Accepted to WoDiSEE workshop at the International Conference on Software Engineering 2004.

Software Engineering Environment Support for Frameworks
A Position Paper

George Fairbanks
Carnegie Mellon University
Pittsburgh, PA 15232
George.Fairbanks@cs.cmu.edu

Abstract

Modern software engineering environments provide programmers with contextual information on methods and classes in their software system. However, programmers increasingly encounter frameworks such as Microsoft Foundation Classes or Enterprise Java Beans whose required plug-in mechanisms cannot be adequately described with method or class documentation. The needs of programmers using frameworks are different from the needs when using libraries. Software engineering environments should be augmented with tools to aid programmers in understanding the demands of using frameworks.

1. Introduction

Modern software engineering environments provide context to inform and guide the programmer. Examples include structured method- and class-level documentation like JavaDoc [2], method name and parameter completion, a tree view presenting packages in the project, and an outline of the class being edited. These features help a programmer make use of a code library by prominently exposing its API. A programmer from 1974 would be elated to use a modern Software Engineering Environment (SEE) because it makes using libraries of code much easier.

At the same time that mainstream SEE's have gained these capabilities, object oriented frameworks have increased in popularity. Frameworks such as Microsoft Foundation Classes [8] or Enterprise Java Beans [5] provide many benefits at the cost of greater complexity and more constraints. Modern SEE's do not yet provide enough support to address the difficulties programmers encounter when using a framework. Better tools for working with libraries have been matched by new difficulties using complex frameworks. This increased complexity means that despite the progress in SEE's, programmers in 2004 may be no better off overall in handling the complexity of their software systems.

The following example illustrates how plugging into a framework differs from using a library. In this example a programmer is using the Eclipse tool framework [7] and has created a tree view that appears on the screen. (Note: in this example the programmer is adding new capabilities to the Eclipse software engineering environment, not just using the environment). He wants his class to be notified when the user selects one of the entries on the tree view. The class he has created must do the following:

1. It must subclass from ViewPart, a class defined in the Eclipse framework.
2. It must implement the template method design pattern [1] named createPartControl defined in the ViewPart class.
3. It must create and hold a reference to its TreeViewer, the framework widget representing a tree.
4. Inside the createPartControl method, it must register for selection changed events by invoking the addSelectionChangedListener method on its TreeViewer, passing itself as a parameter.
5. It must implement the ISelectionChangedListener interface (defined in the Eclipse framework) and its required method, selectionChanged.
6. The implementation of selectionChanged must receive the SelectionChangedEvent parameter and invoke the method getSelection on it, yielding an ISelection. Because the programmer has implemented a tree view, that means it is safe to downcast the ISelection to a IStructuredSelection and invoke getFirstElement, yielding the object that the user selected.

In summary, to accomplish his goal of receiving events related to the user selection, the programmer must implement two methods defined by the framework, subclass from a framework class, implement an interface from the framework, hold a reference to a framework class, invoke methods on four framework classes, and know that in this usage he can safely downcast a framework class. Furthermore, this not just one solution or the most popular solution; it is the only permissible solution within the framework constraints.

This complexity of this example is typical of what is required to plug into a framework. Although the example is taken from the Eclipse framework, examples with similar complexity and structure can be found in most frameworks.

A comparison of tools available in a SEE with the needs of the programmer who is attempting to plug into a framework will identify some gaps. Most of these gaps arise because the plugin point has no representation in the SEE and exists only as a concept in the programmer's mind.

The remainder of this position paper will elaborate on what frameworks are, why they make programming more complex, what tasks programmers perform when using a framework, and how they cope with limited tool support. This paper intends to identify and describe areas where programmers need additional help.

2. Frameworks

An object-oriented framework is characterized by an inversion in the normal control relationship between the programmer and the library [6]. In the traditional usage of a library, the programmer decides when to invoke library functions and which ones to invoke. In a framework, the programmer is told by the framework when and where his code will be called.

Programmers are willing to sacrifice this control in order to receive other benefits. Examples of benefits include automatic management of transactions and concurrency (Enterprise Java Beans), operation within a windowing operating system (Microsoft Foundation Classes), and operation within a software engineering environment (Eclipse framework). Typically the size of the framework code will dwarf the size of applications written within it.

