Orwell
A Configuration Management System
For Team Programming

Dave Thomas and Kent Johnson

School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6

Abstract: In this paper, we describe the design and implementation of Orwell, a configuration management tool for multiperson Smalltalk projects. Although the system described has been implemented for Smalltalk, the design is applicable to other languages such as C++, Objective-C or ADA. Smalltalk is well recognized as a productive programming environment for an individual programmer, but its lack of team support is currently a major obstacle in using Smalltalk for a large software project. To support multiperson Smalltalk programming, Orwell provides both source and object code sharing as well as version control on a network of personal workstations. Class ownership is used as the primary means for dividing work among programmers during the lifecycle of a project. Orwell also supports groups of programmers not physically connected to a common file server. We describe our implementation which preserves the productive exploratory environment of Smalltalk. Seamless integration and performance are essential for Orwell to be accepted and used by Smalltalk programmers.

The research was supported by DREO (Defense Research Establishment Ottawa).

1. Introduction

We are developing a family of tools which support object oriented programming for embedded systems [Thomas 87]. Embedded applications are developed by large teams of programmers and are subject to strict configuration management (CM). In this paper we describe Orwell, a CM tool which supports team programming in Smalltalk. Orwell is the first system specifically designed to support the software engineering requirements of CM, while retaining the personal productivity of the incremental Smalltalk environment. It provides the ability to manage the source and object code of both classes and applications. Managing object code reduces the time to perform system integration and to package an application release.

1.1 The OOP CM Problem

Numerous tools exist for configuration management such as [AT&T 78] and RCS [Tichy 85], however none of these tools have been designed to support object oriented programming and its associated class library. Indeed support for package libraries has been identified as one of the major areas to be addressed by the ADA APSE. Current CM tools are module oriented, reflecting the unit of work in existing languages. In an OOL, the unit of compilation is the method, which is typically much smaller than a module. While it is certainly feasible to place every method and class definition in a separate file, this wastes disk space and leads to an unmanageably large set of files. The major difference with object oriented CM is the complex set of dependencies imposed by the inheritance of both state and behaviour. This problem has been widely experienced by users of Objective-C [Schmucker 88]. These dependencies cannot be

maintained manually and existing tools such as make file builders are unsuitable. It is not unusual for large projects to recompile and relink all of their code to ensure a proper application build. A common practice is to maintain only the source code for the system due to the difficulty of maintaining object code in a CM system. Many companies have a computer system dedicated to system builds. While it might be possible to consider such systems for Objective-C or C++, they are clearly inappropriate for an incremental environment such as Smalltalk.

1.2 Smalltalk meets Software Engineering

Introducing CM into any programming project is often seen as an unnecessary "big brother" overhead which will further reduce the productivity of the team. For many Smalltalk programmers an RCS style environment is the antithesis of exploratory programming and productivity. Smalltalk already supports the notion of change control and tracking via its change log and image files. It is the incremental nature of the Smalltalk environment which makes it a challenge to build a CM tool acceptable to both managers and programmers. Smalltalk is well recognized as a productive environment for an individual programmer; however, there are currently no integrated systems which support multiperson Smalltalk programming. The implementors of Smalltalk-80 recognized this problem and introduced the notion of projects as a way for one or more programmers to work on different projects while working on the same system/image. Yet projects do not address the needs of teams of programmers distributed over a network of workstations.

In Smalltalk, the *source* code and *changes* are kept in separate files while the persistent objects and the compiled code are kept in the *image* file. This leads to three monolithic files, in particular the image, which cannot be shared by members of a team. At present the only way to perform group programming is to give each programmer a copy of some common image file and then export and inport (fileIn, fileOut) source code between the team members. Recompiling the source is a tedious process, given that the current ST compilers were designed for incremental rather than batch compilation. The distribution and management of source and its associated dependencies is a time consuming and error prone process. A program librarian must carefully assemble the fileIns and/or change logs into a new version of the image. It is also very difficult to manage for a group of more than 6 - 10 programmers!

