Tactical patterns for the real world: Validation and informational patterns

Darrow Kirkpatrick

PREVIOUSLY IN THIS series of articles on patterns for efficiently implementing and managing domain models we looked at a family of patterns for dealing with instantiation issues. Interior Decorator showed how to share a set of useful behaviors that may be selectively needed throughout a family of classes in a broad hierarchy. Epitome showed how to consolidate and share the default values for an object's attributes. Actuator showed how to convert a constant attribute of an object to one that can vary during the lifetime of the object.

In this issue we explore two additional families of patterns: the first dealing with validation issues—checking and protecting domain objects, and the second dealing with informational issues—managing status and validation messages.

VALIDATION PATTERNS

Safeguard (Delayed Validation)

Problem. Where do you put complex validation logic; and how do you prevent invalid domain objects from being used?

Motivation. You want to allow user editing of a slope object that represents the simple equation

$$slope = (elevation2 - elevation1) / length.$$

You wish to encapsulate in the slope class the domain validation rule that *length* may not equal zero, yet this means allowing some domain models to take on illegal values for *length* temporarily, so that it may be validated. You establish an overall domain model validation method, a Safeguard, to be checked before any attempt is made to calculate the model.

Applicability. Use this pattern when you need to validate domain objects to catch data-entry errors or other logical errors that would prevent correct calculation, and when the validation logic requires intimate knowledge of the domain. It is particularly appropriate for top-level validation of complex models with many interrelated parts.

Solution. Associate complex validation logic with domain classes by writing validation methods. Allow data that may be invalid into the domain, but guard calculations with a validation method.

Implementation. Write a public validation method called #isValid for use by clients, delegating to a private #validate method that may be overridden by domain subclasses. The public method should reset error flags as necessary before the specific validation is invoked, and answer the result of the validation:

```
isValid
    "Answer a Boolean, whether this object is currently valid."
    ^self
        clearErrorFlag;
        validate;
        isErrorFlagSet

validate
    "Test all aspects of this object for validity and if any
      fail set the error flag."
    length > 0 ifFalse: [
        self setErrorFlag ]
```

Consequences. This pattern encourages the programmer to keep validation code near the domain state that it protects, improving encapsulation. But to maintain validation logic in the domain means you must allow bad data into the domain temporarily, and validate it later. A benefit of applying this pattern is that expensive validation occurs only on demand, rather than whenever there is a change to the model (which can be slow), or when indicated by a modified flag (which can be hard to maintain). A drawback is that clients must remember to check for validity before using the model.

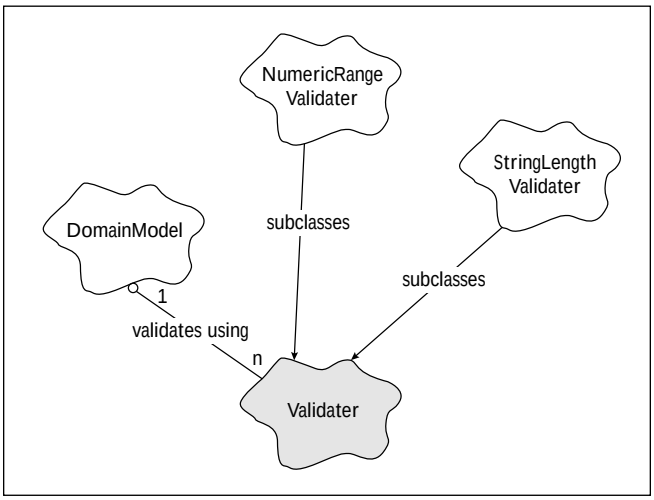

Figure 1. Validater class hierarchy.

Related Patterns. An alternative to this pattern is to use a Memento[1] to copy the domain object and make that Memento instead of the object available for editing. The Memento can then be validated before it is applied to the domain object. The drawback to this approach is the tendency of validation code to drift out of the domain into interface or broker classes.

See Verdict for more discussion of how the messages that result from validation may be managed. The #isValid method is an example of Template Method.

DEFLECTOR (DEFENSIVE SETTER)

Problem. How do you prevent attributes of certain classes from ever taking on illegal values?

Motivation. You maintain a global help level that user interface elements check to determine how much help and functionality they should reveal. Rather than require those clients to deal with illegal values, it is convenient to force the help level to remain within a certain range. To guarantee this range you implement a help level setter method that rejects illegal values, and you require all clients to use it.

```
helpLevel: anInteger
    "Set the help level for the system to the passed integer,
        if it is valid."
    (anInteger between: 1 and: 3 )
        ifTrue: [ helpLevel := anInteger].
    ^helpLevel
```

Applicability. Often in a running system an attribute can be set from many different sources, such as the user interface, client code, or an initialization file. Some of these sources may be unreliable, yet the attribute must never become invalid. Use Deflector when attributes of critical objects must never take on illegal values and there is no opportunity to perform interactive validation before the value takes effect. The client that is setting the attribute does not require that the operation succeed for the system to remain stable, and is not prepared to handle an exception.

