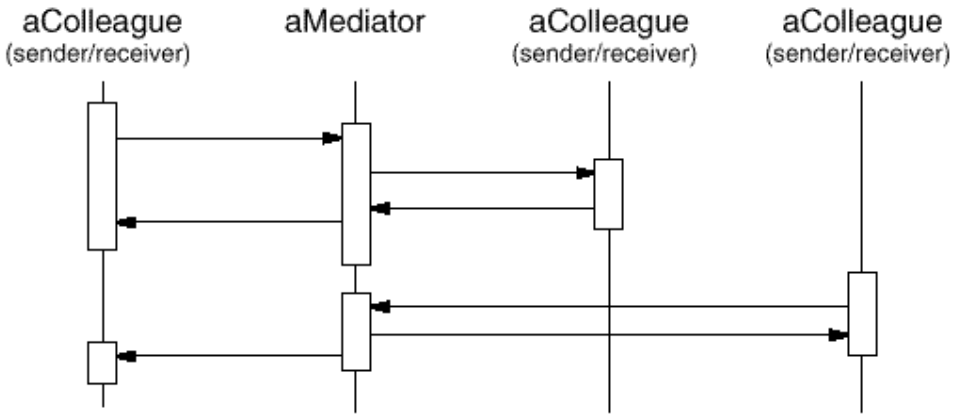
The Command object provides a simple interface for issuing the request(that is, the Execute operation). Defining the sender-receiverconnection in a separate object lets the sender work with differentreceivers. It keeps the sender decoupled from the receivers, makingsenders easy to reuse. Moreover, you can reuse the Command object toparameterize a receiver with different senders. The Command patternnominally requires a subclass for each sender-receiver connection,although the pattern describes implementation techniques that avoidsubclassing.

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.

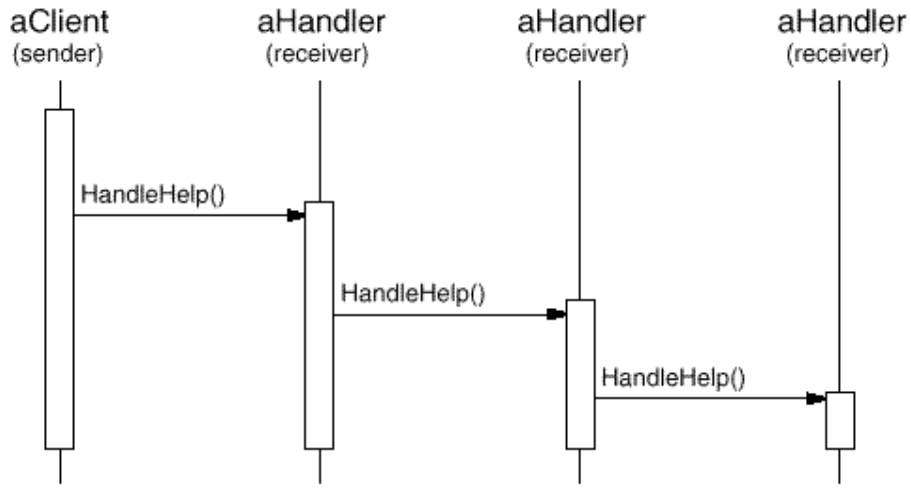
The Mediator pattern decouples objects by having them refer to eachother indirectly through a Mediator object.



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 (360).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 (274) 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 (183) 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 (196) 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 componentchange its behavior as its state changes. The composition itself mightbe created using the approach in Builder (110), and itmight be treated as a Prototype (133) by some otherpart of the system.

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

---

[12]This theme runs through other kinds of patterns, too. Abstract Factory (99), Builder (110), and Prototype (133) all encapsulate knowledge about howobjects are created. Decorator (196) encapsulates responsibility that can be added to an object. Bridge (171) separates an abstraction from its implementation, letting them vary independently.

6. Conclusion

It's possible to argue that this book hasn't accomplished much. Afterall, it doesn't present any algorithms or programming techniques thathaven't been used before. It doesn't give a rigorous method fordesigning systems, nor does it develop a new theory of design—itjust documents existing designs. You could conclude that itmakes a reasonable tutorial, perhaps, but it certainly can't offermuch to an experienced object-oriented designer.

We hope you think differently. Cataloging design patterns isimportant. It gives us standard names and definitions for thetechniques we use. If we don't study design patterns in software, wewon't be able to improve them, and it'll be harder to come up with newones.

This book is only a start. It contains some of the most common designpatterns that expert object-oriented designers use, and yet peoplehear and learn about them solely by word of mouth or by studyingexisting systems. Early drafts of the book prompted other people towrite down the design patterns they use, and it should prompt evenmore in its current form. We hope this will mark the start of amovement to document the expertise of software practitioners.

This chapter discusses the impact we think design patterns willhave, how they are related to other work in design, and how you canget involved in finding and cataloging patterns.

What to Expect from Design Patterns

Here are several ways in which the design patterns in this book canaffect the way you design object-oriented software, based on ourday-to-day experience with them.

A Common Design Vocabulary

Studies of expert programmers for conventional languages haveshown that knowledge and experience isn't organized simply aroundsyntax but in larger conceptual structures such as algorithms, datastructures and idioms [AS85, Cop92, Cur89, SS86], and plans for fulfilling a particulargoal [SE84]. Designers probably don't think about the notation they're usingfor recording the design as much as they try to match the currentdesign situation against plans, algorithms, data structures, andidioms they have learned in the past.

Computer scientists name and catalog algorithms and data structures,but we don't often name other kinds of patterns. Design patternsprovide a common vocabulary for designers to use to communicate, document,and explore design alternatives. Design patterns make a system seemless complex by letting you talk about it at a higher level ofabstraction than that of a design notation or programming language.Design patterns raise the level at which you design and discuss designwith your colleagues.

Once you've absorbed the design patterns in this book, your designvocabulary will almost certainly change. You will speak directly interms of the names of the design patterns. You'll find yourselfsaying things like, "Let's use an Observer here," or, "Let's make aStrategy out of these classes."

A Documentation and Learning Aid

Knowing the design patterns in this book makes it easier to understandexisting systems. Most large object-oriented systems use these designpatterns. People learning object-oriented programming often complainthat the systems they're working with use inheritance in convolutedways and that it's difficult to follow the flow of control. In largepart this is because they do not understand the design patterns in thesystem. Learning these design patterns will help you understandexisting object-oriented systems.

These design patterns can also make you a better designer. Theyprovide solutions to common problems. If you work withobject-oriented systems long enough, you'll probably learn thesedesign patterns on your own. But reading the book will help you learnthem much faster. Learning these patterns will help a novice act morelike an expert.

Moreover, describing a system in terms of the design patterns that ituses will make it a lot easier to understand. Otherwise, people willhave to reverse-engineer the design to unearth the patterns it uses.Having a common vocabulary means you don't have to describe the wholedesign pattern; you can just name it and expect your reader to knowit. A reader who doesn't know the patterns will have to look them upat first, but that's still easier than reverse-engineering.

We use these patterns in our own designs, and we've found theminvaluable. Yet we use the patterns in arguably naive ways. We usethem to pick names for classes, to think about and teach good design,and to describe designs in terms of the sequence of design patterns weapplied [BJ94]. It's easy to imagine moresophisticated ways of using patterns, such as pattern-based CASE toolsor hypertext documents. But patterns are a big help even withoutsophisticated tools.