The lack of CM facilities combined with the cavalier style of many Smalltalk programmers is often cited as a major reason why Smalltalk cannot be used as a serious development environment. If Smalltalk is to be used in the large projects for which it holds much promise, a CM tool such as Orwell is required. Those who today are using Smalltalk for production applications do so with a small group of super programmers who must painstakingly cooperate to share their reusable components. Recently several projects have tried existing tools such as RCS and object oriented databases [Maier 86]. Unfortunately, both of these tools are only capable of source code management and source code based systems require each user to have a monolithic image. Although databases such as Gemstone are accessible from within Smalltalk, they are not integrated with the Smalltalk environment. Like most existing CM systems, they are seen as a necessary but unproductive overhead to the programming process.

In this paper, we describe Orwell, a configuration management tool for multiperson Smalltalk projects which addresses the problems discussed. Orwell provides both source and object code sharing as well as version control on a network of personal workstations. Class ownership is the primary means used for allocating work among programmers during the lifecycle of a project. We extend the scope rules for Smalltalk to provide private methods in public classes as well as private classes within applications. Section 2 of the paper describes the team organization underlying the successful use of the CM tool. The current implementation of Orwell is described in section 3.

2. The Orwell Environment

2.1 Organizing the Project Team

Successful configuration management requires more than a good tool. It requires the organization and management of a programming team. At a minimum, individual programmers must be made aware of their responsibilities as members of the team. It is important that the tool mesh with the reality of how a software project is managed. In this section, we briefly describe the new responsibilities and organization of Smalltalk programmers who work in the Orwell CM environment.

Programmers are organized into two groups that we refer to as *class programmers* and *application programmers*. They assume roles of class producers and consumers, respectively [Cox 86], [Jacobson 87]. *Class programmers* are responsible for the production of reusable components which are of general use to the organization. *Application programmers* seek to reuse as much code as possible from the existing class library. Their role is to configure existing classes, augmenting them where necessary for their specific application. They work to convince class programmers to improve their classes so that the amount of code specific to an application can be reduced. Note that this doesn't actually require that programmers be placed in one group or the other, but it does require them to wear the appropriate hat.

Programmers have mixed reactions to any management structure and Smalltalk programmers are reluctant to give up control over their personal image. As with any organizational structure, some programmers are attracted to a reduced scope of responsibility and others are reluctant. We have no new insights into managing teams of talented programmers; however, it is definitely impossible to do so without a tool. Indeed, an integrated tool can be used as an incentive to convince reluctant programmers to participate in a CM system.

Class Programmers

Class Programmers need to have long term ownership of their class or classes if projects in the company are to benefit from the class library. The reason is simple, every class needs to be polished and enhanced in order to achieve the promised goal of reusability. If no one has responsibility for the quality of a class and associated control over its evolution, experience shows that each project (in some cases each programmer) will modify the class to suit his needs. The programmer is often unaware of his impact on others or on the future reusability of the class. Some classes are born perfect, but most need to be polished, reorganized and augmented based on their use in different applications. Class programmers need to look to see how their classes are being used in applications if they are to improve them. This also encourages classes to be written and tested independent of a particular application. The production of a highly reusable class is no less valuable than the design of a new custom chip!

Application Programmers

Where does this leave application programmers, are they relegated to the role of second class programmers who must accept the classes produced by the components group? Clearly the answer is no, both groups are equally important. Properly motivated application programmers are concerned with delivering the solution, not the making of the ICs. Application programmers should seek to reduce the component count in their application by pushing the class programmers for more generalized classes. Fortunately, there are an increasing number of application programmers who are more interested in solving the problems of the user than in writing large amounts of new code. Object oriented programming, through toolkits and common libraries, allows more and more users to fill the role of an application programmer.

Class Ownership

Orwell is based on the proven concept of individual module ownership. Ownership allows project managers to assign responsibility for sections of the project to specific team members, thus avoiding conflicts in development and maintenance. Responsibility takes the form of ownership or control over class definitions and their methods, such that for each class defined in an application, a team member (application programmer) owns the definition and its methods. Applications are collections of classes and methods as described in detail in [Thomas 88]. The granularity of classes makes it feasible for a programmer to own one or more classes. Team members are restricted from redefining classes or adding/modifying methods to classes owned by other members.

Each existing class in the class library, which is extended or changed for use in an application, is also controlled by an application team member who is then responsible for the new methods. The class' definition and base methods are owned by the class programmer responsible for evolution of the class. By having only one team member responsible for an existing class in the library, any communication with the class programmer is then done by only one member from the application. Thus, each class programmer can work with a known group of application programmers to evolve his individual class. Class ownership is essential for the evolution of a large class library.

