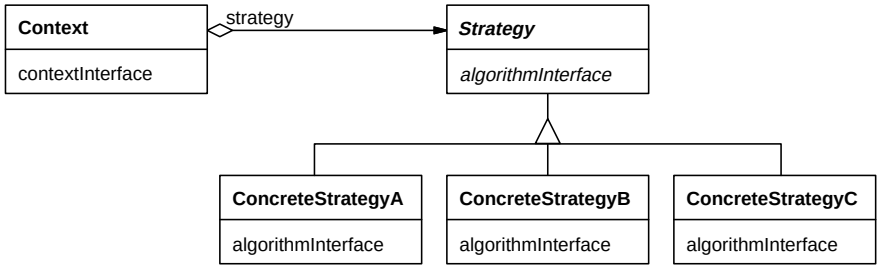
STRATEGY (DP 315) Object Behavioral

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Structure



Discussion

The Strategy pattern is used when we wish to encapsulate an algorithm in a class, or more precisely, when we wish to encapsulate multiple variations of a particular service, system function, or algorithm in multiple separate classes. When would we want to do this? Well, many systems must choose among multiple strategies based on the context or situation, and many interactive applications allow users to select one of several possible different algorithmic strategies. For example, storage/compression programs, which allow users to save files in compressed formats, often allow users to select which compression standard to use. At times, there are space-vs.-speed considerations in making this selection: one algorithm operates faster than another but has greater memory requirements. At times the choice involves a space-vs.-quality trade-off; this is seen, for example, when compressing video data: some algorithms retain high quality video integrity but result in large data files which require high bandwidth playback capabilities, whereas other algorithms may be somewhat lossy yet result in smaller file sizes and simpler playback requirements. Thus, compression programs must allow users to select the algorithm which best matches their needs. The question at hand is, How do we implement, and programmatically choose among, the various algorithms, each of which may be quite complex, within an application.

Let's answer this by considering another example loosely fashioned after the *Design Patterns* example on page DP 315. Here, a document editor operates on a composition object containing text and graphics; the composition is responsible for formatting itself for output and can choose among several layout strategies based on user preference. One approach to implementing several strategy choices in such situations would be to "bloat"

the implementation of the responsible class itself, by implementing the various algorithm choices as individual methods therein; in the composition example, this would mean incorporating the various layout algorithms as distinct methods in the composition class definition. Then, each time this functionality is required, the composition would invoke the appropriate method via a set of conditional statements. Of course, with regard to extensibility, this implies revisiting this conditional if we add a new strategy in the future.

Instead, we can invoke the Strategy pattern to provide a more modular and extensible solution. Here, we'll (1) implement each formatting strategy/algorithm in its own—separate—class, and (2) have the composition point to an instance of one of these Strategy classes and call upon this separate "formatter" object to perform the layout function. Thus, the formatting function is actually performed by an external helper object. The *choice* of formatting algorithm is effected simply by instantiating one or another of the layout/Strategy classes. This implies the strategy may be changed on the fly by a user (e.g., by menu selection), or programatically based on some changing condition, at runtime. Implementation-wise, doing so entails simply creating a new Strategy object and pointing to it. Now, each time the layout has to be recomputed, the client (the composition) invokes sends a single message to its formatter, as opposed to repetitively executing a conditional to decide which message to send itself.

As new formatting requirements arise later in the life of our application, it will be an easier task to implement and integrate such new algorithms since they have been separated into distinct classes. The composition implementation is also easier to understand and maintain because it does not include all of the various formatting algorithms. Instead, in the best tradition of object-oriented design, it merely communicates with *some* line-breaking object using an abstract interface and it really doesn't care what *sort* of line-breaking it is, as long as it adheres to this interface.

Let's take a quick look at the difference in the code required to invoke the line-breaking functionality (a) if we code the various line-breaking algorithms as separate methods within the `Composition` class, and (b) when using the Strategy pattern (this corresponds to the C++ code examples on DP 318, with some modification). Suppose the `Composition` class implements a `repair` method which is invoked to update the entire layout of the document, and which, in turn, calls the layout code.

