Chapter

34

ENVY

When organizations start developing in Smalltalk, they often use the fileIn and fileOut mechanism to manage changes to classes and methods. After a while, especially if the number of people working on an application increases, most organizations look for a way to better manage source code and configurations. The dominant player in this field for Smalltalk is Object Technology International, Inc (OTI), with their ENVY®/Developer product. The part of ENVY/Developer that manages source code and configurations is called ENVY®/Manager, and this is what most developers are referring to when they talk about ENVY. This chapter provides an overview of ENVY/Manager and perhaps gives a few pointers to those already using it.

ENVY/Manager helps you manage code development. In brief, it provides a source code repository that stores and tracks all changes to methods and classes. It allows developers to modify and test their code in isolation from each other, possibly even working on parallel versions of the same code. Once developers are satisfied that their code works, they can release it for general use. Other developers now have the choice of loading the new code into their images or continuing with the older version until they are ready to integrate. Between the extremes of isolation and integrating all the code released by a developer, ENVY/Manager also provides a way for developers to view or load individual classes or methods that another developer is working on. At the project level, ENVY/Manager keeps track of product releases. By creating a configuration map that specifies the versions of the classes that comprise a release, you can easily recreate any release of your application. We'll look at each of these facets in the following sections.

Code sharing

ENVY/Manager keeps the source code in a central repository called a library, so all developers have access to the same code (it actually keeps both source code and byte-compiled code for faster access). ENVY/Manager is conceptually a client-server application, where developers work in client images which communicate with the library server. Thus, whenever anyone makes a change to a class or a method, all other developers have access to that change should they want to view it or load it into their image (subject to security, which we'll talk about shortly).

ENVY/Manager automatically keeps track of every revision to every class and method rather than using the more common manual check-out, check-in mechanism used by source code managers such as RCS and SCCS. The manual check-out, check-in mechanism works well with large source files where work can be fairly well compartmentalized and source files checked out for relatively long periods of time. However, the Smalltalk object-oriented paradigm means that changes are likely to affect many cooperating classes, and since methods are generally short, a developer is likely to make small changes to many methods. In this world, tracking all changes is a lot less intrusive than a manual check-out and check-in scheme.

Because all the versions of the code are stored, it becomes a very simple matter to go back to a previous version if necessary. Even if methods or classes are deleted from the local image, they still exist in the repository and can easily be found by looking for *available* methods or classes in the appropriate browser.

ENVY/Manager makes it very easy for developers to work on code in parallel. For example, one developer might be working on a class for the next release of the product, while another developer is making bug fixes to support a previous release, and another is trying out some optimization techniques. All three developers can work on their changes without affecting each other, then when it comes time to integrate the changes, ENVY/Manager provides tools to show the differences between versions and to load alternative versions.

Applications and subapplications

In ENVY/Manager, everything is done within the context of an *application*, which is simply a collection or grouping of classes. Thus, within applications you have many classes, all pertaining to the application. To further organize the classes, you can create subapplications within the application. For example, a common way to organize an application (let's call it EmployeeApp), is to have three subapplications, one for the domain model classes, one for the user interface classes, and one for the data access classes. So we might call the subapplications EmployeeDomainApp, EmployeeUIApp, and EmployeeDataAccessApp. Subapplications are also commonly used to break out platform specific code because when an application is loaded it can use logic to determine which subapplications to load. (In what follows, we'll use the term application to mean either an application or subapplication, unless explicitly stated otherwise.)

When an application is created, it contains exactly one class, which has the same name as the application. This class serves several purposes. It starts off being a class template which developers can use to create new classes. It also provides an opportunity to write class side methods that will be invoked on pre-defined occasions (related to loading or unloading the application and starting and exiting the image). The most useful is the `loaded` method. Because source files are not filed in, class side `initialize` methods are not automatically invoked. Instead, you have to invoke them explicitly in the `loaded` method which is executed when the application is loaded into the image. (Another, less known method is `addToSystemMenu`, which allows you to easily extend the ENVY/Manager menu.)

In ENVY/Manager, every class is *defined* in a single application. However, you can *extend* a class in another application, which gives you the opportunity to add application specific behavior to that class (note, however, that you can't *modify* a base method in an extension). For example, suppose you are working on two products that share some but not all code. ProductOne needs a new method `foo` added to Object. You might create a new application called ProductOneSystemExtensions, then extend Object by adding `foo` to Object in this application.

Version Control

Applications and classes can exist in one of two basic states: as *versions*, which cannot be modified, or as *editions* which can be modified. You can tell the state of an application because versions display with a version name and editions display with a timestamp between parentheses.

ENVY/Manager allows developers to modify any method of any class (subject to access control). To modify any component, its container must be modifiable — ie, must be an edition. So, while methods are always editions, to modify a method, the class has to be an edition. Similarly, to modify a class, the containing application or subapplication must be an edition. Fortunately, ENVY/Manager makes this all very easy. The developer simply has to modify and accept a method and the enclosing class will automatically be made an edition (if it's not already one). If the application containing the class is not already an edition, ENVY/Manager will also ask the developer if he wants to create a *scratch edition* of the application when he accepts the method (we'll talk about the difference between an edition and a scratch edition later).