Solution. Perform validation against the passed value in the setter method itself. If the value is invalid, simply do not set it. There is no exception or error return. The current value of the attribute is always returned.

Implementation. Move validation code to the lowest level in the object implementation. If extra performance or bypassing validation are sometimes required, consider implementing a second, low-level, "basic" accessor that performs no validation before setting the attribute.

Consequences. This pattern leads to robust but possibly obtuse behavior. Because there is silent validation of attributes, users will be protected from—but not notified of—error conditions. Because there is some overhead to perform validation in accessors, this pattern is not appropriate for performance-critical code.

Related Patterns. Deflector is an example of early validation; Safeguard is an example of late validation.

VALIDATER (CONFIGURABLE VALIDATION)

Problem. How do you provide default validation of domain attribute values, while allowing end users to modify default validation logic safely?

Motivation. You are designing a system that models the flow of water, requiring a specific gravity to characterize the water. In certain situations it may be possible for an expert to use the system to model the flow of other kinds of fluids. To protect novice users you implement a default validation to check values entered for specific gravity against the range of legal values for water. However you objectify that validation as a persistent, editable object so that expert users may configure the system to accept values for other fluids.

Applicability. Use the Validater pattern when you wish to implement domain validation rules that can be relaxed or adjusted by the end user. The user-configurable portion of the rules must be represented by values that can be edited in a running program. The pattern as described here is for field-level attribute validation, such as range-checking, that requires no external context to perform.

Solution. Create an abstract validater class with concrete subclasses embodying state and behavior for different validation strategies.

Implementation. One approach to managing validaters is to have domain classes maintain symbolic names to specify the type of validation each of their attributes should receive. These symbolic names are keys into a global or project-level dictionary of available validaters. Have the object that is responsible for accepting edited attribute values—perhaps an Adapter—pass those values to the associated validater object for approval before being committed.

It may be convenient to have validaters share the same

protocol as code blocks, so they can be used interchangeably. This way the same client code can perform validation with runtime-specified validaters, or with more complex logic specified at development time via a block.

Consequences. This pattern assumes a more complex interaction with the user. A view should be in control of the validation process so the user can be notified and given the opportunity to correct domain attributes that fail validation. Also, you may need to provide separate editors for each validater subclass that can be configured by the user. The subtleties of editing validation parameters may confuse some end users.

Related Patterns. Validaters are an example of the Strategy pattern. They objectify different algorithms for performing validation, and make them interchangeable. Often a Validater will be used by an Adapter that has responsibility for interfacing between an editor and domain model.

Reviewing the three validation patterns, Validater is another example of early validation, while Deflector is an example of the earliest possible validation, and Safeguard is an example of late validation. You might use Validater to support editing domain attributes from dialogs where field-level validation is required. You would use Deflector to protect attributes which are subject to change from any source, including other application code. And you would use Safeguard when performance is critical, or more context is needed to perform validation than is available from the attribute value alone.

INFORMATIONAL PATTERNS

Verdict (Visitor Message Token)

Problem. How do you manage the results of a complex and expensive validation across a series of domain objects so that their status or validity may be queried at a later time?

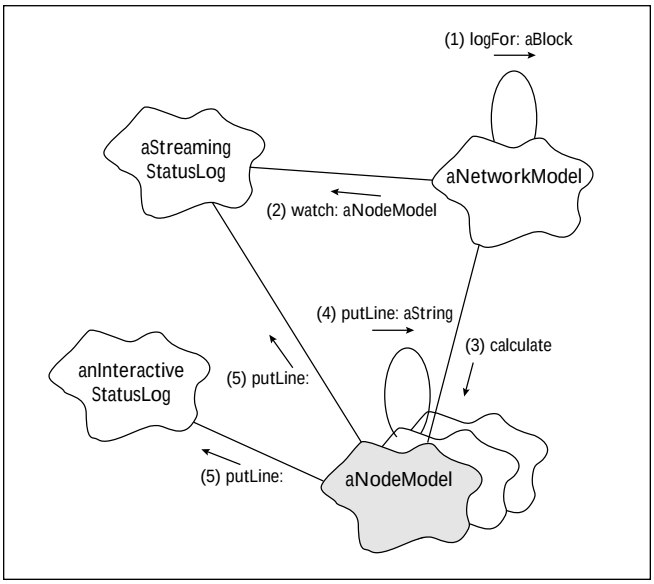

Figure 2. Ticker Tape object messages.