Without the Strategy pattern

(Of course, there would be a number of ways to implement this; we'll look at one simple approach.) Here, `Composition` contains an instance variable named `formatting-Strategy` which determines the algorithm/method to invoke:

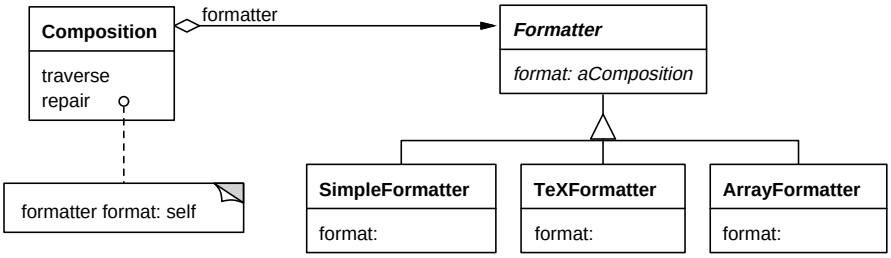
```
Composition>>repair
   "Without the strategy pattern."
   formattingStrategy == #SimpleStrategy
      ifTrue: [self formatWithSimpleAlgorithm]
      ifFalse: [formattingStrategy == #TeXStrategy
         ifTrue: [self formatWithTexAlgorithm]
         ifFalse: [...]
```

Using the Strategy pattern

Here, `Composition` contains an instance variable named `formatter` which points to its layout object. The `Composition` couldn't care less about the exact class of this helper object, merely that it responds to the `format:` message:

```
Composition>>repair
   "With the Strategy pattern."
   formatter format: self.
```

Here's the structure of this application after applying the Strategy pattern.



Sample Code[1]

Let's look more in depth at another example. Suppose we have a financial analysis application. It is capable of interacting with the user, on the one hand, and a database, on the other, to allow the visualization of various financial data. For instance, a user could request a report on the gross income of her company for the past four quarters. Once this information has been retrieved from the database, the application is capable of visualizing it in various formats: as a bar chart, a line graph, or a pie chart, among others. The user may select a display format before requesting specific data (e.g., set the application to show subsequent data as bar charts) and may also view the same set of data in various

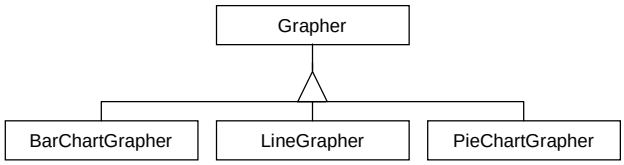
[1] In this pattern, we've switched the order of the Sample Code and Implementation sections: first, we'll present the code for an example application, and in the later section we'll discuss the implementation issues raised by this example.

formats (e.g., once a report, such as the past-four-quarters report, has been selected and formatted as a bar chart, the user may ask to see the same data as a line plot).

So, we start by implementing a `FinancialAnalyst` class. For purposes of our example, we'll do this the simplest way: since the application is interactive, we'll make `FinancialAnalyst` a subclass of `ViewManager` (our example will be portrayed with Visual Smalltalk code). The Financial Analyst window contains a `GraphPane` upon which the financial data will be graphed—this will be referenced by the `graphPane` variable, set in the `open` method in which the window is constructed. `FinancialAnalyst` also contains an instance variable to hold the data retrieved from the database prior to invoking the graph-drawing functionality. A third instance variable will point to the Analyst's `Grapher` object to which we delegate the actual task of drawing the graphic visualization on `graphPane`. We'll explain the `graphMessage` variable later. So far, we have the following class definition and methods:

```
ViewManager subclass: #FinancialAnalyst
  instanceVariableNames: 'graphPane data grapher
graphMessage'
  classVariableNames: ''
  poolDictionaries: ''

FinancialAnalyst>>open
   "Create the subpanes for, and open, the Analyst window."
   self
      label: 'Financial Analyst';
      addSubpane:
         ((graphPane := GraphPane new)
            owner: self;
            ...).
   "Setup a default grapher:"
   self grapher: (LineGrapher new pen: graphPane pen).
   self openWindow

FinancialAnalyst>>grapher: aGrapher
   grapher := aGrapher
```

Now, rather than implementing a set of graphing methods within the `FinancialAnalyst` class, we define a new subhierarchy of `Grapher` classes, as follows:



