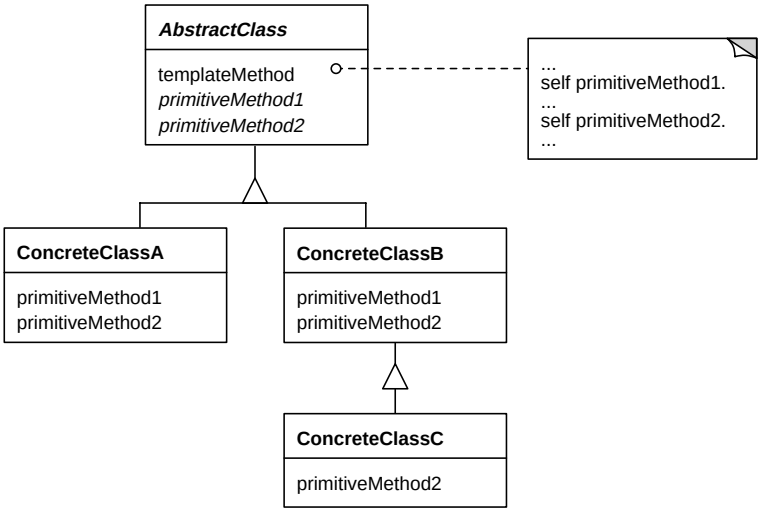
TEMPLATE METHOD (DP 325) Object Behavioral

Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Structure



Description

Template Method may be #1 on the Design Patterns Pop Charts. Its structure and use are at the heart of much of object-oriented programming. Template Method turns on the fundamental object-oriented concept of inheritance. It relies on defining classes that differ only *slightly* from an existing class. It does this by relying on the ability of classes to provide for new behavior by overriding inherited methods.

The key to the Template Method pattern is a method that implements a broad message in terms of several narrower messages in the same receiver. The broader method, a Template Method, is implemented in a superclass and the narrower methods, Primitive Methods, are implemented in that class or its subclasses. The Template Method defines a general algorithm and the Primitive Methods specify the details. This way, when a subclass wants to use the algorithm but wishes to change some of the details, it can override the specific Primitive Methods without having to override the entire Template Method as well.

Reuse through Inheritance

Code and attribute sharing through inheritance is one of the most maligned techniques in object-oriented programming. It is deceptively easy to use, but often difficult to use well. Some authorities have dismissed implementation inheritance as confusing and unnecessary. As early as the first OOPSLA, papers have documented potential problems with inheritance (e.g., Snyder, 1986). The object-oriented language SELF (Ungar & Smith, 1987) does not even support class inheritance; instead, it uses object composition exclusively. *Design Patterns* advocates favoring object composition over class inheritance (pages 18-21). Many patterns transform code that relies on class inheritance to use object composition instead: State, Strategy, Decorator, Bridge, and Abstract Factory are a few of them.

Nevertheless, inheritance is a fundamental feature of most object-oriented languages, including Smalltalk. Good class hierarchies allow for reuse of abstraction, design, and implementation. Because

of inheritance, these hierarchies are easier to understand, maintain, and extend, and their classes are much more reusable. We can define new classes which differ slightly from existing classes by specializing those existing classes rather than starting from scratch. Even as *Design Patterns* shows poor examples of inheritance that work better as composition, it still advocates using inheritance to define and implement interfaces (pages 14-18). Most of the patterns feature at least one class hierarchy with an abstract class at the top that defines an interface and several concrete subclasses that implement it: Adapter's Target hierarchy, Composite's Component hierarchy, State's State hierarchy, etc.

The Template Method pattern encourages proper use of inheritance. The pattern can also be very helpful for refactoring related classes that share common behavior (DP 326). The key to factoring for reuse is to separate the stuff that changes from the stuff that doesn't, and Template Method encourages this by factoring the reusable code into the superclass and the specifics that can change into the subclasses.