An Adjunct to Existing Methods

Object-oriented design methods are supposed to promote good design, toteach new designers how to design well, and to standardize the waydesigns are developed. A design method typically defines a set ofnotations (usually graphical) for modeling various aspects of a design,along with a set of rules that govern how and when to use eachnotation. Design methods usually describe problems that occur in adesign, how to resolve them, and how to evaluate design. But theyhaven't been able to capture the experience of expert designers.

We believe our design patterns are an important piece that's beenmissing from object-oriented design methods. The design patterns showhow to use primitive techniques such as objects, inheritance, andpolymorphism. They show how to parameterize a system with analgorithm, a behavior, a state, or the kind of objects it's supposedto create. Design patterns provide a way to describe more of the"why" of a design and not just record the results of your decisions.The Applicability, Consequences, and Implementation sections of thedesign patterns help guide you in the decisions you have to make.

Design patterns are especially useful in turning an analysis modelinto an implementation model. Despite many claims that promise asmooth transition from object-oriented analysis to design, in practicethe transition is anything but smooth. A flexible and reusable designwill contain objects that aren't in the analysis model. Theprogramming language and class libraries you use affect the design.Analysis models often must be redesigned to make them reusable. Manyof the design patterns in the catalog address these issues, which iswhy we call them *design* patterns.

A full-fledged design method requires more kinds of patterns than justdesign patterns. There can also be analysis patterns, user interfacedesign patterns, or performance-tuning patterns. But the designpatterns are an essential part, one that's been missing until now.

A Target for Refactoring

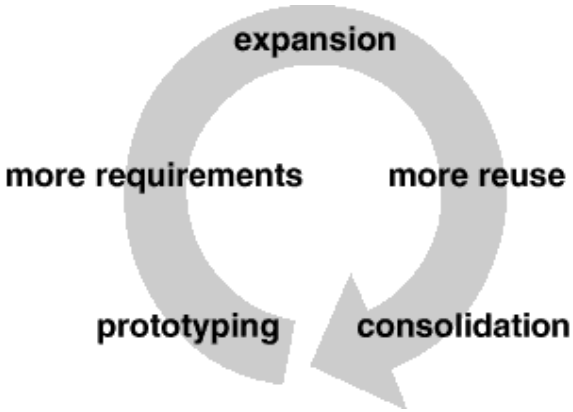
One of the problems in developing reusable software is that itoften has to be reorganized or **refactored** [OJ90]. Designpatterns help you determine how to reorganize a design, and theycan reduce the amount of refactoring you need to do later.

The lifecycle of object-oriented software has several phases.Brian Foote identifies these phases as the **prototyping**, **expansionary**, and **consolidating** phases [Foo92].

The prototyping phase is a flurry of activity as the software isbrought to life through rapid prototyping and incremental changes,until it meets an initial set of requirements and reaches adolescence.At this point, the software usually consists of class hierarchies thatclosely reflect entities in the initial problem domain. The main kindof reuse is white-box reuse by inheritance.

Once the software has reached adolescence and is put into service, itsevolution is governed by two conflicting needs: (1) the software mustsatisfy more requirements, and (2) the software must be more reusable.New requirements usually add new classes and operations and perhapswhole class hierarchies. The software goes through an expansionaryphase to meet new requirements. This can't continue for long,however. Eventually the software will become too inflexible andarthritic for further change. The class hierarchies will no longermatch any problem domain. Instead they'll reflect many problemdomains, and classes will define many unrelated operations andinstance variables.

To continue to evolve, the software must be reorganized in a processknown as *refactoring*. This is the phase in which frameworksoften emerge. Refactoring involves tearing apart classes into special-and general-purpose components, moving operations up or down the classhierarchy, and rationalizing the interfaces of classes. Thisconsolidation phase produces many new kinds of objects, often bydecomposing existing objects and using object composition instead ofinheritance. Hence black-box reuse replaces white-box reuse. Thecontinual need to satisfy more requirements along with the need formore reuse propels object-oriented software through repeated phases ofexpansion and consolidation—expansion as new requirements aresatisfied, and consolidation as the software becomes more general.



This cycle is unavoidable. But good designers are aware of thechanges that can prompt refactorings. Good designers also know classand object structures that can help avoid refactorings—their designsare robust in the face of requirement changes. A thoroughrequirements analysis will highlight those requirements that

arelikely to change during the life of the software, and a good designwill be robust to them.

Our design patterns capture many of the structures that result fromrefactoring. Using these patterns early in the life of a designprevents later refactorings. But even if you don't see how to apply apattern until after you've built your system, the pattern canstill show you how to change it. Design patterns thus provide targetsfor your refactorings.

A Brief History

The catalog began as a part of Erich's Ph.D.thesis [Gam91, Gam92]. Roughly half of thecurrent patterns were in his thesis. By OOPSLA '91 it was officiallyan independent catalog, and Richard had joined Erich to work on it.John started working on it soon thereafter. By OOPSLA '92, Ralph hadjoined the group. We worked hard to make the catalog fit forpublication at ECOOP '93, but soon we realized that a 90-page paperwas not going to be accepted. So we summarized the catalog andsubmitted the summary, which was accepted. We decided to turn thecatalog into a book shortly thereafter.

Our names for the patterns have changed a little along the way."Wrapper" became "Decorator," "Glue" became "Facade,""Solitaire" became "Singleton," and "Walker" became "Visitor."A couple of patterns got dropped because they didn't seem importantenough. But otherwise the set of patterns in the catalog has changedlittle since the end of 1992. The patterns themselves, however, haveevolved tremendously.

In fact, noticing that something is a pattern is the easy part.All four of us are actively working on building object-orientedsystems, and we've found that it's easy to spot patterns when youlook at enough systems. But *finding* patterns is mucheasier than *describing* them.

If you build systems and then reflect on what you build, you will seepatterns in what you do. But it's hard to describe patterns so thatpeople who don't know them will understand them and realize why theyare important. Experts immediately recognized the value of thecatalog in its early stages. But the only ones who could understandthe patterns were those who had already used them.

Since one of the main purposes of the book was to teachobject-oriented design to new designers, we knew we had to improve thecatalog. We expanded the average size of a pattern from less than 2to more than 10 pages by including a detailed motivating example and samplecode. We also started examining the trade-offs and the various waysof implementing the pattern. This made the patterns easier to learn.

Another important change over the past year has been a greater emphasis onthe problem that a pattern solves. It's easiest to see a pattern as asolution, as a technique that can be adapted and reused. It's harderto see when it is *appropriate*—to characterize the problems itsolves and the context in which it's the best solution. In general,it's easier to see *what* someone is doing than to know *why*,and the "why" for a pattern is the problem it solves. Knowingthe purpose of a pattern is important too, because it helps us choosepatterns to apply. It also helps us understand the design of existingsystems. A pattern author must determine and characterize the problemthat the pattern solves, even if you have to do it after you'vediscovered its solution.

The Pattern Community

We aren't the only ones interested in writing books that catalog thepatterns experts use. We are a part of a larger community interestedin patterns in general and software-related patterns in particular.Christopher Alexander is the architect who first studied patterns inbuildings and communities and developed a "pattern language" forgenerating them. His work has inspired us time and again. So it'sfitting and worthwhile to compare our work to his. Then we'll look atothers' work in software-related patterns.

Alexander's Pattern Languages

There are many ways in which our work is like Alexander's. Both arebased on observing existing systems and looking for patterns in them.Both have templates for describing patterns (although our templates arequite different). Both rely on natural language and lots of examplesto describe patterns rather than formal languages, and both giverationales for each pattern.