Frameworks typically provide the superstructure of the application and the skeleton of its control flow [3]. Programmers must attach their code, commonly called plugins, to the framework at predefined points, commonly called plugin points. Somewhat confusingly, the word "plugin" may refer to the entire program written by the programmer or just the subset of it that connects to the plugin point of the framework. Common mechanisms for plugging into a framework include subclassing using the template method, registering for callback events using the observer pattern, or installing new handlers using the strategy pattern [1].

The complexity of libraries can rival that of frameworks in some cases, for example socket programming. These libraries are characterized by constraints on the sequence of permitted library calls. Such libraries are much like frameworks in the demands that they place on the programmer.

3. Complexity of Frameworks

With the benefits of a framework come the costs. An inherent attribute of frameworks is a dramatic reduction in the size of the solution space available to the programmer. At times the solution space may consist of a single path, as seen in the earlier example. Frameworks must constrain the solution space in order to provide their benefits. For example, it is difficult to imagine how Enterprise Java Beans could provide assurances regarding concurrency if beans were allowed to create and mange their own threads. In the worst case the framework may constrain the solution space so much that there is no available path for the programmer to accomplish his goal. For example, for many years the Enterprise Java Beans framework lacked a mechanism to schedule a bean to execute at a particular time.

Since frameworks typically provide the skeleton of the structure and control flow of an application, it is difficult to browse just the plugin source code and comprehend the application. This is in contrast to an application similarly written with libraries, where it is easy to understand the basic structure and control flow from looking at the part the programmer has written. This is true because abstraction works better at the top of a hierarchical decomposition than at the bottom. In the library case, the programmer writes the most abstract parts of the program and the libraries handle the details; understanding the big picture becomes easy. Conversely, when using a framework the programmer provides the details and the framework provides the structure; understanding the big picture is much harder.

Programmers find it difficult to identify the plugin points of a framework [4]. Frameworks are built using the same raw materials as any other object oriented program, namely classes with methods and attributes, so it is difficult to identify which parts of the framework are intended to be hidden machinery and which parts are candidates for replacement or augmentation. Future languages may provide mechanisms to make the distinction explicit but mainstream languages today lack an explicit representation of framework plugin points.

As was the case in the example, plugin points often consist of more than a single method or class in the framework. They instead require the deliberate use of multiple parts of the framework in the correct sequence. Further complicating the situation, a given class or method may be used in multiple plugin points.

There are two additional attributes of frameworks that make them difficult to use. It is often the case that the programmer writing a framework plugin is not allowed to change the framework in any way, perhaps because it has been created by another organization. This limits the programmer's options and forces him to work around

framework bugs or design flaws. In more extreme cases the programmer will additionally have no access to the source code of the framework. This limits his ability to comprehend the framework to reading documentation and experimentation. Both of these cases can also occur with traditional libraries with similar consequences.

4. Provided tools

The tools available in modern SEE's are quite effective at supporting the use of libraries because a library can be used effectively when the programmer understands a single class or even a method in a library. The tools that modern SEE's provide aid the programmer in understanding the method calls that can be made on the library (the API) as well as the structure of his own code.

Examining a list of tools available in a SEE reveals many whose primary purpose is to make library API's more visible and immediate, including structured method and class level documentation (e.g., JavaDoc) and automatic method name and parameter completion.

Other tools are provided to provide insight into the structure of libraries or code that the programmer has produced. Examples of these tools include a tree view displaying packages in the project and an outline view displaying the structure of a class. When using libraries instead of frameworks, the structure of the programmer's code can be used to understand the structure of the entire system.

5. Programmer needs

The tools provided in current SEE's are helpful but insufficient for the needs of programmers using frameworks. Ultimately, the insufficiency derives from the lack of representation for a plugin and a plugin point. Proximately, the following tasks should be supported with tools.

"How do I accomplish this?" In an unconstrained environment, a programmer is limited only by his creativity in solving problems. Programmers learn early on to solve problems given existing resources. When using a framework, the programmer is limited to the mechanisms provided by the framework and likely cannot build his own mechanisms. Finding a path to accomplish his task is harder. Furthermore, discovering resources is easier when using a library, because a quick scan of available method names may reveal one that is helpful. No similar vehicle is provided for frameworks, but should be.