How Template Method Works

Abstract classes and the Template Method pattern go very much hand-in-hand. An abstract class is a superclass that defines the behavior of the classes in its hierarchy while deferring the implementation details to its subclasses. The superclass is abstract because it does not implement all of the behavior it defines. The subclasses are concrete because each one implements all of the behavior, either by inheriting it from its superclasses or implementing the behavior itself.

Template Method is the same technique at the method level instead of the class level. Here is what a Template Method generally looks like:

```
AbstractClass>>algorithmA
   self
      algorithmAStep1;
      algorithmAStep2;
      algorithmAStep3
```

The class defines a method, such as `algorithmA`, for each unit of behavior that it can perform. But the method does not do much work; instead, it provides a template listing the steps to be performed, but delegates each of the steps back to the receiver (`self`). In this way, the Template Method gets to decide *what* steps are done, but the receiver gets to decide *how* each step is done. This separates the steps which must be performed from how they are performed.

The beauty is that not only can the abstract class define how each of these steps is done, but each concrete subclass has the opportunity to override each of these decisions. If a concrete class generally likes the way `algorithmA` works, but doesn't like the way `algorithmAStep2` works, it can subimplement (override) `algorithmAStep2` without changing `algorithmA` or the superclass' implementation of steps 1 and 3. This allows the subclass to override just the parts that it needs to while inheriting the rest, which minimizes the amount of code each subclass must implement itself and emphasizes the subclass' differences with its superclass.

Types of Methods

There are four types of methods which the abstract class may implement: template, concrete, abstract, and hook (DP 327-328). The last three are types of Primitive Methods.

1. *Template.* A Template Method is one in the abstract class that combines concrete, abstract, and/or hook methods together into an algorithm. Subclasses usually inherit it unchanged, although they may subimplement the methods invoked therein—this is the heart of the Template Method pattern.
2. *Concrete.* A Concrete Method is one that the abstract class defines and that subclasses do not subimplement.

3. *Abstract.* An Abstract Method is one in the abstract class that "declares" a message but defers its implementation to the subclasses. Each subclass *must* subimplement the abstract methods it inherits. (*Design Patterns* refers to these as "Primitive operations." This should not be confused with a "primitive" in Smalltalk, which is a different concept entirely.)
4. *Hook.* A Hook Method is one in the abstract class that declares a message's behavior and provides a default implementation for it. Sometimes the default is simply to do nothing. Subclasses *may* subimplement the method to change this default.

Design Patterns also lists a fifth type, Factory Method, but this is really just a special case of a concrete or hook method (see Factory Method).

It is usually easy to identify which methods ought to be concrete. It is more difficult to determine which methods will be abstract and which will be hooks. The question you need to ask here is: Is there a reasonable default implementation for this method? If so, the abstract class' implementation should be a hook method. If not, the method will be abstract.

According to *Design Patterns* (page 329), one design goal is to minimize the number of primitive operations that subclasses must override. One way to reduce the number of primitive operations (abstract methods) is to implement those messages as hook methods instead. An abstract method *forces* a developer implementing a subclass to override the method. On the other hand, he can choose to inherit a hook method unchanged. In this way, hook methods make subclassing easier than abstract methods.

A potential problem with using a hook method rather than an abstract one is that it is may not be obvious to the subclass developer that the method is a candidate for overriding. Only class or method documentation provides this level of information. If a subclass developer forgets to override an abstract method, he will discover his mistake when he runs the code and gets a walkback from a `subclassResponsibility` error. But, should he inadvertently inherit inappropriate behavior from a hook method, his code will run, but not necessarily correctly.

A "Do Nothing" Hook

As mentioned above, a hook method defined in an superclass must provide a reasonable default of an operation for its subclasses. A special case of a reasonable default is an implementation of the method that does nothing, that is, only returns `self`. This sort of "do nothing" method is often useful when there is the potential for an operation to occur, but not all subclasses may need that operation. Two examples of this sort of method are VisualWorks' `ApplicationModel>>preBuildWith:` and `postBuildWith:` methods. These methods provide hooks for those `ApplicationModel` subclasses that need to modify the behavior of UI widgets at two different points in the UI building process. However, `ApplicationModel` itself does not need to do anything at either stage, so it implements the messages to do nothing. This default behavior is appropriate for most subclasses as well.