But there are just as many ways in which our works are different:

1. People have been making buildings for thousands of years, and thereare many classic examples to draw upon. We have been making softwaresystems for a relatively short time, and few are considered classics.
2. Alexander gives an order in which his patterns should be used; we havenot.
3. Alexander's patterns emphasize the problems they address, whereasdesign patterns describe the solutions in more detail.
4. Alexander claims his patterns will generate complete buildings. We donot claim that our patterns will generate complete programs.

When Alexander claims you can design a house simply by applying hispatterns one after another, he has goals similar to those ofobject-oriented design methodologists who give step-by-step rules fordesign. Alexander doesn't deny the

need for creativity; some of hispatterns require understanding the living habits of the people whowill use the building, and his belief in the "poetry" of designimplies a level of expertise beyond the pattern languageitself.[1]But his description of how patternsgenerate designs implies that a pattern language can make the designprocess deterministic and repeatable.

The Alexandrian point of view has helped us focus on designtrade-offs—the different "forces" that help shape a design. Hisinfluence made us work harder to understand the applicability andconsequences of our patterns. It also kept us from worrying aboutdefining a formal representation of patterns. Although such arepresentation might make automating patterns possible, at this stageit's more important to explore the space of design patterns than toformalize it.

From Alexander's point of view, the patterns in this book do not forma pattern language. Given the variety of software systems that peoplebuild, it's hard to see how we could provide a "complete" set ofpatterns, one that offers step-by-step instructions for designing anapplication. We can do that for certain classes of applications, suchas report-writing or making a forms-entry system. But our catalog isjust a collection of related patterns; we can't pretend it's a patternlanguage.

In fact, we think it's unlikely that there will *ever* be acomplete pattern language for software. But it's certainly possibleto make one that is *more* complete. Additions would have toinclude frameworks and how to use them [Joh92], patterns foruser interface design [BJ94], analysispatterns [Coa92], and all the other aspects of developingsoftware. Design patterns are just a part of a larger patternlanguage for software.

Patterns in Software

Our first collective experience in the study of software architecturewas at an OOPSLA '91 workshop led by Bruce Anderson. Theworkshop was dedicated to developing a handbook for softwarearchitects. (Judging from this book, we suspect "architectureencyclopedia" will be a more appropriate name than "architecturehandbook.") That first workshop has led to a series of meetings, themost recent of which being the first conference on Pattern Languagesof Programs held in August 1994. This has created a community ofpeople interested in documenting software expertise.

Of course, others have had this goal as well. Donald Knuth's *TheArt of Computer Programming* [Knu73] was one of the firstattempts to catalog software knowledge, though he focused ondescribing algorithms. Even so, the task proved too great tofinish. The *Graphics Gems*series [Gla90, Arv91, Kir92] is anothercatalog of design knowledge, though it too tends to focus onalgorithms. The Domain Specific

Software Architecture programsponsored by the U.S. Department of Defense [GM92] concentrates on gathering architectural information. Theknowledge-based software engineering community tries to representsoftware-related knowledge in general. There are many other groupswith goals at least a little like ours.

James Coplien's *Advanced C++: Programming Styles andIdioms* [Cop92] has influenced us, too. The patterns inhis book tend to be more C++-specific than our design patterns, andhis book contains lots of lower-level patterns as well. But there issome overlap, as we point out in our patterns. Jim has been active inthe pattern community. He's currently working on patterns thatdescribe people's roles in software development organizations.

There are a lot of other places in which to find descriptionsof patterns. Kent Beck was one of the first people in the softwarecommunity to advocate Christopher Alexander's work. In 1993 hestarted writing a column in *The Smalltalk Report* onSmalltalk patterns. Peter Coad has also been collecting patternsfor some time. His paper on patterns seems to us to contain mostlyanalysis patterns [Coa92]; we haven't seen his latest patterns, though we know he is stillworking on them. We've heard of several books on patterns thatare in the works, but we haven't seen any of them, either. All wecan do is let you know they're coming. One of these books will befrom the Pattern Languages of Programs conference.

An Invitation

What can you do if you are interested in patterns? First, use themand look for other patterns that fit the way you design. A lot of books and articles about patterns will be coming outin the next few years,so there will be plenty of sources for new patterns. Develop yourvocabulary of patterns, and use it. Use it when you talk with otherpeople about your designs. Use it when you think and write about them.

Second, be a critical consumer. The design pattern catalog is theresult of hard work, not just ours but that of dozens of reviewers whogave us feedback. If you spot a problem or believe moreexplanation is needed, contact us. The same goes for any other catalog ofpatterns: Give the authors feedback! One of the great things aboutpatterns is that they move design decisions out of the realm of vagueintuition. They let authors be explicit about the trade-offs theymake. This makes it easier to see what is wrong with their patternsand to argue with them. Take advantage of that.

Third, look for patterns you use, and write them down. Make them apart of your documentation. Show them to other people. You don'thave to be in a research lab to find patterns. In fact, findingrelevant patterns is nearly impossible if you

don't have practicalexperience. Feel free to write your own catalog of patterns...butmake sure someone else helps you beat them into shape!

A Parting Thought

The best designs will use many design patterns that dovetail andintertwine to produce a greater whole. As Christopher Alexander says:

> It is possible to make buildings by stringing together patterns, in arather loose way. A building made like this, is an assembly ofpatterns. It is not dense. It is not profound. But it is also possibleto put patterns together in such a way that many patterns overlap inthe same physical space: the building is very dense; it has manymeanings captured in a small space; and through this density, itbecomes profound.
>
> *A Pattern Language* [AIX+77, page *xli*]

---

[1]See "The poetry of thelanguage" [AIS+77].

Appendix A: Glossary

abstract class

A class whose primary purpose is to define an interface. An abstract class defers some or all of its implementation to subclasses. An abstract class cannot be instantiated.

abstract coupling

Given a class *A* that maintains a reference to an abstract class *B*, class *A* is said to be *abstractly coupled* to *B*. We call this abstract coupling because *A* refers to a *type* of object, not a concrete object.

abstract operation

An operation that declares a signature but doesn't implement it. In C++, an abstract operation corresponds to a **pure virtual member function**.

acquaintance relationship

A class that refers to another class has an *acquaintance* with that class.

aggregate object

An object that's composed of subobjects. The subobjects are called the aggregate's **parts**, and the aggregate is responsible for them.

aggregation relationship

The relationship of an aggregate object to its parts. A class defines this relationship for its instances (e.g., aggregate objects).

black-box reuse

A style of reuse based on object composition. Composed objects reveal no internal details to each other and are thus analogous to "black boxes."

class

A class defines an object's interface and implementation. It specifies the object's internal representation and defines the operations the object can perform.

class diagram

A diagram that depicts classes, their internal structure and operations, and the static relationships between them.

class operation

An operation targeted to a class and not to an individual object. In C++, class operations are are called **static member functions**.

concrete class

A class having no abstract operations. It can be instantiated.

constructor

In C++, an operation that is automatically invoked to initialize new instances.

coupling

The degree to which software components depend on each other.

delegation

An implementation mechanism in which an object forwards or *delegates*

a request to another object. The delegate carries out the request on behalf of the original object.

design pattern

A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

destructor

In C++, an operation that is automatically invoked to finalize an object that is about to be deleted.

dynamic binding

The run-time association of a request to an object and one of its operations. In C++, only virtual functions are dynamically bound.