We believe that this organization or a similar one is required if multiple products are to be developed using a common class library. It is the separation of roles that is important while using Orwell; whether or not programmers are formally classified into a category is an issue which can only be addressed in the context of a particular project/organization.

2.2 Releases, Versions and Editions

In Orwell, the stages of application development are divided into the various *releases*, *versions* and *editions*. For clarity we will define the meaning of these terms as they are used in our system. A *class* consists of a Smalltalk class definition and a set of methods associated with that class definition. An *application* consists of a collection of classes which make up that application as well as a set of applications which it requires in one way or another. We call the applications it requires, *prerequisite applications*. An application makes use of classes in prerequisite applications by using their existing public methods. An application may also extend a class from a prerequisites application by subclassing it or adding application specific methods to that class (see 2.3 Visibility of Classes/Methods). A typical application consists of 5 - 10 classes and 1 - 4 prerequisite applications.

Both classes and applications have *versions* which are clearly identified points in the lifecycle of a class/application. The responsibility of deciding when to create a new version lies with the class/application owner. A *class version* consists of a class definition and the *editions* of its methods. An *application version* consists of a set of class versions and prerequisite application versions. Developers exchange versions to build an application/class.

Applications are the units of software managed within Orwell. An application is *released* by its owner to make it available to the users of that Orwell CM Environment. Implicitly when an application is released, its classes are also released to the other users of the system. A *release* constitutes a set of immutable versions which are subject to strict CM control and which exist for the lifecycle of the application. A separate tool, the *application packager* is used to construct stand-alone applications [Thomas 88] from an application release.

Class *editions* provide their owners with a complete development history. Class and method editions are only visible to their class/application programmer and allow her to experiment with improvements and bug fixes. The method version browser allows the programmer to move quickly between various versions and editions. Multiple releases of a product can be supported by a single programmer. For example, a programmer can open one edition to fix a bug for a version of a class and open another for a separately released version of the class to enhance it in some way. When a programmer is satisfied that an important progress point in the development cycle has been reached, she makes a version of the class from the appropriate edition. While there may be many editions of a method, class versions are always constructed from the current editions of the methods; in other words, there is no notion of a method version.

The use of releases, versions and editions addresses the different needs of class/application users and owners. It allows fine grained changes at the method level without inflicting an excessive burden on class users. Initially we allowed class users access to method versions, but this was to cumbersome to manage. Similarly, although outsiders may have to wait for a new release to gain access to an application class, we found the management of individual class releases required too much overhead. If class releases are really required, it is still possible to release an application containing just that single class.

Support for Multi-Site Development

In some organizations, geographical location or security dictate that the developers cannot be connected using a common file server. The latter situation is a requirement for our project. By using Orwell at the multiples sites and allocating ownership of the classes for the project among all the developers, the creation of a complete application is controlled. In such a case, each site would release its application with or without the source code to the other sites. Once receiving a release, the team at the site would merge it with their development environment and thus, still maintain the versions and owners of the classes and methods.

2.3 Visibility of Classes/Methods

Orwell partitions the Smalltalk name space into a number of separate applications. This has the benefit of reducing the name space which must be navigated by application programmers. It also provides additional information hiding beyond that already available in Smalltalk. Orwell provides this capability with an environment, whereas Modular Smalltalk [Wirfs-Brock 88] advocates a revised language to support similar notions. Within each application, the system restricts modification, using access control, of all public classes and methods to the application programmer for each corresponding class. If desired, the system will only show the specification of a method (its comment) and not its implementation. At present, we allow read access to the methods of all public classes. The access and visibility rules are described below. This part of the system can be easily tailored by the system manager to support more elaborate security, if it is required.

Public Methods: are available for use by all users in the system who have access to the class. This is the current way methods appear in Smalltalk.

Private Methods: are intended for internal use in the implementation of a class. In Smalltalk, private methods are defined by a commenting convention. We restrict their definition and use to the class programmer responsible for the class.

