Delivering and sharing components using Smalltalk link libraries

Makarand Utpat

RECENT WORK ON frameworks[1–4] depicts the advantages of components and frameworks in general. Developing a system architecture consisting of components and subsystems is an excellent idea from the modular design and software maintenance points of view. In this article, I will show how construction of Smalltalk link libraries (SLLs) aids in maintaining a developed application and delivering component-oriented software. I will describe a way to ship a Smalltalk application by creating SLLs. Each SLL can represent a subsystem or a specific component in your application. The process of Smalltalk library construction makes a developer realize the framework benefits at the time of delivery.

The code examples described are based on Digitalk Smalltalk/V Version 3.0.1 for OS/2 PM (Presentation Manager), but the same concept can be tailored to the Smalltalk/V Windows environment. This example also demonstrates how Smalltalk takes advantage of host operating system features for the application delivery. Example code can be obtained on the World Wide Web at http://www.objectpeople.on.ca/software.

SMALLTALK APPLICATION PACKAGING AND MAINTENANCE

It is a well-known fact that once the learning curve barrier is overcome, a Smalltalk developer becomes proficient in writing Smalltalk programs and the development task becomes fairly easy. In large projects, a considerable amount of time may be spent producing and maintaining a good executable image. Full-time resources may need to be allocated to perform this task. As the executable image size begins growing, a Smalltalk developer may hear comments about having a "fat" executable. Everybody expects a nicely trimmed executable for application delivery. There is absolutely nothing wrong with this expectation; it is easier to manage a smaller executable during application shipment. However, some momentum gained in developing Smalltalk applications can be lost in the process of constructing and delivering a Smalltalk executable.

Once development is complete, the organization begins a testing phase. As a result of code enhancements, code optimization, bug fixes, etc., code tuning begins as per feedback and suggestions of users. Often, these changes are specific to particular subsystem classes and/or methods, and classes and/or methods in other subsystems are not affected. Because the Smalltalk image (which uses v.exe, change.log, and recover.log files) is not partitioned, there is no way to focus the efforts on particular subsystem classes or methods to carry out code tuning. Here, SLLs come to the rescue. The organization can create a Smalltalk image that consists of SLLs for each of its subsystems and use them as needed, or distribute them appropriately to clients to satisfy client requirements. These SLLs are used either in a development or runtime environment as per one's needs. SLLs are very handy for accelerating the overall application packaging task and maintaining the Smalltalk application.

A REAL-LIFE EXAMPLE

Assume that an organization has developed a software framework to satisfy its business needs. Consider a simple situation where this framework uses a dependency mechanism (i.e., the user interface is dependent on a business model) and is comprised of two subsystems including a *business model subsystem* (to contain business state and business logic) and a *user interface subsystem* (to display business model data), etc. In reality, framework implementations may contain other subsystems such as a *rule model subsystem* (to handle business rules), an *application model subsystem* (to handle behavior related to the application model), a *communication model subsystem* (to handle message send/receive), a *database subsystem*, etc. Classes in these subsystems collaborate with each other appropriately per their responsibilities to provide the behavior specified by the framework.

Also assume that there is a client base of this organization that wants to use the framework-level classes (i.e., high-level abstract superclasses) from these subsystems to ensure they comply with the overall organization framework. In addition, they want to utilize both framework-level and concrete-level classes in an existing business model subsystem, and customize their user-interface model differently than the one used by the organization.

Until recently, few options were available to handle this situation. If the organization and its clients were using just a Smalltalk (without Team/V) environment, then the only option was for the organization to give its clients the whole Smalltalk image, containing the base Smalltalk image, user-interface classes (abstract superclasses), and business model classes (framework- and concrete-level classes). If the organization and its clients were using Smalltalk with Team/V, then clients could access business model subsystem classes using the individual "package migration" technique[5] or by creating specific tools (visual or textual) on top of Team/V to handle "migrating a list of packages." These approaches are on the verge of obsolete for a variety of reasons including limitations encountered during application packaging and maintenance phases; the Parcplace-Digitalk merger, which is resulting in a new architecture; and an overall industry shift toward componentware.

DYNAMIC LINK LIBRARIES AND SLLs

OS/2 provides a powerful way of bundling your application program into a coherent unit called a *dynamic link library* (DLL). As the name suggests, a DLL provides services that are accessed and linked dynamically by different application programs. It provides a way for an application program to dynamically reference and access functions and resources outside its own executable environment.[6] Such resources might be icons, bitmaps, or pointers, etc., whereas functions could be the PM application programming interface (API) calls for handling window management, graphics, device driver routines, or accessing OS/2 executable programs, etc. Smalltalk/V PM with the use of SLLs takes this concept one level higher, allowing one to access the services provided by individual subsystems. By tapping the power of OS/2's DLL mechanism underneath it, Smalltalk/V PM allows one to create SLLs of classes, methods, and metaclasses that can be shared by different teams within an organization. Object libraries were a common means of building DLL files in Digitalk Smalltlalk previous to Version 3.0.1. SLLs are those object libraries with a new face.