encapsulation

The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object. Operations are the only way to access and modify an object's representation.

framework

A set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance

by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.

friend class

In C++, a class that has the same access rights to the operations and data of a class as that class itself.

inheritance

A relationship that defines one entity in terms of another. **Class inheritance** defines a new class in terms of one or more parent classes. The new class inherits its interface and implementation from its parents. The new class is called a **subclass** or (in C++) a **derived class**. Class inheritance combines **interface inheritance** and **implementation inheritance**. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or more existing implementations.

instance variable

A piece of data that defines part of an object's representation. C++ uses the term **data member**.

interaction diagram

A diagram that shows the flow of requests between objects.

interface

The set of all signatures defined by an object's operations. The interface describes the set of requests to which an object can respond.

metaclass

Classes are objects in Smalltalk. A metaclass is the class of a class object.

mixin class

A class designed to be combined with other classes through inheritance. Mixin classes are usually abstract.

object

A run-time entity that packages both data and the procedures that operate on that data.

object composition

Assembling or *composing* objects to get more complex behavior.

object diagram

A diagram that depicts a particular object structure at run-time.

object reference

A value that identifies another object.

operation

An object's data can be manipulated only by its operations. An object performs an operation when it receives a request. In C++, operations are called **member functions**. Smalltalk uses the term **method**.

overriding

Redefining an operation (inherited from a parent class) in a subclass.

parameterized type

A type that leaves some constituent types unspecified. The unspecified types are supplied as parameters at the point of use. In C++, parameterized types are called **templates**.

parent class

The class from which another class inherits. Synonyms are **superclass** (Smalltalk), **base class** (C++), and **ancestor class**.

polymorphism

The ability to substitute objects of matching interface for one another at run-time.

private inheritance

In C++, a class inherited solely for its implementation.

protocol

Extends the concept of an interface to include the allowable sequences of requests.

receiver

The target object of a request.

request

An object performs an operation when it receives a corresponding request from another object. A common synonym for request is **message**.

signature

An operation's signature defines its name, parameters, and return value.

subclass

A class that inherits from another class. In C++, a subclass is called a **derived class**.

subsystem

An independent group of classes that collaborate to fulfill a set of responsibilities.

subtype

A type is a subtype of another if its interface contains the interface of the other type.

supertype

The parent type from which a type inherits.

toolkit

A collection of classes that provides useful functionality but does not define the design of an application.

type

The name of a particular interface.

white-box reuse

A style of reuse based on class inheritance. A subclass reuses the interface and implementation of its parent class, but it may have access to otherwise private aspects of its parent.

Appendix B: Guide to Notation

We use diagrams throughout the book to illustrate important ideas.Some diagrams are informal, like a screen shot of a dialog box or aschematic showing a tree of objects. But the design patterns inparticular use more formal notations to denote relationships andinteractions between classes and objects. This appendix describesthese notations in detail.

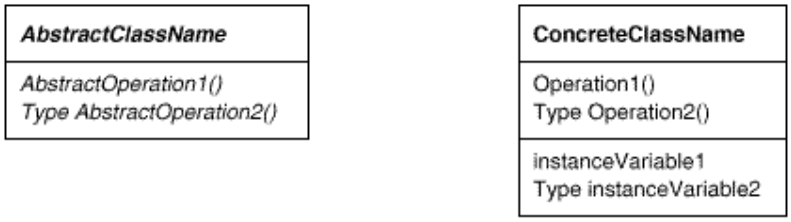
We use three different diagrammatic notations:

1. A **class diagram** depicts classes, their structure, andthe static relationships between them.
2. An **object diagram** depicts a particular object structureat run-time.
3. An **interaction diagram** shows the flow of requests betweenobjects.

Each design pattern includes at least one class diagram. Theother notations are used as needed to supplement the discussion.The class and object diagrams are based on OMT (Object ModelingTechnique) [RBP+91, Rum94].[1]The interaction diagramsare taken from Objectory [JCJO92] and the Booch method [Boo94]. These notations are summarizedon the inside back cover of the book.

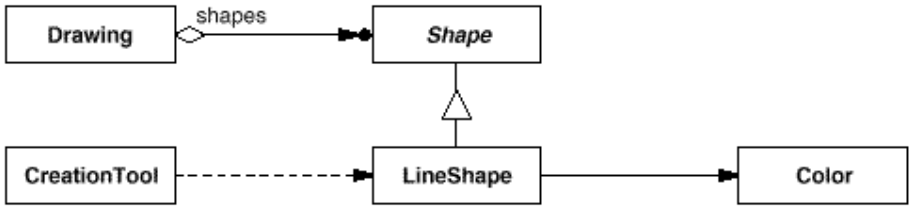
Class Diagram

Figure B.1a shows the OMT notation for abstract and concrete classes. A classis denoted by a box with the class name in bold type at the top.The key operations of the class appear below the class name. Anyinstance variables appear below the operations. Type informationis optional; we use the C++ convention, which puts the type namebefore the name of the operation (to signify the return type),instance variable, or actual parameter. Slanted type indicatesthat the class or operation is abstract.

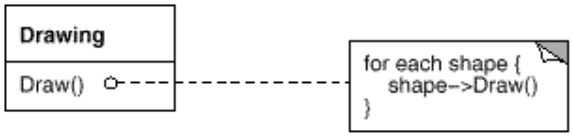


(a) Abstract and concrete classes

(b) Participant Client class (left) and implicit Client class (right)



(c) Class relationships



(d) Pseudocode annotation

Figure B.1: Class diagram notation

In some design patterns it's helpful to see where client classes reference Participant classes. When a pattern includes a Clientclass as one of its participants (meaning the client has aresponsibility in the pattern), the Client appears as an ordinaryclass. This is true in Flyweight (218), for example. When the pattern does not include a Client participant(i.e., clients have no responsibilities in the pattern), butincluding it nevertheless clarifies which

pattern participantsinteract with clients, then the Client class is shown in gray, asshown in Figure B.1b. An example is Proxy (233). A gray Clientalso makes it clear that we haven't accidentally omitted the Clientfrom the Participants discussion.

Figure B.1c shows variousrelationships between classes. The OMT notation for class inheritanceis a triangle connecting a subclass (LineShape in the figure) to itsparent class (Shape). An object reference representing a part-of oraggregation relationship is indicated by an arrowheaded line with adiamond at the base. The arrow points to the class that is aggregated(e.g., Shape). An arrowheaded line without the diamond denotesacquaintance (e.g., a LineShape keeps a reference to a Color object,which other shapes may share). A name for the reference may appearnear the base to distinguish it from otherreferences.[2]

Another useful thing to show is which classes instantiate whichothers. We use a dashed arrowheaded line to indicate this, sinceOMT doesn't support it. We call this the "creates" relationship.The arrow points to the class that's instantiated. In Figure B.1c, CreationTool creates LineShape objects.

OMT also defines a filled circle to mean "more than one." Whenthe circle appears at the head of a reference, it means multipleobjects are being referenced or aggregated. Figure B.1c shows that Drawing aggregatesmultiple objects of type Shape.

Finally, we've augmented OMT with pseudocode annotations to letus sketch the implementations of operations. Figure B.1d shows the pseudocode annotationfor the Draw operation on the Drawing class.

Object Diagram

An object diagram shows instances exclusively. It provides asnapshot of the objects in a design pattern. The objects are named"a*Something*", where *Something* is the class ofthe object. Our symbol for an object (modified slightly fromstandard OMT) is a rounded box with a line separating the objectname from any object references. Arrows indicate the objectreferenced. Figure B.2 shows an example.