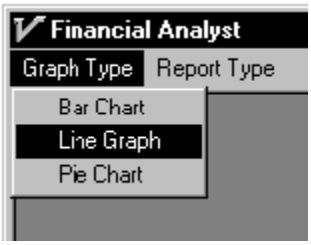
`Grapher` is an abstract superclass with (currently) three concrete subclasses. When the user requests financial information to be portrayed as pie charts, `FinancialAnalyst` instantiates `PieChartGrapher` and installs that instance as its `grapher` object. Similarly for line graphs and bar charts. When the graph must be drawn, `FinancialAnalyst` sends a message to its `Grapher` object to perform this task.

Let's start to define the abstract superclass of the `Grapher` hierarchy:

```
Object subclass: #Grapher
   instanceVariableNames: 'pen'
   classVariableNames: ''
   poolDictionaries: ''

Grapher>>pen: aPen
   "Draw using aPen"
   pen := aPen
```

All `Grapher` objects use a `Pen` instance for their drawing. This works fine for our example application, where graphs are drawn on a `GraphPane` (which has its own `Pen`). This also means that the `Grapher` classes can be reused in applications involving any classes which incorporate a `Pen`, such as `Bitmap` and `Printer`.

The Analyst window has two menus: one to let the user select the type of report she wishes to see (e.g., past four quarters gross, past four quarters net, year-to-date sales, etc.), and another to choose the graph type. Each time a new report is selected, `FinancialAnalyst` retrieves the appropriate data from the database and tells its `grapher` to draw the graph. Each time a new graph type is selected from the menu, `FinancialAnalyst` creates a new `Grapher` object and tells it to re-graph the current data. Suppose the graph selection menu looks like this:



The methods corresponding to these menu items are as follows:

```
FinancialAnalyst>>useBarChart
   "The user has selected 'Bar Chart' from the 'Graph
    Type' menu. Create a new BarChartGrapher."
   grapher := BarChartGrapher new
                pen: graphPane pen.
   self drawGraph
```

```
FinancialAnalyst>>useLineGraph
   "The user has selected 'Line Graph'."
   grapher := LineGrapher new
                pen: graphPane pen.
   self drawGraph

FinancialAnalyst>>usePieChart
   "The user has selected 'Pie Chart'."
   grapher := PieChartGrapher new
                pen: graphPane pen.
   self drawGraph
```

Notice that when a graph type is selected, we not only instantiate the appropriate `Grapher` subclass, but also invoke `drawGraph`. This is due to what we noted above, that is, when a new graph type is chosen, the current data is re-graphed using the new graph type. There are two parts to making this work. First, when any report type is requested, `FinancialAnalyst` "remembers" the name of the method used to perform the graphing portion of that report; it does so by saving the method's selector in the variable `graphMessage` (we'll see this soon). Then, when a new graph type is chosen, `FinancialAnalyst` invokes its `drawGraph` method, whose definition follows; it is in `drawGraph` that the grapher/Strategy object is invoked.

```
FinancialAnalyst>>drawGraph
   graphMessage notNil
      ifTrue: [grapher
                  perform: graphMessage
                  with: data]
```

We now need a set of methods for the various reports that users can request. For each report, we'll have two methods: one in `FinancialAnalyst` for retrieving the appropriate data from the database, and a second in the `Grapher` classes for plotting the graph of this information. Each `Grapher` subclass must implement its own version of each of the latter plotting methods. We're not going to show the details of actually plotting the graphs; we leave that to our readers. But, we want to show the overall structure of the code. We start with the report methods defined in `FinancialAnalyst` and the drawing methods in the abstract `Grapher` superclass; we'll generically define all the plotting methods in `Grapher` so as to identify which methods must be implemented by its concrete subclasses. In the following, note how each report saves the drawing method's selector in `graphMessage`.

```
FinancialAnalyst>>past4Qgross
   "User selected the 'Gross income for the past four
    quarters' report from the 'Report Type' menu. Gather
    the data from the database; then graph it"
   data := FinancialDatabase current past4QGross.
   graphMessage := #drawPast4Q:.
   self drawGraph
```

```
FinancialAnalyst>>salesYTD
    "Retrieve the year-to-date sales info from
     the DB and graph it"
    data := FinancialDatabase current salesYTD.
    graphMessage := #drawSalesYTD:.
    self drawGraph
```

