Tactical patterns for the real world: Optimization patterns

Darrow Kirkpatrick

THIS IS THE final article in a three-part series on patterns for efficiently implementing and managing domain models. The first article presented a family of instantiation patterns—patterns that aid in creating or initializing objects.

The second article presented a family of patterns for dealing with validation issues. Safeguard showed where to put complex validation logic; and how to prevent invalid domain objects from being used. Deflector showed how to prevent attributes of certain classes from ever taking on illegal values. And Validater showed how to provide default validation of domain attribute values, while allowing end users to modify default validation logic safely. The second article also presented a family of patterns for dealing with informational issues. Verdict showed how to 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. And Ticker Tape showed how to collect status information from a lengthy domain operation involving thousands of objects, none of which have visibility to the user interface.

This final article presents a family of patterns for dealing with optimization issues—handling domain models that must perform well while incorporating extra levels of indirection in order to be persistent or transient. So far, the patterns presented have been language-neutral. Although the following two optimization patterns require a specific Smalltalk dialect to implement as described, the principles they embody are generic.

OPTIMIZATION PATTERNS

Avatar (Soft Schema Evolution)

Problem: How do you efficiently implement a persistent object that must be able to add attributes without causing a file shape change, yet must be very fast to access?

Motivation: (Background: The Visual Smalltalk ObjectFiler is able to save and load collections that contain only primitive objects like strings and numbers without requiring a file shape change as the collections change in size. Thus by implementing a property dictionary inside a higher-level domain object you can add simple state to the domain object after it is designed without having to provide code for schema evolution. Unfortunately, because access to a dictionary is via hashed lookup, the access speed for these additional properties is several times slower than for attributes kept in instance variables.)

You are designing an object that holds various calculation parameters that must be accessed efficiently inside tight loops, and must be stored persistently within each project. It is likely that this object will add parameters over time as additional calculation constants are made available for user editing, yet you do not wish to force a file schema change each time a parameter is added. To solve this problem you create two classes with the same interface: one optimized for persistence and one optimized for access, and you design a mechanism to automatically convert between them as required. Avatar is the transient incarnation of a faster memory form of a flexible persistent object.

Applicability: Use this pattern when you need to implement a simple, persistent object with the following characteristics: it needs to be able to add and hold properties without explicitly mutating shape, yet access to the properties must be extremely fast, not paying the overhead of generalized property dictionary access.

Though the implementation of this pattern presented here is Smalltalk-specific, the general principle of mutating between a flexible persistent representation and a fast memory representation of an object should be applicable in any language that provides object persistence mechanisms.

Solution: Implement a pair of classes with the same interface. One uses a property dictionary to store attributes, the other uses instance variables. The property dictionary object is persistent and has an activation method that mutates it into the instance variable object when it is read from disk. The instance variable object is always used for the in-memory representation of the object. It has a surrogate creation method that answers the property dictionary version of itself for save operations.

Implementation: The Visual Smalltalk ObjectFiler adds two important methods to the Object class: fileInActivate: and fileOutSurrogate:, as hooks for transforming an object at load and dump time, respectively. To implement the Avatar pattern, the property dictionary object's fileInActivate: method instantiates the instance variable object; the instance variable object's fileOutSurrogate: method instantiates the property dictionary object. As new attributes are added to the object over time, lazy initialization in accessing methods allows new code to work for older files containing objects that lack those attributes. Note that it may be possible to share the interface of these two classes using inheritance: generally the instance variable class would subclass the property dictionary class—using its interface but not its dictionary.

Consequences: Application of this pattern results in two class implementations that must be maintained in parallel. This is only justified when profiling indicates a performance-critical situation.

Related Patterns: This pattern is related to Bridge[1] in that it provides multiple implementations for the same interface. However, where Bridge provides parallel interface and implementation hierarchies that can vary independently, Avatar simply provides two subclasses whose instances are swapped back and forth.

SPEEDWAY (FAST LIBRARY INTERFACE)

Problem: How do you minimize the cost of referencing a class within a demand-loaded library component?