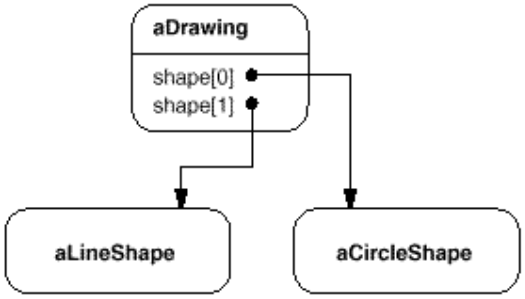


Figure B.2: Object diagram notation

Interaction Diagram

An interaction diagram shows the order in which requests between objectsget executed. Figure B.3 is aninteraction diagram that shows how a shape gets added to a drawing.

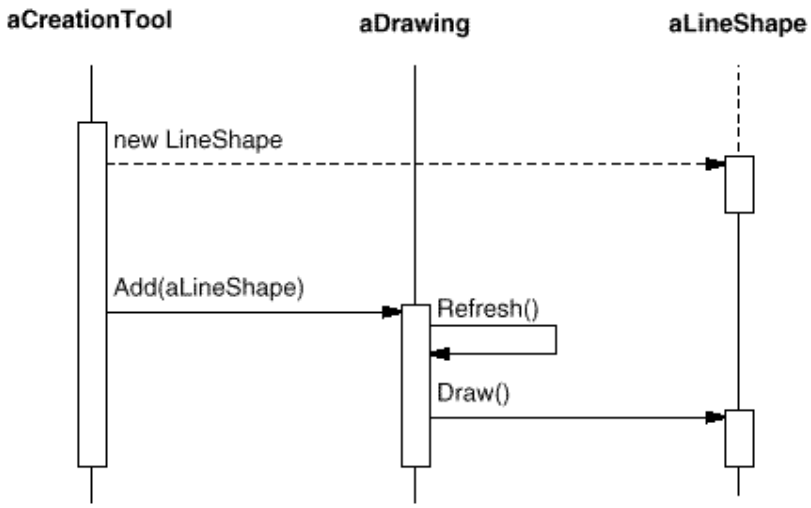


Figure B.3: Interaction diagram notation

Time flows from top to bottom in an interaction diagram. A solidvertical line indicates the lifetime of a particular object. Thenaming convention for objects is the same as for object diagrams—theclass name prefixed by the letter "a" (e.g., aShape). If the objectdoesn't get instantiated until after the beginning of time as recordedin the diagram, then its vertical line appears dashed until the pointof creation.

A vertical rectangle shows that an object is active; that is, it ishandling a request. The operation can send requests to other objects;these are indicated with a horizontal arrow pointing to the receivingobject. The name of the request is shown above the arrow. A requestto create an object is shown with a dashed arrowheaded line. Arequest to the sending object itself points back to the sender.

Figure B.3 shows that the first request is from aCreationTool tocreate aLineShape. Later, aLineShape is Added to aDrawing, whichprompts aDrawing to send a Refresh request to itself. Note thataDrawing sends a Draw request to aLineShape as part of the Refreshoperation.

---

[1]OMT uses the term "object diagram" torefer to class diagrams. We use "object diagram" exclusively torefer to diagrams of object structures.

[2]OMT also defines **associations**between classes, which appear as plain lines between class boxes.Associations are bidirectional. Although associations are appropriateduring analysis, we feel they're too high-level for expressing therelationships in design patterns, simply because associations mustbe mapped down to object references or pointers during design.Object references are intrinsically directed and are thereforebetter suited to the relationships that concern us. For example,Drawing knows about Shapes, but the Shapes don't know about theDrawing they're in. You can't express this relationship withassociations alone.

Appendix C: Foundation Classes

This appendix documents the foundation classes we use in the C++sample code of several design patterns. We've intentionally kept the classes simple and minimal. We describe the following classes:

- List, an ordered list of objects.
- Iterator, the interface for accessing an aggregate's objects in a sequence.
- ListIterator, an iterator for traversing a List.
- Point, a two-dimensional point.
- Rect, an axis-aligned rectangle.

Some newer C++ standard types may not be available on allcompilers. In particular, if your compiler doesn't definebool, then define it manually as

```
typedef int bool;
const int true = 1;
const int false = 0;
```

List

The List class template provides a basic container forstoring an ordered list of objects. List stores elements byvalue, which means it works for built-in types as well as classinstances. For example, List declares a list ofints. But most of the patterns use List tostore pointers to objects, as in List. That wayList can be used for heterogeneous lists.

For convenience, List also provides synonyms for stackoperations, which make code that uses List for stacks moreexplicit without defining another class.

```
template <class Item>
class List {
public:
        List(long size = DEFAULT_LIST_CAPACITY);
        List(List&);
        ~List();
        List& operator=(const List&);

        long Count() const;
        Item& Get(long index) const;
        Item& First() const;
        Item& Last() const;
        bool Includes(const Item&) const;
```

```
        void Append(const Item&);
        void Prepend(const Item&);

        void Remove(const Item&);
        void RemoveLast();
        void RemoveFirst();
        void RemoveAll();

        Item& Top() const;
        void Push(const Item&);
        Item& Pop();
};
```

The following sections describe these operations in greater detail.

Construction, Destruction, Initialization, and Assignment

`List(long size)`

initializes the list. The size parameter is a hint forthe initial number of elements.

`List(List&)`

overrides the default copy constructor so that member data areinitialized properly.

`~List()`

frees the list's internal data structures but *not* theelements in the list. The class is not designed for subclassing;therefore the destructor isn't virtual.

`List& operator=(const List&)`

implements the assignment operation to assign member data properly.

Accessing

These operations provide basic access to the list's elements.

`long Count() const`

returns the number of objects in the list.

`Item& Get(long index) const`

returns the object at the given index.

`Item& First() const`

returns the first object in the list.

`Item& Last() const`

returns the last object in the list.

Adding

`void Append(const Item&)`

adds the argument to the list, making it the last element.

`void Prepend(const Item&)`

adds the argument to the list, making it the first element.

Removing

`void Remove(const Item&)`

removes the given element from the list. This operation requiresthat the type of elements in the list supports the== operator for comparison.

`void RemoveFirst()`

removes the first element from the list.

`void RemoveLast()`

removes the last element from the list.

`void RemoveAll()`

removes all elements from the list.

Stack Interface

Item& Top() const

returns the top element (when the List is viewed as a stack).

void Push(const Item&)

pushes the element onto the stack.

Item& Pop()

pops the top element from the stack.

Iterator

Iterator is an abstract class that defines a traversalinterface for aggregates.

```
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();
};
```

The operations do the following:

virtual void First()

positions the iterator to the first object in the aggregate.

virtual void Next()

positions the iterator to the next object in the sequence.

virtual bool IsDone() const

returns true when there are no more objects in the sequence.

virtual Item CurrentItem() const

returns the object at the current position in the sequence.

