The Dangers of Storing Objects

Juanita J. Ewing
Instantiations, Inc.

Smalltalk systems now include the ability to correctly write representations of composite objects to disk. Early Smalltalk systems could not deal with objects containing circular references, so the capability of storing objects was not widely used. Now that many kinds of objects can be written, other issues have arisen. When is it appropriate to use this mechanism? Is this a good way to provide long-term storage of objects? Can this capability be overused or misused?

Object storage was first implemented for Tektronix Smalltalk by Steve Vegdahl[1]. In Smalltalk/V this capability is called Object Filing. In Objectworks\Smalltalk, this capability is implemented by BOSS (Binary Object Streaming Service).

In all these implementations, an encoded representation of an object is written to a file. The representation of the object consists of structural information required to recreate the object from the data in the file. Objects recreated from the data on disk are not the same as the original object. These systems do not maintain object identity across read/write operations and are therefore not persistent object systems.

What is written to disk?

When the representation of an object is written to disk, it must include all the data necessary to recreate the object. The class name is written to designate the class of object to be recreated. Each component of the object, numbered slots and instance variables, is written. If the component is a reference to another object, that object is also written.

Each implementation has different restrictions on precisely which objects are written. The values of global variables such as Transcript are not written. Instead, a reference to the variable's name is stored and when the object is recreated it's reference is bound to the current value of the identifier.

The representation of an object in these systems is the data from the private internal implementation of the object. The public interface to an object is not used to recreate the object. Instead, private low level methods are used.

Why do developers write/recreate objects?

The big advantage of object storage systems is that they permit a Smalltalk developer to externalize objects without designing a special file format, or writing input/output methods. Developers might use object storage systems to “transfer” objects from one image to another. Other members of a development team might need an object that is difficult or time consuming to recreate. A prototype might have objects built by hand instead of programatically, or objects might be created from a data feed.

Developers sometimes use this ability to “save” objects. They want the objects to exist longer than an image. Another use is to reduce the size of an application image by building an external “database” of stored objects. Only the objects that are actual being used need to be loaded.

Sophisticated Use

An application I helped develop had visual components that were used off-screen to generate a composite graphic. The composite graphic was stored in an instance variable, but we didn’t want it saved when we wrote our objects to disk. It was large, and took more time to read from disk than to recreate. We needed a way to control which components of an object are written.

Both Object Filer and BOSS have a mechanism to customize what is written on a per class basis.

- With Object Filer, you implement a method with the selector fileOutSurrogate: that returns a surrogate object to be written to disk in place of the receiver. The surrogate can be a copy of the original object with modified instance variables.

- With BOSS, you implement a method with the selector representBinaryOn: that uses other BOSS methods to write the representation of the object to a stream.

Sophisticated use of these systems requires developers to write special methods that modify the written representation of the object, usually by changing the private instance state of the stored object. The manner in which these systems are customized is an indication of the limitations of these systems: they manage the storage of an object at the structural level.

Dangers

Class definitions are volatile. Instance variables, class variables and pool dictionaries can be added or deleted. Once a change is made to the private implementation of an object, such as adding an instance variable, the written representation on disk is no longer accurate. Because the representation consists of private implementation data, the public interface of that object is not used to recreate the object.

Problems arise from

- renaming a class

- changing representations

- restructuring a class
- refactoring a hierarchy

Most of these systems have mechanisms to handle simple variations in an object's definition. In the case of added and deleted instance variables, Object Filer brings up a graphical interface that interactively lets you map instance variables on disk to the instance variables in your image. This mechanism is particularly useful when instance variables have been renamed. Object Filer also has a mechanism to support classes that have been renamed.

Changing Representations

Suppose a composite object consists of a deeply nested tree structure. When this object is written to disk, a representation of it and all of its composite objects is written. Later, the developers add a cache of recently accessed leaf node to the object. This cache, an instance of OrderedCollection, is stored in an additional instance variable. The representation of the object on disk does not specify a value for the cache instance variable. When the object is recreated, it has a nil value for the cache.

The methods in the composite object must be designed specially to accommodate a value of nil for the cache. Accessing methods for the cache must check for nil instead of assuming an instance of OrderedCollection and create an instance of OrderedCollection if necessary. The developers save some more composite objects to disk.

Later, the cache is changed to be an instance of Dictionary. Accessing methods are again modified to check not only for nil but for instances of OrderedCollection, and the cache is modified to be an instance of Dictionary if necessary. More composite objects are saved to disk.

What is the situation now? The developers now have representations of composite objects with the following variations:

- no cache instance variable
- cache instance variable bound to instance of OrderedCollection
- cache instance variable bound to an instance of Dictionary

In this example of changing representations, what you really have is a mess, with code for backwards compatibility in every relevant accessing method. The situation is even worse if you don't use accessing methods and instead directly reference instance variables. You end up with code for backwards compatibility in every method that references that instance variable.

The series of modifications I've described is very typical. The original definition of a class is rarely right. Definitions are changed to accommodate optimizations, as described above. Functional extensions also require modifications. For example, an ellipse class describes a elliptical element with a border width and color. It has instance variables to store the attributes width and color. Later, the developers add functionality for filling the inside area of the ellipse. The class definition is modified: another instance variable stores the fill color.

Refactoring and Restructuring

The most devastating kind of change is not addition or deletion of instance variables. It is the refactoring and restructuring of classes into sets of classes, or the combination of several classes into a single class. As developers create an application, the design evolves: responsibilities are redistributed and new classes are created.

Let's look at a simple example of restructuring: Suppose your application records information about people, including their name which is an instance of String. Later you decide a single string is not a good representation and you need to model the first name and last name as two separate entities. If you have stored objects with the name represented by an instance of String, your must make extensions to the object storage system to

- read the name,
- detect the class,
- and potentially parse the string to model both the first and last names separately.

An example of refactoring, discussed in several publications lately, is from the Objectworks\Smalltalk user interface library. The class View has been refactored into a number of smaller classes, each with less functionality. Is it possible to take a view that has been stored on disk and recreate it in terms of the new classes? No doubt it would be easier and less time consuming to rewrite the code used to create the view than to recreate its equivalent from the object representation on disk.

Alternative

Why would it be easier to rewrite code to make a view? Because the rewritten code uses the public interface to objects. Writing objects to disk using the private implementation data is okay for a quick transfer, but not a good idea for any long term needs.

Object storage systems are very handy for short-term use, but because of the dynamic nature of classes, they are unsuitable for long-term use. These systems encode the structural implementation rather than the semantics of the information.

Every major Smalltalk application I know of that used an object storage system for long-term storage ultimately had to be modified to use a less implementation dependent storage format. A good format captures the data without directly specifying objects and the values of their instance variables. Instead it captures the relevant data in an object independent format by storing only semantic data. Methods that read the data instantiate new objects by sending public messages.

References

[1] Vegdahl, Steven R. "Moving Structures between Smalltalk Images." Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications. Portland, Oregon. September, 1986, pp. 466-471