Tactical patterns for the real world: Instantiation patterns

Darrow Kirkpatrick

BETWEEN DESIGN AND code lies a set of important issues that must be mastered to ship a product out into the real world of paying endusers. This article is the first in a three-part series that deals with efficiently implementing and managing domain models using patterns discovered in the course of creating engineering products for numerical modeling.

Domain models are the fundamental objects that represent the business logic of an application and store the state underlying the user interface. The patterns presented in this series of articles form a family of generic approaches for working with these domain models: setting their attributes, validating them, presenting information about them, and optimizing them for size and speed. Each of these patterns represents a generic solution that we have applied over and over during the creation of our engineering products.

Many of these patterns arose out of the need to create shrink-wrapped software that would be competitive in the resource-constrained Windows environment. I call these patterns *tactical* because they operate at the class or method level to solve implementation, not strategic or architectural, problems.

Here, we consider a family of *instantiation patterns*—patterns that aid in creating or initializing domain objects.

INTERIOR DECORATOR (MIX-IN STATE)

Problem: How do you share a set of useful behaviors that may be selectively needed throughout a family of classes in a broad hierarchy without using multiple inheritance?

Motivation: You are developing an application where many of the objects need common behavior for maintaining a label. However, not every kind of object needs this label behavior. You implement the methods for maintaining labels in the top-level class, referencing the label attribute via an accessor method.

```
Object subclass: #AbstractDomainModel
    instanceVariableNames:''
    classVariableNames:''
    poolDictionaries:''

label
    "Subclasses must provide access to label state to
     enable label behavior."
    ^self implementedBySubclass

reportOn: aStream
    "Implement a simple reporting mechanism. Requires
     that label behavior be enabled by subclasses."
    aStream
        nextPutAll: self label;
        nextPutAll: self results
```

You then add label instance variables to only those classes that need labels.

```
AbstractDomainModel subclass: #ConcreteDomainModel
    instanceVariableNames: 'label'
    classVariableNames:''
    poolDictionaries:''

label
    "Answer a string, the object's label. This method
     provides the state needed to enable inherited label
     behavior."
    ^label
```

Applicability: This pattern is for use in single-inheritance environments where the only other implementation choice would be a proliferation of redundant subclasses or methods. It requires that the classes needing to share behavior have a common parent class or that you introduce one. Use this pattern when many classes don't need all the possible behaviors and some need none. Each behavior requires some state to support its implementation. Interior Decorator lets you avoid the size penalty of adding unused state throughout the entire hierarchy.

Solution: Create a fat interface with optional state: Place all the methods to support the required behavior high in the hierarchy in abstract classes. Allocate the state that supports the behavior as needed low in the hierarchy in concrete classes.

Implementation: This pattern requires the use of accessor methods to encapsulate the references to instance variables that are only allocated in concrete classes. Typically, you enable a desired behavior by adding instance variables to the class definition and implementing these accessors.

Consequences: This pattern trades off small amounts of state or instance variable redundancy in the leaves of an inheritance hierarchy to share behavior and avoid code redundancy. Note that this pattern works against type safety: It is possible to send a message to an object requiring a behavior that is present in the class but has not been enabled with the requisite state, resulting in a walkback in a low-level abstract accessor.

Related Patterns: This pattern is like Decorator[1] in that it appears to add selective, small behaviors to a class. However, the behaviors are already present in the parent class and it is the addition of storage for related state that enables them. Interior Decorator does not require replicating the decorated object's public interface in a separate decorator class and does not pay the cost of an extra level of delegation through the decorator.

EPITOME (ATTRIBUTE FACTORY)

Problem: How do you consolidate and share the default values for an object's attributes?

Motivation: You are designing a domain model whose behavior requires certain critical initial values that the user may edit or optionally return to factory defaults. You do not wish to duplicate these initial values throughout the code so you embed them in a single method.

Applicability: Use this pattern when the default values for a domain object's attributes may be referenced in several places: for example, in accessors for lazy initialization, in an initialize method, or in a method that resets to factory defaults. We have found this pattern is most useful for high-level global or project options. In certain low-level objects that must be instantiated and accessed quickly, this pattern may be a performance bottleneck.

Solution: Create a class method that answers a dictionary with one association per attribute, where the key is the symbol for the attribute's accessor and the value is the object that should be the default value of the attribute. Have all references to the attribute's default retrieve the value from this default attribute map.

Implementation: This should be a private method.

```
attributeDefaultMap
    "Answer a dictionary, the default attribute values for
     instances of this class."
    ^IdentityDictionary new
    at: #errorMessage put: String new;
        at: #flags put: 0;
        at: #label put: 'Element';
        yourself
```

Note: If subclasses add many attributes, allocating a larger initial dictionary can offer a substantial performance optimization.

Consequences: Building the Epitome map dynamically and accessing it attribute by attribute can be very expensive. The map could be built when code is loaded and cached in a class variable, otherwise it should be cached in a temporary variable when used by clients. A benefit of using this pattern, instead of embedding constant default values inside lazy-initializing accessors, is that the initial state of the object is available for review in one method rather than being spread out over many methods. Note that this pattern does not prevent the use of lazy-initialization, it simply moves the values elsewhere.

Related Patterns: An alternative to this pattern is to keep a constant-valued Prototype object available to the class at runtime and use it to seed the initial values of instances. Epitome is essentially a way to build and answer the Memento for that Prototype in code. The benefit of using Epitome is that you do not need to implement mechanisms to manage a Prototype object; the drawback of the pattern is slower performance.