ListIterator

ListIterator implements the Iterator interfaceto traverse List objects. Its constructor takes a list to traverse asan argument.

```
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;
};
```

Point

Point represents a point in a two-dimensional Cartesiancoordinate space. Point supports some minimal vector arithmetic.The coordinates of a Point are defined as

```
   typedef float Coord;
```

Point's operations are self-explanatory.

```
class Point {
public:
        static const Point Zero;

        Point(Coord x = 0.0, Coord y = 0.0);

        Coord X() const;  void X(Coord x);
        Coord Y() const;  void Y(Coord y);

        friend Point operator+(const Point&, const Point&);
        friend Point operator-(const Point&, const Point&);
        friend Point operator*(const Point&, const Point&);
        friend Point operator/(const Point&, const Point&);

        Point& operator+=(const Point&);
```

```
        Point& operator-=(const Point&);
        Point& operator*=(const Point&);
        Point& operator/=(const Point&);

        Point operator-();

        friend bool operator==(const Point&, const Point&);
        friend bool operator!=(const Point&, const Point&);

        friend ostream& operator<<(ostream&, const Point&);
        friend istream& operator>>(istream&, Point&);
};
```

The static member Zero represents Point(0, 0).

Rect

Rect represents an axis-aligned rectangle. ARect is defined by an origin point and an extent (thatis, width and height). The Rect operations areself-explanatory.

```
class Rect {
public:
        static const Rect Zero;

        Rect(Coord x, Coord y, Coord w, Coord h);
        Rect(const Point& origin, const Point& extent);

        Coord Width() const;   void Width(Coord);
        Coord Height() const;  void Height(Coord);
        Coord Left() const;    void Left(Coord);
        Coord Bottom() const;  void Bottom(Coord);

        Point& Origin() const; void Origin(const Point&);
        Point& Extent() const; void Extent(const Point&);

        void MoveTo(const Point&);
        void MoveBy(const Point&);

        bool IsEmpty() const;
        bool Contains(const Point&) const;
};
```

The static member Zero is equivalent to the rectangle

```
Rect(Point(0, 0), Point(0, 0));
```

Bibliography

[Add94]

Addison-Wesley, Reading, MA. *NEXTSTEPGeneral Reference: Release 3, Volumes 1 and 2*, 1994.

[AG90]

D.B. Anderson and S. Gossain. Hierarchyevolution and the software lifecycle. In *TOOLS '90 ConferenceProceedings*, pages 41–50, Paris, June 1990. Prentice Hall.

[AIS+77]

Christopher Alexander, Sara Ishikawa, MurraySilverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel.*A Pattern Language*. Oxford University Press, NewYork, 1977.

[App89]

Apple Computer, Inc., Cupertino, CA. *MacintoshProgrammers Workshop Pascal 3.0 Reference*, 1989.

[App92]

Apple Computer, Inc., Cupertino, CA. *Dylan.An object-oriented dynamic language*, 1992.

[Arv91]

James Arvo. *Graphics Gems II*.Academic Press, Boston, MA, 1991.

[AS85]

B. Adelson and E. Soloway. The role ofdomain experience in software design. *IEEE Transactions onSoftware Engineering*, 11(11):1351–1360, 1985.

[BE93]

Andreas Birrer and Thomas Eggenschwiler. Frameworksin the financial engineering domain: An experience report. In*European Conference on Object-Oriented Programming*,pages 21–35, Kaiserslautern, Germany, July 1993.Springer-Verlag.

[BJ94]

Kent Beck and Ralph Johnson. Patterns generatearchitectures. In *European Conference on Object-OrientedProgramming*, pages 139–149, Bologna, Italy, July 1994. Springer-Verlag.

[Boo94]

Grady Booch. *Object-Oriented Analysis andDesign with Applications*. Benjamin/Cummings, Redwood City,CA, 1994. Second Edition.

[Bor81]

A. Borning. The programming language aspectsof ThingLab—a constraint-oriented simulation laboratory.*ACM Transactions on Programming Languages and Systems*,3(4):343–387, October 1981.

[Bor94]

Borland International, Inc., Scotts Valley, CA.*A Technical Comparison of Borland ObjectWindows 2.0 andMicrosoft MFC 2.5*, 1994.

[BV90]

Grady Booch and Michael Vilot. The design of theC++ Booch components. In

Object-Oriented Programming Systems,Languages, and Applications Conference Proceedings, pages1–11, Ottawa, Canada, October 1990. ACM Press.

[Cal93]

Paul R. Calder. *Building User Interfaceswith Lightweight Objects*. PhD thesis, Stanford University,1993.

[Car89]

J. Carolan. Constructing bullet-proof classes.In *Proceedings C++ at Work '89*. SIGS Publications,1989.

[Car92]

Tom Cargill. *C++ Programming Style*.Addison-Wesley, Reading, MA, 1992.

[CIRM93]

Roy H. Campbell, Nayeem Islam, David Raila,and Peter Madeany. Designing and implementing Choices: Anobject-oriented system in C++. *Communications of theACM*, 36(9):117–126, September 1993.

[CL90]

Paul R. Calder and Mark A. Linton.Glyphs: Flyweight objects for user interfaces. In *ACM UserInterface Software Technologies Conference*, pages 92–101, Snowbird, UT, October 1990.

[CL92]

Paul R. Calder and Mark A. Linton. Theobject-oriented implementation of a document editor. In*Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 154–165, Vancouver, British Columbia, Canada, October 1992. ACM Press.

[Coa92]

Peter Coad. Object-oriented patterns.*Communications of the ACM*, 35(9):152–159, September1992.

[Coo92]

William R. Cook. Interfaces and specificationsfor the Smalltalk-80 collection classes. In *Object-Oriented Programming Systems, Languages,and Applications Conference Proceedings*, pages 1–15, Vancouver, British Columbia, Canada, October 1992. ACM Press.

[Cop92]

James O. Coplien. *Advanced C++Programming Styles and Idioms*. Addison-Wesley, Reading,MA, 1992.

[Cur89]

Bill Curtis. Cognitive issues in reusing softwareartifacts. In Ted J. Biggerstaff and Alan J. Perlis,editors, *Software Reusability, Volume II: Applications and Experience*, pages 269–287. Addison-Wesley, Reading, MA, 1989.

[dCLF93]

Dennis de Champeaux, Doug Lea, and PenelopeFaure. *Object-Oriented System Development*.Addison-Wesley, Reading, MA, 1993.

[Deu89]

L. Peter Deutsch. Design reuse and frameworksin the Smalltalk-80 system. In Ted J. Biggerstaff andAlan J. Perlis, editors, *Software Reusability, Volume II: Applications and Experience*, pages 57–71. Addison-Wesley, Reading,

MA, 1989.

[Ede92]

D. R. Edelson. Smart pointers: They're smart,but they're not pointers. In *Proceedings of the 1992 USENIXC++ Conference*, pages 1–19, Portland, OR, August 1992. USENIX Association.

[EG92]

Thomas Eggenschwiler and Erich Gamma. TheET++SwapsManager: Using object technology in the financial engineering domain. In *Object-Oriented Programming Systems,Languages, and Applications Conference Proceedings*, pages 166–178, Vancouver, British Columbia, Canada, October 1992. ACM Press.

[ES90]

Margaret A. Ellis and Bjarne Stroustrup.*The Annotated C++ Reference Manual*. Addison-Wesley,Reading, MA, 1990.

[Foo92]

Brian Foote. A fractal model of the lifecyclesof reusable objects. *OOPSLA '92 Workshop on Reuse*,October 1992. Vancouver, British Columbia, Canada.

[GA89]

S. Gossain and D.B. Anderson. Designing aclass hierarchy for domain representation and reusability. In *TOOLS '89 Conference Proceedings*,pages 201–210, CNIT Paris—La Defense, France, November 1989. Prentice Hall.

