Should Classes Have Owners?

Juanita J. Ewing
Instantiations, Inc.

Derived from Smalltalk Report

As Smalltalk engineering projects grow larger, the need for reusable code increases. Developers need to build larger applications even faster. The easiest way to increase the capabilities and scope of an application is to reuse more classes. Large applications require teams of Smalltalk programmers to glue these reusable classes together, and write some application specific code too.

What is a reusable class? Classes are reusable in two ways: as a client making instances or as the basis for new subclasses. The characteristics of these two kinds of reusable classes are different. For client use you want a fleshed-out and general class. For subclassing you want a minimal and flexible class. Beyond these characteristics, how do you tell if a class is reusable? To paraphrase Ralph Johnson, a class isn't reusable until proven reusable. That means it has been used in more than one application.

It takes extra time and effort to write classes that are reusable. This extra effort is a separate programming activity. Developers who are caught up in deadlines for delivering an application often don't have the time necessary to flesh out and polish their classes. For example, developers will initially create a single class that should be refactored into a combination of an abstract class and a concrete class. The concrete class can be reused by making instances of it, and the abstract class can be reused by making new subclasses derived from it. The reusability of classes written with the goal of multiple uses is much greater than those written for specific roles in an applications.

In conjunction with supporting teams of developers, some Smalltalk environments actively promote the creation of reusable classes. One of the goals of these environments is to separate application engineering from the creation of reusable units of code.

Is class ownership a good basis for promoting the creation of reusable classes? Suppose each class is owned by a single developer. The theory is that an owner feels responsible for and will take the extra effort to make a class truly reusable. The creation of reusable classes is important to the entire organization as well as the developers. Programming environment capability by itself is not enough. To back up this capability in the programming environment, the developers' organization must reward the production of reusable code. Responsibility and ownership are established management techniques for motivating employees. It's become common practice in manufacturing environments to give employees more responsibility and have them provide input about the manufacturing process. Employees don't just screw on lug nuts anymore.

Let's assume the owner of a class is rewarded for producing a reusable class. What if another developer finds a bug in that class, or thinks of a useful extension? In a system with class ownership, the owner writes the code to fix the bug or writes a new method. He is the one who is motivated to make the class more reusable.

Who is best qualified to fix the bug or write the new method? In the case of the bug, the best qualified person may be the developer who detected the symptom of a problem and isolated the error. After the detective work, fixing the bug may be simple. And, sometimes it is difficult to reproduce a bug. In the case of the new method, maybe the person who thought of the extension knows best how to implement it. Maybe in both cases the owner and the person suggesting the change need to work together to come up with the best solution. The best qualified person depends on the situation. Flexibility in the programming environment is critical.

Systems with class ownership are not flexible. Even the motivational aspects are wrong for flexibility. What is the motivation for developers who are not owners?

Do classes exist in isolation? When a class is part of an application, it interacts, or collaborates, with other classes. Sometimes the collaboration is part of a framework. For example, a view and a controller collaborate as part of the MVC framework. An instance of view is never used alone. It is always paired with a controller. Because of the relationship between these two classes, coupled with the fact that modifications in one class will probably require corresponding modifications in the other class, there is a strong reason for the same developer to own both of these classes. It makes sense that any related classes should also be owned by the same developer. Evidently all parts of a framework should be owned by the same developer.

Continuing this example, what about the view's relationship with its model? Some views have a close connection with their models. This argues that the model should be owned by the same developer that the view and controller are owned by. And yet, in different applications the same view may collaborate with different models. Are all of those models owned by the developer that owns the view? Class ownership doesn't take into account the flexibility required by multiple applications.

A subclass is closely related to its superclass. If the behavior of a class changes, there may be ramifications in the subclass, requiring corresponding changes in subclasses. This implies that the same developer should own classes that are hierarchically related. Obviously, if one developer owns the entire image, we aren't talking about teams of Smalltalk programmers anymore.

If classes have owners and related classes are owned by the same developer to improve the efficiency of the team, how do you devise a reasonable partitioning if the ownership is restricted to a single developer per class? The answer is, you can't. The goal of grouping related classes conflicts with the goal of distributing classes to individual owners.

Do multiple developers affect the quality of classes? Close collaboration between developers is important in the production of reusable classes. People who are working together tend to be more creative. Multiple perspectives increase the likelihood of more general abstractions. Multiple developers are an advantage. The result of multiple developers is classes that are well fleshed out and suitable for client use, and classes that are general abstractions suitable for subclassing.

Should classes be accessable to multiple developers? The programming environment needs to promote developers working together. One way to do this is to make classes accessable to multiple developers. That way, each developer could make changes when most appropriate. If you have one owner, what do you do when that owner goes on vacation? What if the owner is ill at a critical time in the project? The programming environment should make it easy to implement contingency plans to keep a project going.

Since a reusable class is produced by a team of people, the entire team should be rewarded. Team programming environments usually have author designations for accountability. Outstanding efforts will continue to be noticed in these environments because of accountability features.

How does the programming environment keep things from falling between the cracks? How do you ensure that the entire class hangs together? You don't want to end up with classes that are a hodgepodge of functionality. Some automatic checks could be installed to produce warnings if, for example, a method contains no references to self or instance variables.

Most of the consistency checks for a class cannot be automated at this time. A human still needs to browse and understand a class to see if it follows basic design principles. In a cooperative team, this responsibility can be shared. Peer reviews, or more formal code reviews, are an essential part of team efforts.

The programming environment should be able to restrict the set of developers for a class to avoid unauthorized modifications. Many operating systems offer these kinds of limitations. A team programming environment could be even more selective. Also, it is a good idea to place at least one experienced person with a group of inexperienced people. People who have good rapport generally program together well.

A programming environment for teams of Smalltalk developers should promote the creation of reusable classes by rewarding all developers. The advantage of multiple developers is to allow multiple perspectives and therefore create more general classes. Another benefit of multiple developers is more apparent in the final stage of the software life cycle. Classes that are developed by multiple programmers are therefore understood by multiple programmers. It is easier for the organization to maintain classes because more than one person has the knowledge and understanding required for the job.