Motivation: You maintain a library of mathematical classes for performing numerical methods. Because memory is precious and this library is needed only during calculations it is referenced indirectly and loaded on demand at the first reference to a mathematical class. The indirect reference consists of a symbolic class reference via the Smalltalk dictionary, plus a search through a collection of library interfaces to find the library containing the class. This indirection is relatively slow compared to numerically intensive code, especially when it appears inside iterative calculations. To optimize the reference you encapsulate and cache it inside an object that provides a speedy gateway to the library.

Applicability: Use this pattern whenever you indirectly reference a component from within performance-critical code, and the indirection is prohibitively expensive.

Solution: Write a class to act as a fast gateway to the library. The class implements one instance variable for each public class in the library. In performance-critical code, instead of embedding indirect library references, send a message to the Speedway object. The corresponding

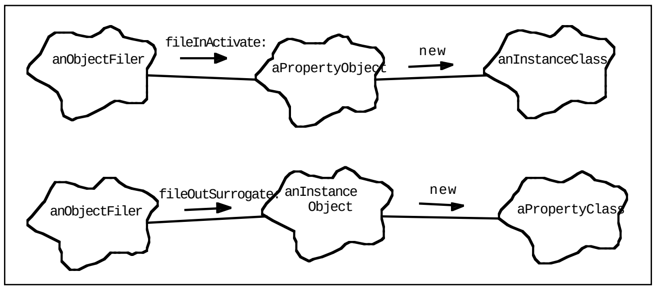

Figure 1. Avatar objects transforming from and to disk.

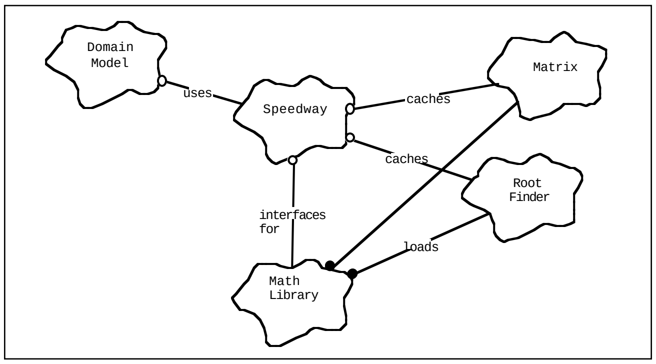

Figure 2. Class hierarchy of math library speedway and client.

method in the Speedway initializes its appropriate instance variable to point to the actual class, first triggering a library load if necessary. Thus in future references the indirection previously in the code is optimized to a single message send which requires a single lazy initialization test.

Implementation: The Speedway is typically a Singleton object managed by a parent class of the domain model, so it can easily be accessed by any domain object. When preparing a runtime application you need to flush the cached library references to avoid statically binding the library component to the image.

Rather than lazy initializing a single variable at the first library reference it may be desirable to populate the entire Speedway. This may provide a slight performance enhancement and allow consolidating some initialization code.

Consequences: This pattern requires maintaining an additional class. Since the library indirection described here is not slow relative to most code, this technique is justified for performance reasons only in critical code. However Speedway has another benefit—it objectifies and documents the public interface to a library subsystem, as far as that interface is limited to symbolic class references.

Related Patterns: Speedway is similar to Facade in that it provides a public interface to a subsystem. But the intent is different: Speedway provides an optimization whereas Facade usually provides some additional thin layer of behavior to make high-level use of the subsystem easier. Note that it may be useful to implement a Facade that incorporates Speedway's caching behavior.

Another approach to solving the library interface prob-

continued on page 34

OPTIMIZATION PATTERNS *continued from page 19*

lem is to implement class Proxy objects. When a class Proxy is referenced it loads the necessary library and then *becomes* the real class object. This approach requires a more sophisticated and intrusive implementation, but results in complete transparency for clients who use library classes.

CONCLUSION

This series documented families of implementation patterns used by our development group for creating domain models in engineering applications. In presenting a particular "handbook" of tactical patterns for a specific domain, these articles are examples of a management or documentation pattern. The application of this higher-level management pattern results in a pattern language—a concise narrative of the principles pervading a body of code.

Pattern languages are useful at different levels of abstraction in different domains by different teams. Through pattern languages Alexander, Beck, Gamma, Helm, Johnson, Vlissides, and others have given us a powerful tool for communicating the craft of software engineering to each other. I hope these articles will encourage other engineers to discover and publish handbooks of domain-specific patterns that have helped them to deal with the demands of creating real-world software.

Reference

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