DELIVERING A SMALLTALK APPLICATION

In Smalltalk/V PM, the Smalltalk image consists of the executable environment (v.exe file, change.log file, recover.log file), one or more DLL files, and one or more SLLs. DLLs are divided into *base class DLLs* and *development class DLLs*. Base class DLLs contain classes such as collections, streams, windows, etc., whereas development class DLLs contain the Smalltalk compiler, debugger, etc.[6] A Smalltalk programmer uses the development classes to create a Smalltalk application program. As development progresses, the code created by the programmer (once saved) is added into the v.exe file, which begins to grow. Typically a programmer uses the v.exe file to deliver the application. This approach works well whether the design contains a relatively small or large number of classes. Then, by making appropriate changes in the #startUpApplication method in the NotificationManager class, the programmer uses v.exe to start the application. This standard approach always works.

Another way to deliver a Smalltalk application is filing out all the classes from one programmer's image and installing them on another programmer's image.

Both approaches are cumbersome from the application maintenance point of view. A better way is to build Smalltalk libraries of classes to provide flexibility in delivering Smalltalk applications.

DIFFERENT WAYS TO CONSTRUCT SLLs

Before transferring the executable to users, programmers test their applications to confirm that they meet requirements. It might be a good idea to start building SLLs at the initial application development phase, to ease the future maintenance task. Such SLLs can be loaded in a development or runtime environment. Here, the task of Smalltalk library construction can be accomplished in a variety of ways. One could categorize it based on different aspects such as the existing services, different subsystem frameworks, or available standalone classes (explained next). Referring back to our example, assume that a simple business model for the organization consists of classes such as Person, Address, HomeAddress, BusinessAddress, Phone, HomePhone, BusinessPhone, etc., while the user-interface model consists of classes such as UserInterfaceModel, UserInterfaceModelForOrganization, and PersonInformationWindowDialog (to display Person information). Combined, these different classes could provide a "view" service that enables one to view information about persons.

Construction based on the service behavior

The organization could treat the aforementioned business model and user-interface classes as one entity, and construct a single Smalltalk library representing a Person information "view" service, i.e., construct one Smalltalk library that contains business model subsystem classes (abstract and concrete) and user-interface subsystem classes (abstract and concrete) to provide this specific service. If the organization needs a "change" service, consisting of additional user-interface classes such as ChangePersonInformationDialog (to change the Person information) and other business objects created to handle the changes, the organization could then construct another Smalltalk library representing a "Person information change service." In general, this approach facilitates development efforts within the organization.

Smalltalk library for view service:

> BusinessModel Subsystem (abstract and concrete classes) and UserInterfaceModel Subsystem Smalltalk Library (abstract and concrete classes related to viewing the person)

Smalltalk library for change service:

> BusinessModel Subsystem (abstract and concrete

classes) and UserInterfaceModel Subsystem Smalltalk library (abstract and concrete classes related to changing the person information)

Construction based on framework implementation

In construction based on framework implementation, the organization treats the aforementioned business model classes and user-interface classes as two different entities and constructs two separate SLLs. Thus, by constructing two SLLs for business model subsystem classes (abstract and concrete) and user-interface subsystem classes (abstract classes), the organization creates a server environment that provides services such as knowing the state of the business objects and providing user-interface protocols to display the current state of the business objects. The organization and its client base use this server environment to customize user-interfaces appropriately. In general, this approach is best used when the organization has to satisfy client base requirements.

The task of Smalltalk library construction can be accomplished in a variety of ways.

Server SLLs:

BusinessModel Subsystem Smalltalk library	UserInterfaceModel Subsystem Smalltalk library

Client SLLs:

UserInterfaceModel Subsystem Smalltalk library (concrete classes for organization)

UserInterfaceModel Subsystem Smalltalk library (concrete classes for clients)

Construction based on standalone classes

One can construct a Smalltalk library of standalone classes that don't belong to a particular subsystem but are required by different subsystems, such as classes for managing application configuration, setting application environment, other helper classes, etc. This Smalltalk library can then be treated as a standalone component in the application.

CREATING A BUSINESS MODEL SMALLTALK LIBRARY

Assume that the organization follows the second approach to constructing a Smalltalk library, which results in their having two DLLs—one for business model classes and one for user-interface classes.

The typical steps to create a business model Smalltalk library are described below:

1. Open the Library Builder dialog. Select the package containing business model subsystem classes. Select the following menu option: Module —>Build Library.
2. The Library Builder dialog offers two options. Customize the classes that you would like to add into the library by clicking on the Customize option or just let create an SLL for the classes contained within the selected package/cluster. Also, one can optionally include the source code for classes not in SLL one is creating.

BINDING SLLs TO A SMALLTALK IMAGE

The Digitalk help manual describes three ways to package Smalltalk application using SLLs; these approaches let the developer bind SLLs statically or dynamically.