"Have I done all I need to do?" If the documentation of a library call is reasonably complete, a programmer can assure himself that his code invokes the library correctly and handles its return values. As was shown in the example, plugging into a framework consists of a number of coordinated tasks yet it is hard or impossible to be sure that all the necessary tasks have been done. The method- and class-level documentation provided are insufficient to convey the complexity of a plugin point that spans methods and classes.

"What is going on here?" When browsing an existing program that calls a library, it is easy to identify the calls to the library. In contrast, when browsing code that plugs into a framework, it is difficult to identify the parts of a program that are participating in the plugin. The design intent of plugging into the framework is missing, as is a list of the parts that are participating in the plugin. Consequently, programmers browsing existing code for maintenance or to learn how to use a framework may make mistakes.

The ultimate problem is that there is no representation of the framework plugin except in the programmer's mind. An explicit representation of some form will be needed for SEE's to address these identified gaps between the needs of programmers and the tools that are provided to them.

6. Coping Strategies

Despite the complexities of using frameworks, programmers do manage to work within their constraints. A short description of some of the current coping strategies and how current tools are used is illuminating.

First, programmers browse example code. The example code is known to work. Naming conventions usually indicate which classes are from the framework and this aids identification of plugin points. Programmer-written code that subclasses from framework classes or implements framework interfaces is a place to begin looking for plugin points also. Determining the full extent of the plugin point is still difficult but programmers can err on the side of caution. Programmers may inadvertently copy extra code, mistakenly thinking it to be required.

Second, programmers can use their SEE's to browse available APIs and API documentation. Even when the documentation for a single method is not the whole story, it is at least part of the story. Classes and methods are the building blocks of framework plugin points and their documentation helps the programmer build a mental model of the framework and its plugin points.

Third, programmers can browse the framework source code. Since the framework code is often large, browsing can be inefficient. Once browsing the framework source code, the programmer is exposed to all the detail needed, but also much more than is required.

Fourth, programmers consult external documentation on the framework such as books. Books on frameworks

are usually not comprehensive and become stale as the framework evolves. Despite these drawbacks, narrative descriptions of the framework and its plugin points are probably the most effective way to learn to use a framework. Notably, the existence of such books is a strong indication that our current API-level documentation alone is not fully meeting the needs of programmers.

7. Conclusions

Improved understanding of a library call can prevent errors due to incorrect parameter passing, return value handling, or inappropriate usage. If there is an analogy between understanding a library call and understanding the framework plugin points then the most significant outcome of supporting frameworks in SEE's will be a reduction in errors in programs. It is even easier for programmers to make errors when plugging into frameworks because of their increased complexity, so improving understanding of the plugin point has great potential to reduce errors.

If structured representations of framework plugin points are created, it will become possible for tools to detect errors automatically instead of just enabling the programmer to more easily detect his own errors.

Software engineering environments are an appropriate high leverage place to provide tools for programmers to interact with frameworks. The relative lack of tools for frameworks in modern SEE's should be seen as an opportunity to support current programming practice.

8. Acknowledgements

The author wishes to acknowledge support from the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298 and the NASA Director's Research and Development Fund.

References

[1] Erich Gamma et al., *Design Patterns*, Addison-Wesley, 1995.

[2] Javadoc Tool Home Page, http://java.sun.com/j2se/javadoc/

[3] Ralph Johnson and Brian Foote. ``Designing Reusable Classes." Journal of Object-Oriented Programming. SIGS, 1, 5 (June/July. 1988), 22-35.

[4] Ralph Johnson, "Frameworks = Patterns + Components," Communications of the ACM, Vol. 40, Oct 1997.

[5] Richard Monson-Haefel, *Enterprise Java Beans - 2nd Edition,* O'Reily, 2000.

[6] Douglas Schmidt and Frank Buschmann, "Patterns, Frameworks, and Middleware: Their Synergistic Relationships," Proceedings of 25th International Conference on Software Engineering, 2003.

[7] Sherry Shavor et al., *The Java Developer's Guide to Eclipse,* Addison-Wesley, 2003.

[8] George Shepherd and Scot Wingo, *MFC Internals,* Addison-Wesley, 1996.