And likewise for each report made available by the Financial Analyst. Now we need the corresponding plotting messages in `Grapher`:

```
Grapher>>drawPast4Q: data
    "Draw the graph for the past four quarters'
     financial data"
    self implementedBySubclass

Grapher>>drawSalesYTD: data
    "Draw the graph for the year-to-date sales data"
    self implementedBySubclass
```

Next, we define the `Grapher` subclasses including the concrete implementations of the plotting messages:

```
Grapher subclass: #BarChartGrapher
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

BarChartGrapher>>drawPast4Q: data
    "Draw a bar chart depicting the quarterly results
     information contained in 'data'."
    ...

BarChartGrapher>>drawSalesYTD: data
    "Draw a bar chart for the year-to-date
    sales data showing a bar per month."
    ...

Grapher subclass: #LineGrapher
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

LineGrapher>>drawPast4Q: data
    "Draw a line plot depicting the quarterly results
     information contained in 'data'."
    ...

LineGrapher>>drawSalesYTD: data
    "Draw a line graph for the year-to-date
    sales data."
    ...
```

And so on. The generalized structure of the Strategy pattern appears in the Structure section—relating that diagram to our example, the `FinancialAnalyst` class corresponds to `Context`, and `Grapher` represents the abstract `Strategy`, defining the common interface for all concrete `Grapher` (Strategy) subclasses. `BarChartGrapher`, `LineGrapher`, and `PieChartGrapher` map to `ConcreteStrategyA`, `B`, and `C`.

There's a lot more in this Financial Analyst example other than the Strategy pattern. For instance, we sent a message to the `Grapher` object using `perform:with:`, and we assumed the database object to be a Singleton (and retrieved it from its class with the message `FinancialDatabase current`). But, with regard to the Strategy pattern, the important points are that we have a separate strategy object (defined by the `Grapher` classes) which implements the various graphing strategies, and the application delegates the graphing functionality to this helper object rather than implementing it itself.

Implementation

Here are several implementation issues raised by our example.

implementedBySubclass and subclassResponsibility

Notice that methods in the abstract `Grapher` class are defined with a single statement: `self implementedBySubclass`. Unfortunately, there is nothing in Smalltalk which enforces a class' "abstractness"—that is, there is nothing in the language or environment which prevents a client from creating an instance of the `Grapher` class which we intend to be an abstract class. By convention, programmers add class and code comments to let other programmers know a class is *intended* to be an abstract superclass—that is, it is intended not to be instantiated, but rather serves as a template for its concrete subclasses, defining (at least a portion of) the generic interface protocol for its subhierarchy. But, there is nothing that actually prevents the instantiation of this class.

However, there is a mechanism by which we may *discourage* the instantiation of a class. If a class is intended never to have instances, it may define its generic protocol via methods which include the expression `self implementedBySubclass` or `self subclassResponsibility`, depending on the Smalltalk dialect: in Visual Smalltalk, we use `implementedBySubclass`; in VisualWorks and IBM Smalltalk, the equivalent message is named `subclassResponsibility`. In all cases, the method is defined in the `Object` class, and when invoked, causes a walkback. Thus, for example, if some code creates an instance of `Grapher` and sends it the `drawPast4Q:` message, the user will get a walkback and the programmer will be "made aware" of the error of his ways. In a similar fashion, including `self subclassResponsibility` (or `self implementedBySubclass`) in an abstract superclass' method *forces* immediate subclasses to implement their own version of that method—if they don't override the

otherwise inherited implementation, a walkback will also be produced when a client sends that message. At a minimum, including this message in an abstract class' method acts as documentation for other programmers, informing them that (a) the class is in fact designed to be abstract, and (b) the programmer creating new subclasses must override the methods so defined. Thus, using `subclassResponsibility` is good programming practice.