Another example of this, from IBM Smalltalk, is `Class>>initialize`. Not all classes need initialization, so the default implementation is to do nothing—this makes it possible to safely use `super initialize` in any class's initialization method. By the way, this is something you should not do in class methods in VisualWorks; in VisualWorks, `Object class>>initialize` has the undesirable side effect of removing all dependents!

Another example is `Stream>>close` in VisualWorks and Visual Smalltalk. It is often useful to have clients be unaware of what kind of stream they are using, be it on an internal collection or an external object (like a socket or file handle). Thus, the default implementation of `close` is to do nothing. External stream subclasses redefine this method to clean up external resources.

Designing and Refactoring Hierarchies

When you are refactoring, or initially designing, a hierarchy for reuse, it is often beneficial to move behavior up the hierarchy and state down the hierarchy. Auer (1995) describes this process in four heuristic patterns:

1. *Define Classes by Behavior, not State.* Initially implement a class to define its behavior without regard to its structure.
2. *Implement Behavior with Abstract State.* Implement behavior that needs state information to access that state indirectly through messages rather than referencing the state variables directly.
3. *Identify Message Layers.* Implement a class' behavior through a small set of kernel methods that can easily be subimplemented in subclasses.
4. *Defer Identification of State Variables.* The abstract state messages become kernel methods that require state variables. Rather than declaring those variables in the superclass, declare them in a subclass and defer the kernel methods' implementation to the subclass.

This process develops a hierarchy that separates interface from implementation. A superclass abstractly defines the hierarchy's behavior and one or more subclasses implement it. The superclass is easy to subclass because it uses kernel methods and because it does not force subclasses to inherit state that they may not need.

This process leads to Template Methods and primitive methods. The primitive methods are what Auer's patterns call kernel methods. The Template Methods are the layers that use the kernel methods. What these patterns also show is that Template Method can be nested; each message layer has its own template/primitive relationship. The primitive messages that a Template Message sends can themselves be Template Methods that send other primitive messages.

Here's an example of nested Template Methods from the `Collection` hierarchy in VisualWorks:

```
Collection>>removeAll: aCollection
    "Remove each element of aCollection from the receiver.
     ..."

    aCollection do: [:each | self remove: each].
    ^aCollection

Collection>>remove: oldObject
    "Remove oldObject as one of the receiver's elements. ..."

    ^self remove: oldObject ifAbsent: [self notFoundError]

Collection>>remove: oldObject ifAbsent: anExceptionBlock
    "Remove oldObject as one of the receiver's elements. ..."

    self subclassResponsibility
```

When a `Collection` receives `removeAll:`, the implementation is a Template Method that delegates to the primitive message `remove:`. However, `remove:`'s implementation is not primitive, it is another Template Method that delegates to `remove:ifAbsent:`. Finally, `remove:ifAbsent:` really is a primitive method, an abstract method that delegates to its subclasses. We'll see another example of this iterative Template Method implementation in the `WindowPolicy` example in the Known Smalltalk Uses section.

What to Delegate

When a method is well written, all of its tasks should be performed at the same level of abstraction. That is to say, its steps shouldn't look like, "Do something general; Do something general; Do

something very specific." Instead, the specific code should be factored into another method that this one calls. Then all of this method's steps are at the same level of generality or specificity.

Certain code details jump out as being too specific for general algorithms. These details may include constants, classes, and specific algorithms or portions thereof.

Delegating Constants. One obvious case for a simple Template Method is in factoring out the "declaration" of a constant into a separate method, rather than hardcoding it inline. You usually specify constants in Smalltalk either through the use of a Pool variable, or by defining a separate method that simply returns the constant ("Constant Method," Beck, 1997). Pool dictionaries have many drawbacks. The main arguments against them are that they are poorly supported by the language and class libraries, and that they adversely affect reuse (Ewing, 1994). Because of these drawbacks, specifying a method to return a constant is the more flexible approach, because subclasses can implement the method differently to return different constant values. In this way, the methods that *use* that constant are Template Methods, since the method returning the constant is now a hook.