The first approach is to bind SLLs during startup by including their names in an autobind ascii file, e.g., app.bnd (during development image, it looks for vdevo.bnd file). This approach permits one to save the image without binding it with the SLL, thus avoiding having to bind the SLL to a specific version of v.exe.

The second approach is to bind the SLL dynamically. The developer can then bind and unbind SLLs on demand, which results in low memory overhead.

Finally, one can bind SLLs in a hybrid way using a combination of the aforementioned two approaches: binding some SLLs to the image and binding others dynamically.

ADVANTAGES OF USING SLLs

1. Data sharing—Multiple applications access and share subsystem SLLs. A server environment is created by storing different SLLs on the network. All teams within the organization would then have ready access to it so that consistent access is maintained. Thus, SLLs make data sharing a transparent process across multiple applications.
2. Pluggability—Modular components are created to enhance reusability. Thus, the maintenance task becomes flexible. This aids easy replacement and shipping of appropriate subsystem DLLs.
3. Application of the producer/consumer concept—This key concept of componentware (producing and consuming the components) is easily adopted and applied. Referring to the example at the beginning of this article, the organization becomes a producer of SLLs and the client base becomes a consumer of SLLs.
4. Decreased image size—This conserves hard disk space and reduces v.exe size, resulting in less overhead.
5. Flexibility of use for the organization and its clients—Once the business model Smalltalk library is created, it is ready for distribution to the client base, as well as use by the organization itself. The client base, which is uninterested in the user interface Smalltalk library (containing classes such as UserInterfaceForOrganization, its subclasses, composite panes used in dialogs, etc.), can load the business model Smalltalk library in their development/runtime environment and begin using it alone.
6. Creation of standalone class libraries—Components consisting of standalone classes are constructed and distributed appropriately.
7. Realization of the framework benefits—During the

design phase, classes (both framework- and concrete-level) are logically grouped to form a subsystem based on the intended behavior that a subsystem is supposed to perform. The correct decision to group a certain class in a particular subsystem aids the task of building and maintaining SLLs.

8. Construction simplicity—Previous Digitalk Smalltalk versions used the concept of object libraries, and Smalltalk developers spent a lot of time constructing them. On the other hand, SLLs are constructed simply and quickly.
9. Platform portability between OS/2 and Win32 operating systems—SLL uses a unique and system-independent format.[7] Hence, it is easier to create applications that are portable between these two platforms.
10. Use in runtime and development environments—Works excellently in both runtime and development Smalltalk environments.
11. Scripting—During a release phase, a particular Smalltalk library may have to be reconstructed many times. By taking advantage of scripting facilities,7 the task of constructing SLLs is simplified by creating scripts to reconstruct SLLs.

Classes are logically grouped to form a subsystem based on the intended behavior that a subsystem is supposed to perform.

DISADVANTAGES OF USING SLLs

1. Difficult to use in a Team/V development environment—If you load a Smalltalk library in a Team/V development environment, it loads all the classes in the "unpackaged" package and not where they belong. Because most of the source code related to Team/V classes is hidden, it is difficult to ascertain how to replicate the "Load/Migrate" action (which loads all the classes in a particular package/cluster in one's Smalltalk image) during Smalltalk library loading so that classes included in the Smalltalk library will fall into packages where they belong.
2. Inability to extend existing Smalltalk library environment—To build a Smalltalk library, Digitalk uses a few classes (SmalltalkLibrary, SmalltalkLibraryBind, TeamVLibraryInformation, TeamVInterface, etc.) whose implementation is hidden to the developers. Because they cannot access the code associated behind the methods, they cannot add any enhancements to base Smalltalk library classes.

CONCLUSION

This article reviewed different approaches to constructing SLLs and described advantages to facilitate the Smalltalk application delivery task. It provided shared transparent access to different teams by conserving hard disk space. I found that the ability to create SLLs provides a pluggable approach that facilitates application maintenance tasks. The article also showed how with this approach, the momentum gained in developing Smalltalk applications is retained while delivering Smalltalk applications.

Acknowledgment

The author thanks Anne Marie Frederick at Prudential Insurance Corporation in New Jersey for encouraging him to write this article.

References

1. Johnson, R. and B. Foote. Designing reusable classes. JOURNAL OF OBJECT ORIENTED PROGRAMMING 1(2):22–35, 1988.
2. Taligent. BUILDING OBJECT-ORIENTED FRAMEWORKS, 1993.
3. Harris, J. Object Insider: Breaking out of the object ghetto, OBJECT MAGAZINE 4(8):12–14, 1995.
4. Johnson, R. How to develop frameworks, TUTORIAL NOTES OF OOPSLA'93, 1993.
5. Digitalk Inc. DIGITALK PROGRAMMING REFERENCE FOR SMALLTALK/V FOR OS/2, 1993.
6. Petzold, C. PROGRAMMING THE OS/2 PRESENTATION MANAGER, Microsoft Press, 1989.
7. Digitalk Inc. DIGITALK ONLINE HELP MANUAL FOR SMALLTALK/V FOR OS/2, 1995.

Makarand Utpat is a Senior Consultant at Envision in St. Louis, MO.