An alternative approach to preventing the instantiation of an abstract class would be to override the class' instance creation method as follows:

```
AnAbstractClass class>>new
    self error: 'AnAbstractClass is an abstract class. ',
                'You cannot create instances of it!'
```

The `error:` message will, of course, cause a walkback. One consequence of such an approach, however, is that subclasses of `AnAbstractClass` cannot invoke their abstract class' `new` method via the use of `super`, as Smalltalk programmers often do:

```
AnAbstractClassSubclass class>>new
    ^super new initialize
```

Doing so would result in invoking the `error:` code which causes a walkback. Instead, subclasses would have to send the `basicNew` message defined in the `Behavior` class (and thus inherited by all class objects):

```
AnAbstractClassSubclass class>>new
    ^self basicNew initialize
```

This solution, as well as other approaches for dealing with the abstract class instantiation problem, are discussed in detail by LaLonde (1994, see sections 6.1.9–6.1.10). Liu (1996) also discusses `subclassResponsibility` and other approaches for implementing abstract classes in Smalltalk.

Coupling the Context and Strategy Objects

When the `FinancialAnalyst` needs to graph its data, it calls upon its `Grapher` object. An implementation issue that arises here is how to structure the interface between these two objects (and, in the general case, between the Strategy and Context objects) so as to share information. In fact, the issue of how to couple two objects such that they can share information is involved in many of our patterns. As discussed in *Design Patterns*, there are several possible approaches.

In the `FinancialAnalyst` case, the information to be shared is the `Pen` with which the `Grapher` should draw, and the data to plot. These represent different categories of shared information: static—that which will not change during the running of the application (the `Pen`), and dynamic—that which changes over time (the data). In the static case, we can inform the helper object, in this case the `Grapher`, once and be done with

it. On the other hand, we may need to make the helper aware of dynamic information each time the helper is invoked.

Message Arguments. One possibility is to have the `FinancialAnalyst` pass all information to `Grapher` operations in the form of message arguments, thereby keeping the two objects loosely coupled. Thus, a message sent to the `Grapher` might look like:

```
FinancialAnalyst>>drawGraph
   grapher drawPast4Q: data using: graphPane pen
```

Of course, in this case, we would have defined the graph-drawing methods in the `Grapher` classes with two arguments, such as:

```
Grapher>>drawPast4Q: data using: aPen
```

In our `FinancialAnalyst` example, we actually used the `perform:with:` approach to generically send all graph-drawing messages to the `Grapher`, so the message-send with the additional argument would look like:

```
FinancialAnalyst>>drawGraph
   grapher
      perform: graphMessage
         with: data
         with: graphPane pen
```

And since we also included a guard clause to verify `graphMessage` is not `nil`, the final method would look like:

```
FinancialAnalyst>>drawGraph
   graphMessage notNil
      ifTrue: [grapher
                  perform: graphMessage
                  with: data
                  with: graphPane pen]
```

However, the `Pen` object is a static bit of shared data. Thus, we can tell the `Grapher` object about this once, up front, rather than each time we draw the graph. This is what we did above—we passed the `Pen` to the `Grapher` as soon as we had instantiated it, as in:

```
FinancialAnalyst>>useBarChart
   grapher := BarChartGrapher new pen: graphPane pen.
   self drawGraph
```

The `drawGraph` method passes along the dynamic `data` object which *must* be sent each time we draw a graph, but the `Pen` can be set up once. By the way, we also could have set this up using an instance-creation method other than `new` in `Grapher`, as follows:

```
Grapher class>>using: aPen
    "Instance creation method."
    ^self new pen: aPen

FinancialAnalyst>>useBarChart
    grapher := BarChartGrapher using: graphPane pen.
    self drawGraph
```

Self Delegation. Another possible approach would have the `FinancialAnalyst` pass *itself* as an argument to the `Grapher` and let the `Grapher` explicitly request any data it needs from the `FinancialAnalyst` by direct message-sends. Beck (1997) refers to this as *self delegation*. With this approach, when the `FinancialAnalyst` wants to draw a graph, it sends:

```
FinancialAnalyst>>drawGraph
    graphMessage notNil
        ifTrue: [grapher
                    perform: graphMessage
                    with: self]
```