You can see an example of defining a constant value that differs in various subclasses in the `EtBrowser` hierarchy in IBM Smalltalk. The superclass `EtBrowser` defines the method `classDoubleClickSelector` to return `nil`. This means that nothing special happens when you double-click the selection in the classes list of a browser. Subclasses redefine this method to return a message selector. This message selector is used to do something else: in some classes, double-click means browse the selected class; in others, it means expand or contract the class hierarchy rooted at this class.

Also in IBM Smalltalk, the message `fixedSize` in the `OsObject` hierarchy returns the size (in bytes) of the structure each `OsObject` subclass represents. This use of polymorphism allows Template Methods defined in `OsObject`, like `copyToOsMemory`, to operate on any structure, regardless of its actual size.

Delegating Classes. Smalltalk classes are a special case of constants. In the Factory Method pattern, a Smalltalk method returns a class and that class is then used to create instances (see Factory Method). In general, even if you do not wish to use the Factory Method pattern initially, it is a good idea to isolate class names into their own methods. Doing so makes it easier to maintain the code if the name of a class changes.

Delegating Subparts of Algorithms. Often you may want to isolate away a particular sub-part of an algorithm that varies from class to class—again, this is the main motivation behind the Template Method pattern. Part of the challenge of OO design is finding and setting up such relationships between algorithms and their component subtasks—something that simply cannot be done in procedural languages because they lack inheritance and delegation.

You can find an example of this in VisualWorks in the `Controller` hierarchy. One of the fundamental parts of the VisualWorks windowing system is the notion of the "control loop" that defines how each `Controller` gains control of the mouse. A Template Method named `controlLoop`, shown below, implements this notion:

```
Controller>>controlLoop
   [self poll.
   self isControlActive]
      whileTrue: [self controlActivity]
```

The message `controlActivity` is a hook method defined in `Controller` as simply passing control along to the next level of controllers. Subclasses of `Controller` override this method to perform a specific action whenever they receive control. It is interesting to note that one subclass of `Controller`, `ControllerWithMenu`, overrides this method with yet another Template Method, shown below.

```
ControllerWithMenu>>controlActivity
    self sensor redButtonPressed & self viewHasCursor
        ifTrue: [^self redButtonActivity].
    self sensor yellowButtonPressed & self viewHasCursor
        ifTrue: [^self yellowButtonActivity].
    super controlActivity
```

Subclasses of `ControllerWithMenu` can override the messages `redButtonActivity` and `yellowButtonActivity` to perform specific actions while the "yellow" or "red" mouse buttons are depressed.

A Case Study in Refactoring

Examining a case study in refactoring can help to understand how Template Methods arise during this process, and how the issues discussed above play themselves out. Probably the best known and best described example of refactoring in Smalltalk was the refactoring of the Smalltalk-80 `View` classes between Objectworks\Smalltalk 2.5 and Objectworks\Smalltalk 4.0.

As both Johnson and Foote (1988) and Rubin and Liebs (1992) describe, the original class `View` in Smalltalk-80 was extremely large and unwieldy. It suffered from trying to be all things to all people. Views were capable of edge decoration (knowing their border thickness and color), containing and managing subviews, and handling their own layout inside a parent view. They were also responsible for communicating with a model (via the Observer pattern), and communicating with their associated controller. As a result, each `View` contained a large number of instance variables; however, each instance only used a subset of those variables.

Objectworks\Smalltalk 4.0 (the precursor to VisualWorks) solved this problem through a significant refactoring of the `View` hierarchy, as described by Rubin and Liebs (1992). Each of the previously listed responsibilities of `View` were refactored into different classes in the `View` hierarchy. There are now three sets of subclasses of the abstract visual superclass `VisualPart`. The `View` hierarchy consists of those objects that have a model and controller. The `Wrapper` hierarchy consists of those objects that use the Decorator pattern to modify the display or positioning of other visuals. Finally, the `CompositePart` hierarchy consists of those visuals that use the Composite pattern to combine other visuals. As a result, the classes are much lighter weight and a developer can choose more carefully where he will subclass to inherit only those parts that he needs.