[Gam91]

Erich Gamma. *Object-Oriented SoftwareDevelopment based on ET++: Design Patterns, Class Library, Tools* (in German). PhD thesis,University of Zurich

Institut für Informatik, 1991.

[Gam92]

Erich Gamma. *Object-Oriented SoftwareDevelopment based on ET++: Design Patterns, Class Library, Tools* (in German). Springer-Verlag,Berlin, 1992.

[Gla90]

Andrew Glassner. *Graphics Gems*.Academic Press, Boston, MA, 1990.

[GM92]

M. Graham and E. Mettala. TheDomain-Specific Software Architecture Program. In *Proceedingsof DARPA Software Technology Conference, 1992*, pages 204–210, April 1992. Also published in *CrossTalk,The Journal of Defense Software Engineering*, pages 19–21, 32, October 1992.

[GR83]

Adele J. Goldberg and David Robson.*Smalltalk-80: The Language and Its Implementation*.Addison-Wesley, Reading, MA, 1983.

[HHMV92]

Richard Helm, Tien Huynh, Kim Marriott, and JohnVlissides. An object-oriented architecture for constraint-basedgraphical editing. In *Proceedings of the Third Eurographics Workshopon Object-Oriented Graphics*, pages 1–22, Champéry, Switzerland, October 1992. Also available as IBM Research DivisionTechnical Report RC 18524 (79392).

[HO87]

Daniel C. Halbert and Patrick D. O'Brien.Object-oriented development. *IEEE Software*,4(5):71–79, September 1987.

[ION94]

IONA Technologies, Ltd., Dublin, Ireland.*Programmer's Guide for Orbix, Version 1.2*, 1994.

[JCJO92]

Ivar Jacobson, Magnus Christerson, Patrik Jonsson,and Gunnar Overgaard. *Object-Oriented Software Engineering—AUse Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.

[JF88]

Ralph E. Johnson and Brian Foote. Designingreusable classes. *Journal of Object-Oriented Programming*,1(2):22–35, June/July 1988.

[JML92]

Ralph E. Johnson, Carl McConnell, andJ. Michael Lake. The RTL system: A framework for codeoptimization. In Robert Giegerich and Susan L. Graham, editors,*Code Generation—Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, pages 255–274, Dagstuhl, Germany, 1992. Springer-Verlag.

[Joh92]

Ralph Johnson. Documenting frameworks usingpatterns. In *Object-Oriented Programming Systems, Languages,and Applications Conference Proceedings*, pages 63–76, Vancouver, British Columbia, Canada, October 1992. ACM Press.

[JZ91]

Ralph E. Johnson and Jonathan Zweig. Delegationin C++. *Journal of Object-Oriented Programming*,4(11):22–35, November 1991.

[Kir92]

David Kirk. *Graphics Gems III*.Harcourt, Brace, Jovanovich, Boston, MA, 1992.

[Knu73]

Donald E. Knuth. *The Art of ComputerProgramming, Volumes 1, 2, and 3*. Addison-Wesley, Reading,MA, 1973.

[Knu84]

Donald E. Knuth. *The TeX book*.Addison-Wesley, Reading, MA, 1984.

[Kof93]

Thomas Kofler. Robust iterators in ET++.*Structured Programming*, 14:62–85, March1993.

[KP88]

Glenn E. Krasner and Stephen T. Pope.A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-OrientedProgramming*, 1(3):26–49, August/September 1988.

[LaL94]

Wilf LaLonde. *Discovering Smalltalk*.Benjamin/Cummings, Redwood City, CA, 1994.

[LCI+92]

Mark Linton, Paul Calder, John Interrante, StevenTang, and John Vlissides. *InterViews Reference Manual*.CSL, Stanford University, 3.1 edition, 1992.

[Lea88]

Doug Lea. libg++, the GNU C++ library. In *Proceedings of the 1988 USENIX C++ Conference*, pages 243–256, Denver, CO, October 1988. USENIX Association.

[LG86]

Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. McGraw-Hill, NewYork, 1986.

[Lie85]

Henry Lieberman. There's more to menu systems than meets the screen. In *SIGGRAPH Computer Graphics*, pages 181–189, San Francisco, CA, July 1985.

[Lie86]

Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 214–223, Portland, OR, November 1986.

[Lin92]

Mark A. Linton. Encapsulating a C++ library. In *Proceedings of the 1992 USENIX C++ Conference*, pages 57–66, Portland, OR, August 1992. ACM Press.

[LP93]

Mark Linton and Chuck Price. Building distributed user interfaces with Fresco. In *Proceedings of the 7th X Technical Conference*, pages 77–87, Boston, MA, January 1993.

[LR93]

Daniel C. Lynch and Marshall T. Rose. *Internet System Handbook*.

Addison-Wesley, Reading,MA, 1993.

[LVC89]

Mark A. Linton, John M. Vlissides, andPaul R. Calder. Composing user interfaces with InterViews.*Computer*, 22(2):8–22, February 1989.

[Mar91]

Bruce Martin. The separation of interface andimplementation in C++. In *Proceedings of the 1991 USENIXC++ Conference*, pages 51–63, Washington, D.C., April 1991. USENIX Association.

[McC87]

Paul McCullough. Transparent forwarding: Firststeps. In *Object-Oriented Programming Systems, Languages,and Applications Conference Proceedings*, pages 331–341, Orlando, FL, October 1987. ACM Press.

[Mey88]

Bertrand Meyer. *Object-Oriented SoftwareConstruction*. Series in Computer Science. Prentice Hall,Englewood Cliffs, NJ, 1988.

[Mur93]

Robert B. Murray. *C++ Strategies andTactics*. Addison-Wesley, Reading, MA, 1993.

[OJ90]

William F. Opdyke and Ralph E. Johnson.Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *SOOPPA Conference Proceedings*,pages 145–161, Marist College, Poughkeepsie, NY, September 1990. ACM Press.

[OJ93]

William F. Opdyke and Ralph E. Johnson.Creating abstract superclasses by refactoring. In *Proceedingsof the 21st Annual Computer Science Conference (ACM CSC '93)*, pages 66–73, Indianapolis, IN, February 1993.

[P+88]

Andrew J. Palay et al. The AndrewToolkit: An overview. In *Proceedings of the 1988 WinterUSENIX Technical Conference*, pages 9–21, Dallas, TX, February 1988. USENIX Association.

[Par90]

ParcPlace Systems, Mountain View, CA.*ObjectWorks\Smalltalk Release 4 Users Guide*,1990.

[Pas86]

Geoffrey A. Pascoe. Encapsulators: A newsoftware paradigm in Smalltalk-80. In *Object-OrientedProgramming Systems, Languages, and Applications Conference Proceedings*, pages 341–346, Portland, OR, October 1986. ACM Press.

[Pug90]

William Pugh. Skiplists: A probabilistic alternativeto balanced trees. *Communications of the ACM*,33(6):668–676, June 1990.

[RBP+91]

James Rumbaugh, Michael Blaha, William Premerlani,Frederick Eddy, and William Lorenson. *Object-Oriented Modelingand Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rum94]

James Rumbaugh. The life of an object model: Howthe object model changes during development. *Journal ofObject-Oriented Programming*, 7(1):24–32, March/April1994.

[SE84]

Elliot Soloway and Kate Ehrlich. Empirical studiesof programming knowledge. *IEEE Transactions on SoftwareEngineering*, 10(5):595–609, September 1984.