The `Grapher`, thus supplied with a pointer to the `FinancialAnalyst` (via the `with: self`), can send the `FinancialAnalyst` messages asking for information such as the data to plot and the `Pen` to use to draw. So, we would have defined the graph-drawing methods in the `Grapher` classes with one argument, the `FinancialAnalyst`, as in this example:

```
BarChartGrapher>>drawPast4QFor: aFinancialAnalyst
    | data |
    data := aFinancialAnalyst data.
    "Draw the bar chart:"
    ...
```

In fact, if we don't adopt the approach of setting up the `Grapher` object with its drawing `Pen` at instance-creation time, this method might look like:

```
BarChartGrapher>>drawPast4QFor: aFinancialAnalyst
    | data pen |
    data := aFinancialAnalyst data.
    pen := aFinancialAnalyst penToDrawWith.
    "Now, draw the bar chart:"
    ...
```

Of course, the `FinancialAnalyst` class must now define a more elaborate interface providing access to its internal objects (e.g., methods such as `data` and `penToDrawWith`). As a result, (a) `FinancialAnalyst`'s encapsulation is "weakened" (it now allows *any* outsider access to its internal objects rather than providing them as arguments in messages it chooses to send) and (b) the `FinancialAnalyst` and its `Grapher` are more tightly coupled. Thus, in the current

example, this may be a less desirable solution, but in cases where lots of information must be shared, it may be a preferable approach.

Back Pointer. A slight variation on the self-delegation option would have the `Grapher` object maintaining a back pointer to the `FinancialAnalyst` in the form of an instance variable. Actually, implementation-wise, it will be a minor variation, but conceptually it's somewhat different: we can consider the `FinancialAnalyst` itself as a piece of static information to be shared with the `Grapher` object—thus, we can share this once, at `Grapher` instance-creation time. Then, any dynamic information the `Grapher` requires can be requested on the fly. Thus, the `Grapher` class would be defined with a `financialAnalyst` instance variable, and would implement an associated setter message and a new instance creation method.

```
Object subclass: #Grapher
   instanceVariableNames: 'financialAnalyst'
   classVariableNames: ''
   poolDictionaries: ''

Grapher>>financialAnalyst: aFinancialAnalyst
   financialAnalyst := aFinancialAnalyst

Grapher class>>for: aFinancialAnalyst
   "Instance creation"
   ^self new financialAnalyst: aFinancialAnalyst
```

This implementation would, of course, be inherited by `Grapher`'s concrete subclasses. This changes the instance creation message-send as follows:

```
FinancialAnalyst>>useBarChart
   grapher := BarChartGrapher for: self.
   self drawGraph
```

Now, messages sent by the `FinancialAnalyst` to its `Grapher` are simpler: they contain no arguments:

```
FinancialAnalyst>>drawGraph
   graphMessage notNil
      ifTrue: [grapher perform: graphMessage]
```

As in the self-delegation case, the `Grapher` requests information directly from its associated `FinancialAnalyst`, but now via messages sent to its `financialAnalyst` instance variable:

```
BarChartGrapher>>drawPast4Q
    | data pen |
    data := financialAnalyst data.
    pen := financialAnalyst penToDrawWith.
    "Now, draw the bar chart:"
    ...
```

One potential drawback to this approach is that, since the Strategy object (the `Grapher`)directly points to a single Context object (the `FinancialAnalyst`), the Strategy object can no longer be shared among different Contexts. Therefore, there may exist applications for which self-delegation is preferable to a persistent back pointer.

Multiple Simultaneous Strategies, One Active at a Time

Sometimes an application can be linked to multiple Strategy objects simultaneously, although only one is *active* at any point in time. That is, rather than creating a new Strategy object each time the algorithm or strategy changes, all of the possible Strategy instances can be created when the application starts, and the application can switch among them. This can be done when the extra instances are not expensive, memory-wise, to keep around, and is preferable when instantiating and initializing Strategy objects is expensive time-wise. This approach implies some way of "remembering" the instantiated Strategy objects, that is, of keeping pointers to them in the main application. One approach would be to have an instance variable reference a `Dictionary` containing all of the Strategy objects and another variable to point to the currently active instance.