One of the other crucial points to emerge during the refactoring was the previously described principle of trading state for behavior in an abstract superclass. In particular, a problem arose when trying to deal with control in the different types of visuals that resulted from the refactoring.

In the refactored hierarchy, the three basic branches described above suffice for most purposes. In most cases, `View`s (visuals with `Controller`s) are the leaves of a visual tree made up of Composites and Decorators. However, there is one case where this breaks down. Occasionally you need a `View` that has a controller, but that also contains other visuals; this is a `CompositeView`. An example of a `CompositeView` is the VisualWorks canvas editor, which is an instance of `UIPainterView`. The individual views within a `UIPainterView` have their own controllers that describe how they react when they are selected and moved. However, the `UIPainterView` itself also needs a controller to describe how the menu for the canvas as a whole operates. At first glance this problem of how to give `View` behavior to a subclass of `CompositePart` appears to require multiple inheritance. However, VisualWorks solves it through a clever application of Template Method.

The abstract superclass `VisualPart` implements the method `objectWantingControl` as a Template Method. The method `objectWantingControl` returns the controller in the receiver's visual tree that wants control. The method utilizes the method `getController` which is a hook method. Here's the method from the VisualWorks image:

```
VisualPart>>objectWantingControl
   "The receiver is in a control hierarchy and the
    container is asking for an object that wants control.
    If no control is desired then the receiver answers
    nil. If control is wanted then the receiver answers
    the control object."

   | ctrl |
   ctrl := self getController.
   ctrl isNil ifTrue: [^nil].
   "Trap errors occurring while searching for
   the object wanting control."
   ^Object errorSignal
      handle: [...]   "Handle error"
      do: [ctrl isControlWanted
             ifTrue: [self]
             ifFalse: [nil]]
```

The default behavior of `getController` as defined in `VisualPart`, is to return `nil` meaning there is no `Controller` that wants control. However, this method is reimplemented as state in the subclasses `View` and `CompositeView` to return their respective controllers. In this way, the objects in both hierarchies can get the behavior they want without unnecessary code duplication.

There are many other uses of Template Method in the VisualWorks visual component hierarchy. Finding them and understanding their use can be an enlightening exercise for the interested reader.

Implementation

Here's one way to implement a complex method that we want to be a Template Method:

1. *Simple implementation.* Implement all of the code in one method. When trying to factor code, it is often more convenient to just write the code before trying to refactor it. This large method will become the Template Method.
2. *Break into steps.* Use comments to break the method apart into logical steps: use a comment to describe each step.
3. *Make step methods.* Make a separate method out of each step in the Template Method. Put these methods in the same class as the Template Method. Name each step method based on the comment from step #2 and move its code from the Template Method. If the step method needs temporary variables from the Template Method, pass them in as parameters.
4. *Call the step methods.* Simplify the Template Method to perform each step by calling the corresponding step method. What's left in the Template Method is an "outline" of the entire algorithm. This is now a true Template Method.
5. *Make constant methods.* If the Template Method still contains literals, class names, or other constants, factor them into separate methods. For each constant, define a method that does nothing but return the constant. Then change the Template Method to call these methods instead of hardcoding the constants.
6. *Repeat.* Repeat this process for each of the step methods you've created. This will factor them into Template Methods that call primitive methods. Continue until all of the steps in each method have the same level of generality and until all constants have been factored into their own methods.

Sample Code

One of the more common uses of Template Method (described by Auer (1995)) is to isolate away a default value in a lazy initialization method. Consider the following method:

```
Circle>>radius
    radius == nil ifTrue: [radius := 10].
    ^radius
```

While this method works fine in a superclass, the problem occurs when subclasses want to substitute a default value for `radius` other than 10. In this case they must reimplement the `radius` method to include all of the above code except for the constant. A better version of this method makes uses Template Method and defines a hook method called `defaultRadius`. Subclasses can override this default, or they can choose to inherit the superclass' default as is.

```
Circle>>radius
    radius == nil ifTrue:[radius := self defaultRadius].
    ^radius

Circle>>defaultRadius
    "This can be overridden in subclasses."
    ^10
```