Application Methods: From time to time, it is necessary to extend the behaviour of a class in a way which cannot be achieved via subclassing. Typically, such methods are for a specific application (otherwise the class requires improvement). Application specific methods are only visible to the application team and the class programmer responsible for the class (read). Modification of the methods (read/write) is restricted to the application programmer in the team who is controlling the class.

Application Classes: Every application develops classes which are used internally for that application. We limit the visibility of such classes to team members working on that application. Each class is owned by one application programmer, who then assumes the role of class programmer for the class. It is his responsibility to enhance and expand the class given the input of the team in the same fashion as a class programmer enhances his class.

2.4 Global Objects Considered Harmful

Distributing a class or application between developers and users becomes overly complicated if global variables or persistent objects are used in the class/application. Our only solution to this problem is to require class programmers to provide initialize methods for all global, shared or class variables. These initialize methods are then executed the first time a class is loaded to restore the object to its previous state.

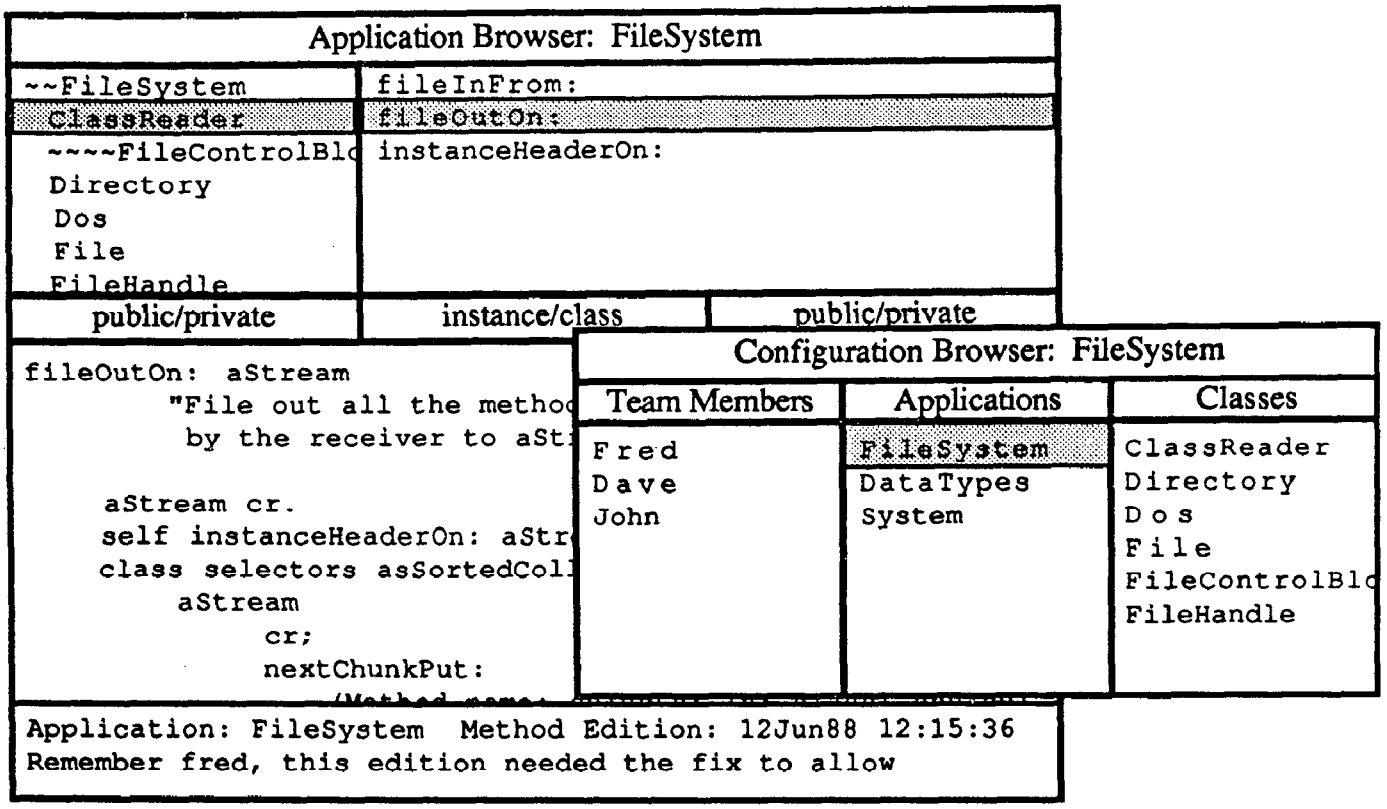

Figure 1: An example Configuration browser and Application browser for the application FileSystem

3. Implementation

3.1 Navigating The CM Environment

As displayed in the above figure, additional browsers have been implemented to manage class ownership and browse individual applications, for example.