Motivation. You are designing the solution of a large network model. Each node in the network has associated constraints that must be met in order for the solution to be considered valid. If a node fails constraints the solution should proceed, but the failure must be stored so the user can be notified. To accomplish this you design the constraint checking algorithm to set a message symbol, a Verdict, into any node that fails constraints. Later, when the user inspects the node, the symbol is discovered and converted into a message for display.

Applicability. Use this pattern when it is not possible to interrupt validation, for example when validating domain objects requires context from an external object during an expensive traversal. And use it when the *results* of the validation must be stored for later use, for example when display of result strings may needto happen in a dialog during later editing of a domain object.

Usually it is an external, visiting object that leaves a token to be interpreted later. But rather than the visitor being another object, it might be a subclass that is providing validation behavior to add a Verdict into the object's message collection as validation proceeds.

Solution. Create a facility for storing within each domain object a collection of zero or more symbols that index into a global map of validation, warning, and error messages. When a validation is performed—and problems are found—add the appropriate symbols to the domain object's message collection. If there are no validation problems, remove the appropriate symbols from the object's collection, if they have been added during a prior validation.

Implementation. A more flexible solution for managing messages is to allow multiple categories by maintaining a dictionary in the domain object whose keys are types of messages (*constraint*, *warning*, and *error* for example), and whose values are sets of symbols representing specific messages. If a message dictionary is used, subclasses and clients can add their own message categories.

Consequences. This pattern moves validation behavior out of low-level domain objects into higher-level objects. Because of the threat to encapsulation this may not always be desirable, though it can be essential if additional context is required to perform the validation.

Applying this pattern throughout a domain hierarchy may result in wasted space if many objects don't need to maintain validation messages, especially if message category dictionaries are used. Consider applying the Interior Decorator pattern to save space. Note that the need to remove message tokens once objects are valid requires additional logic and can be prone to subtle bugs.

Related Patterns. A Verdict may be left by a Visitor performing validation. Or, subclasses may add Verdict messages by overriding portions of a validation implemented by a Template Method. A Safeguard method may use the *pres-*

ence of a Verdict message to indicate that the domain object is not valid.

TICKER TAPE (STATUS MESSAGE LOG)

Problem. How do you collect status information from a lengthy domain operation involving thousands of objects, none of which have visibility to the user interface?

Motivation. You are designing a complex network calculation. When standalone test suites are run during development, no status information is desired. However, when a user interface is present and calculations are occurring at runtime, feedback to the user is essential. Therefore you implement a publish and subscribe mechanism for domain model messaging. Domain models trigger messages as they calculate. Clients may subscribe to, collect, and present those messages if they wish.

This pattern moves validation behavior out of low-level domain objects into higher-level objects.

Applicability. Use this pattern when one or more domain objects must perform a lengthy or complex operation for which status information may not always be desirable, and the formatting of the status information is a function of individual domain objects.

This pattern is also useful for debugging and tracing, and any time that status information must be collected for filtering or presenting later. It is not appropriate when heavy formatting or graphics are required—such as for WYSIWYG reporting.

Solution. Create a simple low-level protocol in domain objects for outputting formatted strings. The protocol should simply trigger an event with the string as argument. Create a family of status log classes that can subscribe to these events and present the status information with various levels of formatting and interactivity.

Implementation. The fundamental methods required are:

Model >> #logFor: aBlock to establish a status log for the duration of aBlock. This method is implemented in the top level domain object, which is performing the lengthy operation. The method is responsible for instantiating the status log, traversing all the lower-level domain models so the log can subscribe to their events, evaluating the block (causing the lengthy operation to proceed), and then traversing again so the log can drop the models.

StatusLog >> #watch: aModel and StatusLog >> #drop: aModel in the status log object to subscribe to and cancel receiving the status events triggered by a domain model.

Model >> #putLine: aString to trigger the status event from the domain object. (Use a method of the same name to output the text in the status log.)

Consequences. Note that hooking up numerous domain object events to handlers in the status log object and releasing them afterwards may be expensive, but is relatively fast compared to the long operations for which this pattern is appropriate.

Related Patterns. Ticker Tape uses the Observer pattern to implement a publish and subscribe mechanism: the domain models are the subjects and the status logs are the observers.

COMING UP

The concluding article in this series presents a family of patterns for dealing with optimization issues—implementing domain models that must perform well even though they incorporate extra levels of indirection to be persistent or transient.

Reference

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

Darrow Kirkpatrick is Vice President 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 software engineers who have become experts at coaxing Smalltalk to perform in the real world. He can be contacted by phone at 203.755.1666 or by email at 75166.525@compuserve.com.