Exemplifying this approach with the Financial Analyst, we'll continue to use the `grapher` variable to point to the active `Grapher` object. The `Dictionary` containing all `Grapher` instances will be referenced by a variable named `allGraphers`:

```
ViewManager subclass: #FinancialAnalyst
    instanceVariableNames:
        'graphPane data grapher graphMessage allGraphers'
    classVariableNames: ''
    poolDictionaries: ''
```

We add a method to initialize `allGraphers` and we modify the `open` method to invoke this initialization method:

```
FinancialAnalyst>>initializeGraphers
    "Set up the Dictionary containing all my Strategy
objects."
    allGraphers := Dictionary new.
    allGraphers
        at: #BarChart   put: (BarChartGrapher for: self);
        at: #LineGraph  put: (LineGrapher for: self);
        at: #PieChart   put: (PieChartGrapher for: self)
```

```
FinancialAnalyst>>open
    "Create the subpanes for, and open, the Analyst window."
    self
        label: 'Financial Analyst';
        addSubpane:
            ((graphPane := GraphPane new)
                owner: self;
                ...).
    "Setup all of my graphers and a default grapher:"
    self
        initializeGraphers;
        grapher: #LineGraph;
        openWindow.
```

We'll change the `grapher:` setter method to require a `Symbol` as its argument rather than a `Grapher` instance. The `Symbol` specifies the key in the `allGraphers Dictionary`. We'll also move the `drawGraph` message-send here so each time we change the graphing strategy, we redraw the graph (if possible):

```
FinancialAnalyst>>grapher: aSymbol
    "Change my current graphing strategy. Also redraw
     the current data (if any) using the new grapher"
    grapher := allGraphers at: aSymbol.
    self drawGraph
```

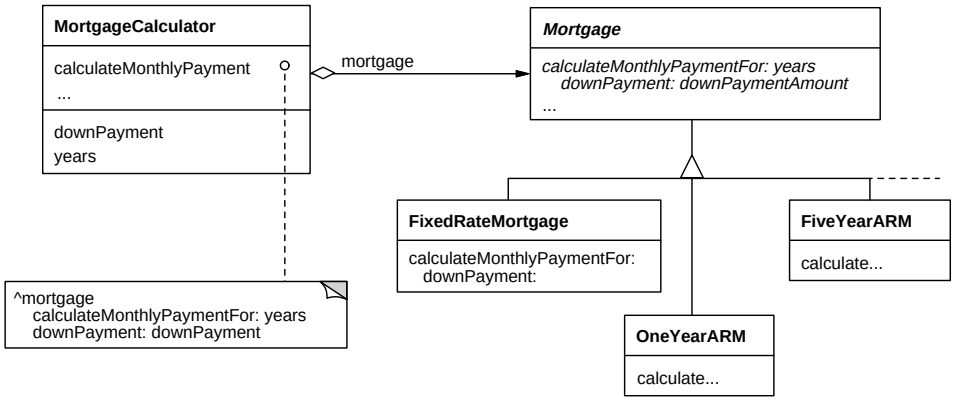
Finally, when the user selects an item from the "Graph Type" menu, we invoke `grapher:`, as follows:

```
FinancialAnalyst>>useBarChart
    self grapher: #BarChart

FinancialAnalyst>>useLineGraph
    self grapher: #LineGraph

FinancialAnalyst>>usePieChart
    self grapher: #PieChart
```

Domain-Specific Objects as Strategy Objects

A Strategy object need not merely reify a particular algorithm; it may also be a bona fide application-specific object that also assumes a Strategy role. Here's an example from a financial domain that demonstrates this approach. Our example application concerns mortgages. Let's start with a simple scenario: a potential customer walks into a bank and wants to know how much her monthly payments would be for a loan of a specific amount. The problem is, there are many different types of mortgages and each results in different monthly payments for the same prinicipal amount. So, imagine a Mortgage Calculator application: the user interacts with the system by selecting a mortgage type, a down payment amount, and the length of time she wants the mortgage to run, and the

application calculates the monthly payments. This sort of application could be used in kiosks on a banking floor or on a bank's Web site. The structure for this application might look like this:



When the user selects a mortgage type, one of the `Mortgage` subclasses is instantiated (by the way, ARM stands for "adjustable rate mortgage"). Then, to determine the monthly payments, the `Mortgage` instance itself acts as a Strategy object for the application and performs the algorithmic calculation—the choice of algorithm is coincident with the choice of Mortgage-type. So, domain-specific objects that have other responsibilities may also act as Strategies.