Then `BigCircle`, a subclass of `Circle`, can easily change its default radius without having to reimplement the lazy initialization.

```
BigCircle>>defaultRadius
    "Override the inherited default."
    ^100
```

Even if you do not use lazy initialization, the same technique applies in an `initialize` method. Here's the non-Template Method approach:

```
Circle>>initialize
    radius := 10.
    "other variable declarations"

BigCircle>>initialize
    "Have to re-set the value of variableName in the subclass"
    super initialize.
    radius := 100.
```

A better version of the previous implementation would be:

```
Circle>>initialize
    variableName := self defaultRadius.
    "other variable declarations"
```

The implementations of the `defaultRadius` methods are the same as in the previous example.

Known Smalltalk Uses

The previous discussion has given many examples of the use of the Template Method pattern in the various Smalltalk dialects. There are several other places the pattern has been used in the Smalltalk base class libraries. We offer these not only as known uses, but as additional Sample Code examples for the pattern.

WindowPolicy

In Visual Smalltalk, every window has an associated window policy object which is called upon to decide which (if any) pulldown menus should be added to the window's menu bar. `WindowPolicy` defines the interface for all concrete policy classes. Instances of the `SmalltalkWindowPolicy` subclass, for example, ensure that the "Smalltalk" pulldown is added to a window's menu bar. `StandardWindowPolicy` objects add the standard "File" and "Edit" pulldowns. A generic Template Method in `WindowPolicy` is invoked to add all of the pulldown menus; it looks like this:

```
WindowPolicy>>addMenus
    "Private - add the menus for the window to the
     menu bar."
    ...
    self
        addSystemMenus;
        addStandardLeftMenus;
        addApplicationMenus;
        addStandardRightMenus
```

All of these `add...Menus` messages are implemented in `WindowPolicy` as empty methods. Concrete subclasses of `WindowPolicy` override these implementations as necessary. For example:

```
StandardWindowPolicy>>addStandardLeftMenus
    "Private - add the menus that are to be located
    on the menu bar before any application-specific
    menus (File & Edit)."

    self addFileMenu.
    self addEditMenu.
```

Note that this is, in turn, another Template Method, allowing still lower-level subclasses to determine for themselves which menu items to include on the "File" and "Edit" pulldowns.

Collection Examples

The `Collection` hierarchies in the various Smalltalk dialects contain examples of all four types of methods found in abstract classes. Let's look at a couple of examples from VisualWorks.

The method `collect:` is a Template Method that sends several primitive messages:

```
Collection>>collect: aBlock
    "Evaluate aBlock with each of the values of the
     receiver as the argument. Collect the resulting
     values into a collection that is like the receiver.
     Answer the new collection."

    | newCollection |
    newCollection := self species new.
    self do: [:each |
        newCollection add: (aBlock value: each)].
    ^newCollection
```

The message `species` returns the class for an object's "kind," rather than its actual class. Since `Collection` uses it as a primitive message, you might expect to find it implemented in `Collection`, but `Collection` actually inherits it from `Object`:

```
Object>>species
    "Answer the preferred class for reconstructing the
     receiver. ..."

    ^self class
```

This is a hook method because it permits overriding but provides a default implementation that is suitable for most subclasses. One subclass it is not suitable for is `Interval`. It overrides `species` like this:

```
Interval>>species
    "Answer the preferred class for reconstructing the
     receiver, that is, Array."

    ^Array
```

The second primitive method is `do:`. It's default implementation in `Collection` looks like this:

```
Collection>>do: aBlock
    "Evaluate aBlock with each of the receiver's elements
     as the argument."

    self subclassResponsibility
```

It is an abstract method because it does not provide a default implementation and is overridden by numerous `Collection` subclasses. It does not provide a default implementation because the way you iterate through the elements in, say, an `OrderedCollection` versus a `Set` is very different.