Visibility

All changes are recorded in the library — every change to every method. However, changes are not automatically propagated to other developers' images, so developers can work in isolation, unaffected by other people's work. However, if a developer chooses, he can view or load other people's changes. In a normal development environment a developer will load changes for two reasons: to integrate his code with that of others, and to load in a class or method to get an immediate fix from another developer. Lets look at the visibility of the different types of components, because the rules for viewing and loading them are different.

Methods

A developer can browse all editions of an individual method and load any edition into his image (remember that all methods exist as editions in the library and every time you accept a method a new edition is recorded).

Classes

A developer can browse all editions and versions of a class and can load any version of the class. However, editions of a class can be loaded only by the person who created the edition.

Applications

A developer can browse all editions and versions of an application or subapplication. He can also load any edition or version. But which classes are loaded when he loads another edition or version of the application? Only those classes that have been *released* to the application. By definition, releasing a class makes it available to the application and denotes that is should be loaded when the application is loaded. A class must be versioned — made immutable — before it can be released to an application. Similarly, if an application consists of several subapplications, loading the application will load the released versions or editions of the subapplications.

Products and subsystems

A developer will often want to load in the current baseline of the product under development, or of a subsystem. Loading in the current baseline gives the developer a chance to test his changes with all the released changes. Just as an application is a collection of classes that have meaning together, a *configuration map* (often known as a config map) is a collection of applications that have meaning together. By loading in a configuration

map, the developer loads in all the application versions or editions that have been specified in the configuration map. The order of loading of applications is controlled by specifying the *prerequisites* of each application.

Configuration maps provide a one-step way to load a specified set of applications. The main reasons to manage configurations are: to load the latest development build, to load end-user builds, and to give the ability to recreate releases of the product should it be necessary to track down a problem in a previous release. Typically an organization would have several different configuration maps: one for loading the latest development build, one for each major component or subsystem, and one for creating an end-user image.

Scratch Editions

Scratch editions of an application are similar to editions but have the quality that they are automatically created and are therefore unintrusive. However, you cannot release classes into a scratch edition of an application, and scratch editions cannot be browsed or loaded by other developers. Thus, if you exit the image without saving it, you will not be able to load the scratch edition you were working on and will have to individually load all the appropriate editions or versions of the classes you modified. Most developers let ENVY/Manager create a scratch edition for them, then convert it to a "regular" edition at a convenient stopping point. Besides being a convenience, scratch editions are useful in their own right for adding debug code that will later be removed (such as `self halt`). Since you are only adding debug code and then removing it, you don't really want an edition of the application to be permanently recorded in the library (although all the method changes are recorded). Scratch editions display with the version name surrounded by double angle brackets.

Management of Applications and Classes

ENVY/Manager works on the premise that applications have *managers* and classes have *owners* (the concepts are the same; just the terminology differs). Each application has a manager who is responsible for understanding the application and the relationships between the classes in the application. Each class has an owner who is responsible for the integrity and consistency of the class. The idea is that a class is likely to be better encapsulated, with a more consistent public interface, if one person is responsible for it. Having one person responsible for the class provides both accountability and continuity of perspective.

Classes have to be versioned to make them available to other developers, and have to be versioned and released to make them available to their containing applications. Only the class owner can release a class to its containing application. Similarly, subapplications have to be released to make them available to their containing application. Only the subapplication manager can release a subapplication. Applications are always available to load and hence always released.

However, while only owners and managers can release components, any developer can make changes to classes and methods. If a developer realizes he needs to modify some code in a class he doesn't own, he simply create a new edition of the class, makes the change, and keeps working. When he is done with his changes he will typically create a new version of the class. Since he can't release the new version of the class, he tells the owner of the class about the new version and the class owner browses it. The owner can release the new version of the class, or he can modify it or rewrite it to maintain the internal class consistency, then version it and release the new version.

easier for the class owner, they might name the version with their name and the date. That's as far as they can go since neither is the owner of the class. Now they tell the owner of the class the ID of the version they created.

The owner of the class can browse these new versions and can also browse differences between his version and either of the new versions, or between the two new versions. The owner can then choose how to integrate the new changes, whether by loading in entire methods, by cutting and pasting, or by rewriting. Once he has created a new edition and incorporated the changes, the owner then versions and releases the class, then asks the application manager to version the application.

Code reviews

ENVY/Manager makes it possible to do on-line code reviews. The general process is that once a developer is ready to have his code reviewed, he versions all the classes and applications then informs the reviewers that the code is ready for review. The reviewers then load in the code and review it in their images. They may add comments, rewrite methods, or even restructure methods within classes. Once done with their review, they inform the original developer, who looks at the changes and decides which changes to integrate. This process also makes it easier to stagger the reviews. Rather than have three developers simultaneously review the code and make changes, a better option is to have one person review the code, integrate any proposed changes, have the next developer review the results of the integration, integrate his proposed changes, then have the last developer review this integration.