Configuration Browser

A configuration browser on an application allows the team manager to assign class ownership between the team members and to designate releases of the application. Individual team members use the browser to designate one of their class editions as an actual version when they reach a milestone in the class' development. Menus in each pane provide the various functions permitted, given the selected team member, application and/or classes.

Application Browser

Application programmers browse their application separately with an application browser that is similar to a traditional class hierarchy browser, but only presents the classes owned or extended by the respective application. Classes can be easily added from the class library to the browser in order to extend them within the application. The classes from the application may be viewed with those of its prerequisite applications if required. The ability to do this is needed after defining a new application, which of course has yet to define any classes. Missing superclasses in the class list (likely if only the application is visible) are marked using a ~. Private and Public classes are differentiated to clearly define which are included in the interface of the application. Likewise, the Private and Public methods of each class are separated to clearly display its interface. A description field at the bottom of the browser allows comments to be entered that are specific to the chosen class or method edition.

Version Browser

Any saved method edition or version may be recovered with a method version browser that presents methods in a chronological order similar to single user versions of browsers introduced by Tektronix and Apple. In our case however, not only the source can be inspected, but the object code may also be retrieved quickly. Editions may also be deleted from the database individually or in groups.

3.2 Under The Hood

The Orwell environment is comprised of a *configuration* file for each team member and common class database that stores the source and object code for all applications, classes and methods. The object code is stored in a format

from which it can be easily loaded, thus avoiding the slow process of compiling the respective source code. The format of the class database is described below. Configuration files describe the programmer's current view of the common class database and contain a minimal Smalltalk image sufficient to access the database. The actual memory resident image is recreated at load time from the object code stored in the database. The dynamic construction of the programmer's image at runtime eliminates the need for multiple monolithic image files while allowing code to be shared by a large group of software developers. This saves considerable space on disk, since the average image size per team member increases with the complexity of the application; yet a large percentage of each image contains the same compiled code. In the current system all of the classes/applications identified in the configuration file are loaded into memory.

The existing Smalltalk changes file is used to record class definitions and the source for methods as well as to record all evaluated expressions. Since the changes file now contains only a summary of the actions executed by the user it is called the **doit** log. It is used by the programmer to retrace his steps in the development cycle. There is no longer any need for the Smalltalk programmer to *save* an image since all editions are automatically saved in the database. If for some reason an unusable image has been created (such as bootstrapping a new user interface), the system allows the user to revert back to a previous version or edition.

3.3 Organization of the CM Database

The class database contains the company's class library and the classes and methods of its applications. The database is implemented using a commercial Btree package. The package provides both rapid access to variable length records and concurrency control needed for shared access over the local area network. A user primitive is used to communicate between the Btree package and Smalltalk/V286 [Smalltalk/V286 88]. Classes and methods are stored and retrieved using keyed records. Keys contain the application, class and method names and a timestamp. The timestamp provides a unique identifier for each edition. The use of a common database eliminates class and method naming conflicts within an application. In this regard, the database serves the same role as the traditional data dictionary. Access to classes is controlled by maintaining the owner of each class within each application in the database.

Classes

A class is stored as a set of records, one for each component of the class. Each component is stored as a textual representation of the object. These components include its type (pointer, indexed, byte array...), its superclass, its instance and class variable names and its shared pools. In addition, the metaclass' type and its instance variable names are stored with the class. The class records are retrieved using the application name, class name and timestamp as the key. Each class edition contains a descriptive record which is used to explain the rationale for this edition of the class. The class' visibility and release status is contained in a separate entry.