The third primitive operation is `add:`. Although it is actually not sent to the receiver (`self`), it is sent to another `Collection` and thus is a primitive part of the larger collecting algorithm. It's base implementation looks like this:

```
Collection>>add: newObject
    "Include newObject as one of the receiver's elements. ..."

    self subclassResponsibility
```

Thus `add:` is another abstract method that all direct subclasses must override.

Another method in `Collection`, one that seems simple enough to be primitive but is actually a Template Method, is `remove:`.

```
Collection>>remove: oldObject
    "Remove oldObject as one of the receiver's elements. ..."

    ^self remove: oldObject ifAbsent: [self notFoundError]
```

The first primitive message is `remove:ifAbsent:`. Like `add:`, it is deferred to subclasses:

```
Collection>>remove: oldObject ifAbsent: anExceptionBlock
    "Remove oldObject as one of the receiver's elements. ..."

    self subclassResponsibility
```

Thus `remove:ifAbsent:` is an abstract method. The important insight is that `remove:` doesn't remove anything; it gets `remove:ifAbsent:` to do all of the work. Thus `remove:` isn't really a primitive method, it's a Template Method.

The other primitive message that `remove:` sends is `notFoundError`.

```
Collection>>notFoundError
    "Raise a signal indicating that an object is not
     in the collection."

    ^self class notFoundSignal raise
```

This works for all of the `Collection` classes, so none of them subimplement it. Thus it is a concrete method because it implements real behavior and the subclasses don't change that behavior. If any subclasses did change the behavior, `notFoundError` would become a hook method.

Magnitude Examples

Another excellent place to look for Template Methods is in the `Magnitude` hierarchy. For instance, consider the abstract class `Magnitude` in IBM Smalltalk. Its Common Language Definition API protocol supplies the messages `>`, `<=`, `>=`, `between:and:`, `max:`, and `min:`. The first method, `>`, is an abstract method that all subclasses must implement. The other methods are Template Methods that are defined in terms of `>`, so any new subclass that implements `>` inherits the others for free. For example, if you implement `DollarAmount` as a new subclass of Magnitude, you would only need to override `>` to gain the rest of the `Magnitude` behavior.

Similarly, in VisualWorks, different `Magnitude` subclasses may implement local versions of the `<` or `>` methods. Other `Magnitude` methods invoke these methods by sending "`self >`" or "`self <`". For example, the `>=` method is defined to use `<` as follows:

```
Magnitude>>>= aMagnitude
    "Answer whether the receiver is greater than or
     equal to the argument."

    ^(self < aMagnitude) not
```

The beauty of this approach is that subclasses may redefine `<` and then need not redefine `>=`.

`Object>>printString`

Perhaps the best example of a hook method providing a reasonable default is `Object>>printOn:`. The Template Method `printString` defined in `Object` operates by using the current implementation of `printOn:` as shown below (these methods are from the Visual Smalltalk image).

```
Object>>printString
    "Answer a String that is an ASCII representation
    of the receiver."

    | aStream aString |
    aString := String new: 20.
    self printOn: (aStream := WriteStream on: aString).
    ^aStream contents
```

The definition of `printOn:` in `Object`, shown below, conveys only a minimum of information about the object receiving the message.

```
Object>>printOn: aStream
    "Append the ASCII representation of the receiver
    to aStream. This is the default implementation which
    prints 'a' ('an') followed by the receiver class name."

    | aString |
    aString := self class name.
    (aString at: 1) isVowel
       ifTrue: [aStream nextPutAll: 'an ']
       ifFalse: [aStream nextPutAll: 'a '].
    aStream nextPutAll: aString
```

However, in some subclasses, more information is useful. For instance, in `String`, the actual characters are shown as a quoted string. To achieve this, `String` overrides the default implementation of `printOn:`.

```
String>>printOn: aStream
    "Append the receiver as a quoted string
    to aStream doubling all internal single
    quote characters."

    aStream nextPut: $'.
    self do: [ :character |
       aStream nextPut: character.
       character = $'
          ifTrue: [aStream nextPut: character]].
    aStream nextPut: $'
```

In this way, `printString` and any other methods that rely on `printOn:` can remain unchanged.