In describing the mechanics of doing on-line code reviews, I am assuming that you are using the security provided by ENVY/Manager and that developers cannot simply become other users. I am also assuming that anyone asked to review code is a member of that application group.

The code author versions and releases all his classes and the application managers version the containing applications. The application managers then create new editions of the applications and re-version them with a name such as "For Review, 96/03/20". There are now versions of the applications for the reviewer, and editions of the applications which can be used to continue development or can be re-versioned. Next, the application managers change the application manager of the "For Review" version of each application. This is a key point: each edition or version of an application can have a different manager and different class owners (the default is to keep the same manager and owners as the previous version). Finally, the code author creates a new configuration map of the applications that will be reviewed, again naming it something like "For Review, 96/03/20", and changes the manager of the configuration map to be the reviewer.

The author then tells the reviewer that the code is ready for review and gives him the name of the configuration map. The reviewer loads the map and starts browsing all the code, making code changes where appropriate. Once the reviewer is done, he versions the classes he has changed, giving them a name such as "Reviewed by Bill, 96/03/24". At this point he can't release the classes since he is not the owner, so he simply makes himself the class owner and releases the class. (He can do this as he has already been made the manager of the review version of the application. It seems a little confusing, but by becoming the owner of the class in this edition of the application, he is *not* changing the ownership of the class in the main-line code.) The reviewer then versions the application with a similar name. Lastly, he creates a new version of the configuration map with a similar name.

The author then looks at the new configuration map to see which applications have been changed. He does not load the configuration map; instead he goes to each application that has been modified and browses changes.

Depending on how much work he has done in the meantime, he will either browse changes between the "For Review" and "Reviewed" versions of the application or between the current and "Reviewed" versions. If the application has subapplications, he will have to also browse changes for each subapplication. As he looks at the changes, he can load in the alternative code if he chooses. Figure 34-2 illustrates the process of reviewing a single application.

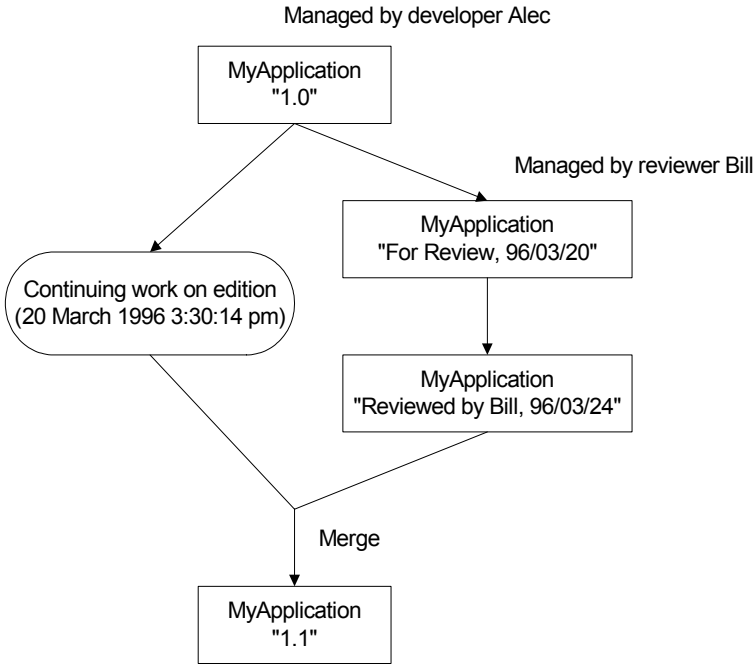

Figure 34-2.
The code review process.

For additional information on using ENVY/Manager to manage code reviews, see *Implementing Peer Code Reviews in Smalltalk*, by S. Sridhar, in the July/August 1992 issue of The Smalltalk Report. Note, however, that this article describes a process where there is little security and developers can simply become another user to accomplish the described tasks.

ENVY/Developer 3.01 enhancements

Most of the above describes the 1.43 release of ENVY/Manager. Release 3.01 provides some additional features and I've included information from the release notes here.

- Nested configuration maps.
- Browse changes between two configuration maps.
- Browse changes between application and all its subapplications.
- Export differences for configuration maps and applications.
- A more flexible component ownership model for releasing classes into applications and applications into configuration maps.
- Better support for distributed development teams.
- Version control for C and other non-Smalltalk languages. This includes using the browsers for non-Smalltalk development, using make files, and being able to version the files created during the build.

- Version control for other files such as bitmaps and help files.

Additional Information

OTI can be reached in the following ways:

Object Technology International, Inc
2670 Queensview Drive
Ottawa, Canada K2B 8K1
Voice: (613) 820-1200
Fax: (613) 820-1202

Object Technology, Inc
301 East Bethany Home Road
Suite A-100
Phoenix, AZ 85012, USA
Voice: (602) 222-9519
Fax: (602) 222-8503
e-mail: info@oti.com
Web page: http://www.oti.com

For a product review of ENVY/Manager version 1.43, see the review by Jan Steinman and Barbara Yates of Bytesmiths in the October 1992 issue of the Smalltalk Report. The review, with some updated information, can also be found at the web address:

http://www.bytesmiths.com/pubs/9209Envy.html.