Methods

Stored in the class database for each method is its source and object code. The source string is stored as is. Compiled methods are converted into a relocatable representation by scanning the methods for variable references, constants and method symbols. A textual representation of each symbol, variable name or constant is placed in the literal frame, which follows the actual bytecodes. The records are stored with a key comprised of the application name, class name, method name and timestamp. Each method edition contains a descriptive record which is used to explain the rationale for this edition of the method. This can be used to record fixes or features and for tracking maintenance activities. It facilitates the location of problem classes/methods which exhibit a history of problems and which therefore require redesign and reimplementation. We keep the full source for each method in a compressed format rather than using forward or reverse deltas [Tichy 85]. Fortunately the Smalltalk code is very compact and the available disk space on todays file servers allows the current 2mb image to grow to 200mb, which is more than adequate for our projects.

3.4 Dynamic Image Creation

The configuration file is used to guide the dynamic construction of each user's memory image. To reload a memory image the object code for each method is read from the database and linked. The essential methods needed to accomplish this bootstrapping process are part of the configuration file and are not stored in the database.

Recreating a compiled method from the stored representation in the database is a straightforward transformation of the stored string representations into the corresponding symbols, variable associations or constants. This activity is similar to linking object modules of other languages. To accelerate the process, each individual reference is stored with a corresponding type header to distinguish it from the others. The various types include method symbols, class, shared and global variables; plus all forms of literals, ranging from strings to large integers and arrays.

Smalltalk programmers expect responsive environments and will not tolerate excessive compile or load times. Fortunately, the linkediting process is only performed when an image is loaded and it can be performed quickly. To load our current image using our Smalltalk\V286 implementation on an IBM AT attached to a Novell file

server requires just a little more time than that needed to load the same image from a local disk. Most programmers load their image one to four times per day so the delay isn't significant. This delay is more than an acceptable price to pay for team programming. There is no perceived difference in the time to compile methods or retrieve source code. It is also possible to defer the loading of some classes or applications until they are referenced, however the additional effort is only appropriate for a very large image containing several disjoint applications.

4. Summary and Conclusions

Orwell is a configuration management tool for multiperson Smalltalk projects. It allows groups of Smalltalk programmers to develop code from a common class library. The system as currently implemented has negligible impact on the productive Smalltalk programming environment. It provides additional facilities for information hiding and security which can be tailored to meet the needs of the development organization. We have also described an organizational framework for multiperson software development.

The prototype system has been operational for a number of months and will be placed in production this fall. We expect that additional visibility controls are useful such as those suggested by [Synder 86]. Much more support is required for the software management including software metrics and bug/feature tracking. Our current solution eliminates persistent objects by placing the responsibility for their creation with the class/application owner. Ideally it should be possible to manage such objects in the same database. Using Orwell, Smalltalk can now be used to develop serious embedded computer applications using a team of programmers.

References

1. AT&T *Source Code Control System.* UNIX System V Programmer's Guide, Prentice-Hall, Inc. Chapter 14, pp. 659-700.

2. Cox, Brad J. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Don Mills, Ontario, 1986.

3. Jacobson, Ivar *Object Oriented Development in an Industrial Environment.* OOPSLA '87, Orlando, Florida, October, 1987, pp. 183-191.

4. Maier, D., Stein, J., Otis, A. and Purdy, A. *Development of an Object Oriented DBMS.* OOPSLA '86, Portland, Oregon, September, 1986, pp. 472-482.

5. Thomas, D. *Tools for Object-Oriented Software Engineering.* Internal Technical Note, School Of Computer Science, Carleton University, October, 1987.

6. Thomas, D.A, and Johnson, K. *Encapsulation & Instantiation of Smalltalk Applications.* Technical Note, School Of Computer Science, Carleton University, March, 1988.

7. Schmucker, K. Personal communication, January 1988.

8. *Smalltalk/V286 Object-Oriented Programming System (OOPS), Tutorial and Programming Handbook.* Digitalk Inc., 1988.

9. Synder, Alan *Encapsulation and Inheritance in Object-Oriented Programming Languages.* OOPSLA '86, Portland, Oregon, September, 1986, pp. 38-45.

10. Tichy, Walter F. *RCS - A System for Version Control.* Software Practice and Experience, Vol. 15(7), July, 1985, pp. 637-564.

11. Wirfs-Brock, A. and Wilkerson, B. *An Overview of Modular Smalltalk.* OOPSLA '88, San Diego, California, September, 1988.