ACTUATOR (INITIALIZING SETTER)

Problem: How do you convert a constant attribute of an object to one that can vary during the lifetime of the object?

Motivation: You have defined a domain model class that is initialized with a diameter. During initialization, the model must size other parameters based on the initialized diameter, like this:

```
initialize
    "Initialize instances to default values."
    diameter := 10.
    self sizeFittings.
```

Later, you decide to modify the domain model so that clients can change its diameter at runtime. To accomplish this you provide a setting accessor method and move the dependent sizing code from the domain model's initialization method to the new accessor:

```
diameter: aFloat
    "Set the diameter of this object to the passed floating
     point value and execute dependent actions."
    diameter := aFloat.
    self sizeFittings.
```

You designate the new accessor for public use by clients when they wish to change the diameter at runtime as well as modifying the domain model's own initialization method to use it:

```
initialize
    "Initialize instances to default values."
    self diameter: 10.
```

Applicability: This pattern is applicable when an object has been designed with some attribute or collaborator that is set at initialization and originally does not change during the life of the object. Special initialization code must run when the identity of that attribute or collaborator is known. Use the pattern when you wish to change the original design to allow clients to dynamically configure the attribute or collaborator during the lifetime of the object. Actuator is also a candidate for use whenever special initialization actions must be taken once the identities of an object's attributes or collaborators are known.

Solution: Create a setting accessor method for the attribute. Move dependent initialization code into the accessor immediately after the value is set. Ensure that the object itself, when created, uses this accessor for initializing the attribute and that clients use it for changing the attribute's value during the lifetime of the object.

Implementation: Move code from initialization and other methods into a new accessor method. (If the object was initially designed for the given attribute to be constant, some research may be required to find all the initialization code that is dependent on the attribute.) Note that in some cases (e.g., when event handlers have been established on a collaborator), it may also be necessary to write code in the accessor to perform finalization actions before the collaborator can be replaced.

Consequences: Application of this pattern may be beneficial even when attributes aren't expected to change at runtime because it associates dependent initialization logic more closely with the attribute it applies to. Actuator can reduce the size of complex initialize methods by moving their logic into separate accessors.

Related Patterns: Application of this pattern is similar to Template Method in that it turns an initialize method with much attribute-specific logic into a skeleton that delegates to a series of lower-level accessor methods. However, unlike Template Method, those lower-level methods are concrete and not usually intended for overriding.

Actuator is also related to Observer in that dependent code runs in response to some change in state. However, Observer is intended for loose coupling between two or more objects at runtime, whereas Actuator is for setting up at development time, quick responses to changes within a single object.

COMING UP

The next article of my three-part series considers two families of patterns: validation patterns for checking and protecting domain objects and informational patterns for managing status and validation messages. The third and final article will review a family of optimization patterns.

Reference

1. Gamma, E. et al. Design Patterns, Addison-Wesley, Reading, MA, 1994.

Darrow Kirkpatrick is VP of Research and Development at Haestad Methods, Inc., which specializes in numerical modeling for hydrology/hydraulics, and has pioneered using Smalltalk for shrink-wrapped Windows applications. Darrow enjoys hunting for patterns while leading a team of talented software engineers who have become experts at coaxing Smalltalk to perform in the real world. He can be contacted at 203.755.1666 (voice) or by email at 75166.525@compuserve.com.

THE BEST OF COMP.LANG.SMALLTALK

continued from page 20

you'll be happier in the long run if you can just hold off a little longer.

There are several reasons for this. First, time spent on optimization isn't being spent on those "meaningless" chores that are often more important to the success of the project. If testing and documentation are inadequate, most people won't notice or care how fast a particular list box updates. They'll have given up on the program before they ever got to that window.

That's not the worst of it. Premature optimization is usually in direct violation of the principle of postponing decisions. Optimization often involves thoughts like "if we restrict those to be integers in the range from 3 to 87, then we can make this a ByteArray and replace these dictionaries lookups with array accesses". The problem is that we've probably made our code less clear and we've greatly reduced its flexibility. It may have felt really good at the time but the other people involved in the project may not be entirely satisfied.

Of course this rule doesn't apply to all optimizations. Most programs will need some optimization sometime and this is particularly true in Smalltalk. As a very high-level language, Smalltalk makes it very easy to write very inefficient programs very quickly. A little bit of well-placed optimization can make the code enormously faster without harming the program.

There's also a large class of optimizations that I call "stupidity removal" that can be profitably done at just about any time. These include things like using the right kind of collection for the job and avoiding duplicated work. Their most important characteristic is that they should also result in improvements to the clarity and elegance of the code. Using better algorithms (as long as their details don't show through the layers of abstraction) can also fall into this category.

OTHER RULES TO LIVE BY

There are many other rules of life that can be extended to the OO design and programming domains. Here are a few more examples. Feel free to make up more and send them to me. Make posters out of them and put them up on your office wall. It'll make a nice counterpoint to those insipid posters about "Teamwork" and "Quality" that seem to be everywhere these days.

- Try not to care—Beginning Smalltalk programmers often have trouble because they think they need to understand all the details of how something works before they can use it. This means it takes quite a while before they can master Transcript show: 'Hello World'. One of the great leaps in OO is to be able to answer the question "How does this work?" with "I don't care".
- Just do it!—An excellent slogan for projects that are suffering from analysis paralysis, the inability to do anything but generate reports and diagrams for what they're eventually going to do.

continued on page 32