[Sha90]

Yen-Ping Shan. MoDE: A UIMS for Smalltalk. In*ACM OOPSLA/ECOOP '90 Conference Proceedings*, pages258–268, Ottawa, Ontario, Canada, October 1990. ACM Press.

[Sny86]

Alan Snyder. Encapsulation and inheritance inobject-oriented languages. In *Object-Oriented ProgrammingSystems, Languages, and Applications Conference Proceedings*,pages 38–45, Portland, OR, November 1986. ACM Press.

[SS86]

James C. Spohrer and Elliot Soloway. Novicemistakes: Are the folk wisdoms correct? *Communications ofthe ACM*, 29(7):624–632, July 1986.

[SS94]

Douglas C. Schmidt and Tatsuya Suda. TheService Configurator Framework: An extensible architecture fordynamically configuring concurrent, multi-service network daemons.In *Proceeding of the Second International*

Workshop onConfigurable Distributed Systems, pages 190–201,Pittsburgh, PA, March 1994. IEEE Computer Society.

[Str91]

Bjarne Stroustrup. *The C++ ProgrammingLanguage*. Addison-Wesley, Reading, MA, 1991. SecondEdition.

[Str93]

Paul S. Strauss. IRIS Inventor, a 3D graphicstoolkit. In *Object-Oriented Programming Systems, Languages,and Applications Conference Proceedings*, pages 192–200,Washington, D.C., September 1993. ACM Press.

[Str94]

Bjarne Stroustrup. *The Design and Evolutionof C++*. Addison-Wesley, Reading, MA, 1994.

[Sut63]

I.E. Sutherland. *Sketchpad: A Man-MachineGraphical Communication System*. PhD thesis, MIT, 1963.

[Swe85]

Richard E. Sweet. The Mesa programmingenvironment. *SIGPLAN Notices*, 20(7):216–229,July 1985.

[Sym93a]

Symantec Corporation, Cupertino, CA. *BedrockDeveloper's Architecture Kit*, 1993.

[Sym93b]

Symantec Corporation, Cupertino, CA. *THINKClass Library Guide*, 1993.

[Sza92]

Duane Szafron. SPECTalk: An object-oriented dataspecification language. In *Technology of Object-OrientedLanguages and Systems (TOOLS 8)*, pages 123–138, SantaBarbara, CA, August 1992. Prentice Hall.

[US87]

David Ungar and Randall B. Smith. Self: Thepower of simplicity. In *Object-Oriented Programming Systems,Languages, and Applications Conference Proceedings*, pages227–242, Orlando, FL, October 1987. ACM Press.

[VL88]

John M. Vlissides and Mark A. Linton.Applying object-oriented design to structured graphics. In*Proceedings of the 1988 USENIX C++ Conference*, pages81–94, Denver, CO, October 1988. USENIX Association.

[VL90]

John M. Vlissides and Mark A. Linton.Unidraw: A framework for building domain-specific graphical editors.*ACM Transactions on Information Systems*, 8(3):237–268,July 1990.

[WBJ90]

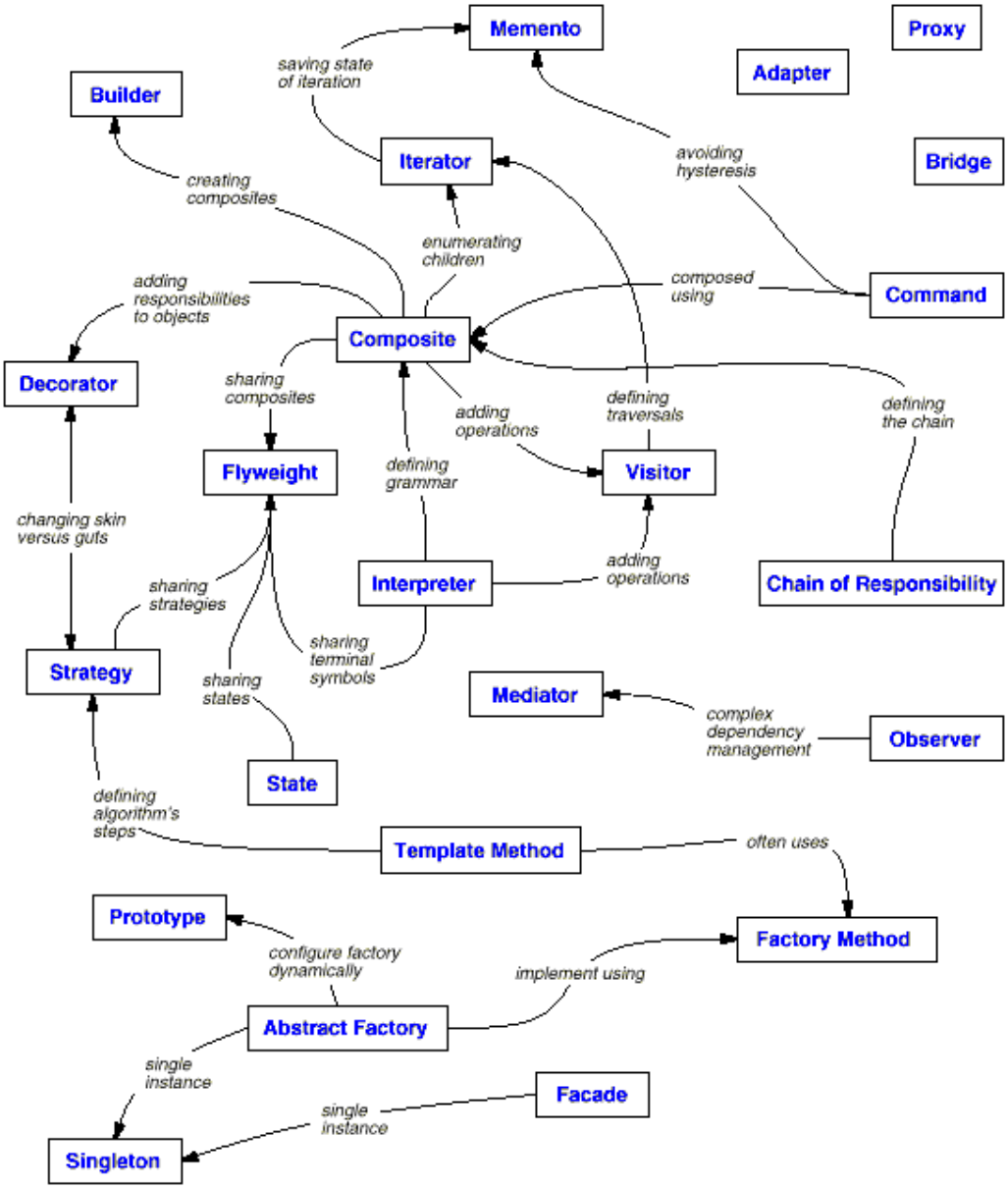
Rebecca Wirfs-Brock and Ralph E. Johnson.A survey of current research in object-oriented design.*Communications of the ACM*, 33(9):104–124,1990.

[WBWW90]

Rebecca Wirfs-Brock, Brian Wilkerson, and LaurenWiener. *Designing Object-Oriented Software*. PrenticeHall, Englewood Cliffs, NJ, 1990.

[WGM88]

André Weinand, Erich Gamma, and Rudolf Marty.ET++—An object-oriented application framework in C++. In*Object-Oriented Programming Systems, Languages, and ApplicationsConference Proceedings*, pages 46–57, San Diego, CA,September 1988. ACM Press.



Design pattern relationships