Known Smalltalk Uses

ImageRenderer

In VisualWorks, `ImageRenderer` "is an abstract class *representing a technique* to render an image using a limited palette" (from the class comment, emphasis ours). An `ImageRenderer` paints an image on a graphics device using an appropriate color palette for that device (e.g., some screens support 256 colors whereas others support only 16; the `ImageRenderer` must map colors in the image to those in the supported palette). Concrete subclasses of `ImageRenderer` encapsulate different rendering algorithms by providing unique implementations of a common rendering message protocol. `ImageRenderer` subclasses include `NearestPaint`, `OrderedDither`, and `ErrorDiffusion`–again, from the class comment: "Among the subclasses are mapping techniques such as `NearestPaint` and halftoning techniques such as `OrderedDither` and `ErrorDiffusion`." Different graphics devices (e.g., printers and the screen) instantiate different `ImageRenderer` subclasses to perform rendering depending on their capabilities (gray-scale vs. color, color/gray-scale depth), but code that sends messages to these rendering objects doesn't care what type of `ImageRenderer` it's talking to because they all implement the same message interface.

View-Controller

In the Model-View-Controller (MVC) framework, the view-controller relationship is an example of the Strategy pattern. A `View` instance (representing a screen widget) uses a `Controller` object to handle and respond to user input via mouse or keyboard. For a `View` to use a different user-interaction strategy, it can instantiate a different `Controller` class for its controller object. This can even occur at runtime to change the user interaction style on the fly; for example a `View` can be disabled (so it does not accept any user input) by instantiating and switching to a controller that ignores input (this example is also mentioned on DP 6).

Insurance Policy Policies

Philip Hartman of ISSC Object Technology Services used the Strategy pattern in a Smalltalk application for an insurance company, in order to implement different business logic for individual automobile insurance policies. One requirement of the application was that the logic for calculating the cost of insurance had to vary by insurance company subsidiary and/or by the state of residence of the policy holder. For example, "points" are charged against drivers based on violations and accidents, and different subsidiaries and states use different rules for determining how, and how many, points are assigned. In many cases, this has a large effect on the premium charged. In the implementation, a `Policy` object represents an insurance policy and points to an instance of one of the concrete subclasses of `PointAssignmentRule`. Each `PointAssignmentRule` subclass encapsulates a different algorithm for assigning points to drivers on the policy. Thus, by instantiating one or another of these classes, the policy logic could be made to vary, without varying the code of the `Policy`.

Related Patterns

The careful reader will notice that the structure diagrams for Strategy are isomorphic to those for the Builder pattern. We might even consider Builder a specialization of Strategy. The difference in the patterns is when and where they are used and the functionality of the helper objects. In the Builder pattern, the helper object has the job of creating a Product in a step by step fashion: the Director object iteratively calls upon it to build subcomponents and then once for the final Product. In the Strategy pattern, the Strategy object acting as an external helper to a Context object is intended to encapsulate any algorithm—not necessarily for creational purposes, but for any runtime service. Multiple algorithms are encapsulated outside of the "main" Context object and each is reified as an object. The Strategy object is called on periodically, as needed, to perform a complete standalone task in one shot.

Strategy and Builder are also related structurally and thematically to Abstract Factory. In the latter, an external helper object is called upon to create some Product on behalf of the main application as well. There may be multiple factory objects to choose from and they all provide the same abstract interface. Thus, the factory's client doesn't know or

care the exact class of the factory. It just sends a generic message to its current factory when a particular type of object has to be constructed. Again, this is similar in structure to the Strategy pattern, but is divergent in intent and when and where it is used: an Abstract Factory is used for one-shot object creation, rather than periodic algorithm execution as in Strategy.

Admittedly, the lines between these patterns is sometimes blurry. In our Financial Analyst example, the `Grapher` Strategy objects might be said to be producing a product: each produces a graph based on the data it receives. But, Strategy may be used for any other runtime service when that service may be implemented in multiple ways.