Hooking into Object-Oriented Application Frameworks

Garry Froehlich, H. James Hoover, Ling Liu, Paul Sorenson
Department of Computing Science
University of Alberta
Edmonton, AB. T6G 2H1
{garry,hoover,lingliu,sorenson}@cs.ualberta.ca

ABSTRACT

An object-oriented application framework provides a generic design within a given domain and a reusable implementation of that design. Frameworks have great potential for reuse, but that potential can only be achieved if the framework can be understood and used effectively by application developers. The complexity of the design and implementation of a framework can make understanding the framework difficult, and so good documentation and guidance to the framework user is needed. Several methods exist for documenting the design of the framework, but less work has been done on documenting the purpose of the framework and how it is intended to be used. In this paper, we introduce the concept of hooks as a means of documenting and providing guidance on the intended use of a framework. Each hook provides a concise solution to a focused requirement or problem within the context of the framework. Solutions for more complex requirements can be provided by combining several hooks. A discussion and means of describing hooks is given, with a focus on the method of adaption used to fulfill the requirement and the level of support provided for the solution within the framework.

Keywords

Object-oriented application frameworks, software reuse, framework documentation

INTRODUCTION

Software reuse is a potential means of improving software productivity and quality; however, small pieces of software often cannot be reused outside of their original context. Object-oriented application frameworks not only provide software for reuse, but also provide the context in which that reuse takes place. Applications are built from frameworks by extending or customizing parts of the framework, while retaining the original design. An object-oriented application framework is the reusable design (the context) of a system or subsystem implemented through a set of abstract classes and their collaborations [1]. The design provides a means of organizing related components without placing the classes or components in a generic library. Each component is reused in the context for which it was created and does not have to be modified. Frameworks solve larger-grained problems than individual components. This makes the effort of finding and reusing them much more cost effective than for small components. Frameworks currently exist in areas such as graphical editors (HotDraw [1, 10], UniDraw [19]), user interfaces (ET++ [20]), manufacturing systems (OSEAFA [17]), client-server communications [3], and operating systems (Choices [5]).

A framework, like any other type of reusable software, should take less time to understand and use than it would take to build an equivalent application without the framework. Due to the potentially large size and complexity of frameworks, the ability to quickly understand and apply a framework is a critical issue. In this paper we introduce the notion of a *hook* as a means of easing the understanding and use of a framework. Hooks focus on how the framework is intended to be used, and provide strong guidance in how and where to perform the changes that will fulfill some requirement within the application being developed. *They provide an alternative and supplementary view to the design of the framework.* A hook may involve something as simple as inheriting from an existing class in the framework, or as complex as adapting the interaction of a large number of classes within the framework. The *framework builder*, who is the most knowledgeable about the framework, defines the hooks and through them passes on his/her knowledge to the *application developer*. In this way, the framework builder can tell the application developer what needs to be completed or extended in the framework, or what choices need to be made about parts of the framework in order to develop an application using the framework. The hooks also help to ensure that changes or extensions integrate smoothly into the design and implementation of the framework by imposing some constraints on the changes.

To further define hooks and to assist in their management, hooks are categorized along two axes. The first states the method of adaption that the hook uses to fulfill a requirement. The second indicates the level of support that is provided for the specific changes needed to fulfill a requirement. The level of support ranges from having pre-defined components available to requiring the developers to extend the framework or provide their own components.

The rest of this paper provides an overview of related work on framework documentation, followed by a more detailed discussion of hooks. Hooks are first described in general with examples from the HotDraw [10] framework. Next, a categorization of hooks by method of adaption and level of support is given. We then discuss how hooks might be used and describe our initial experiences with hooks and frameworks. The last section presents a summary of the work and some future directions.

BACKGROUND

Frameworks can be quite complicated and difficult to understand. Properly documenting a framework is important in order to ease its understanding and use. Johnson [10] proposes that three types of documentation are needed for frameworks.

1. *The purpose of the framework.* A description of the domain that the framework is in, the requirements it is meant to fulfill, and any limitations it has.
2. *The use of the framework.* A description of the way the framework builder intended the framework to be used.
3. *The design of the framework.* A description of the structure and the behavior of the framework.

Much work on framework documentation has focused on the design and architecture of the framework. Three approaches that have been used are: providing several views of the design, using design patterns, and using exemplars.

Campbell and Islam [4] propose a six part approach to documenting the design. They recommend the use of class hierarchies, interface protocols, entity-relationship diagrams, control flow diagrams, synchronization path statements and configuration diagrams to describe the structure and behavior of the framework. All of these diagrams concentrate on the design of the framework and not on the purpose of the framework or how to use it.

Design patterns [6] are a popular means of describing frameworks. Beck and Johnson have used them to help show how the architecture of the HotDraw framework is derived [1]. The OSEFA framework has also been described using design patterns [17]. Further work has been done to allow the interactive visualization of design patterns within a framework [13]. Design patterns can help to show the decisions that were made regarding the design of the framework. Using commonly known design patterns can also help developers understand the framework by serving as a common vocabulary between the framework builder and the application developer. The application developer can see that a design pattern was used and immediately understand some of the advantages and limitations of the design. Although design patterns are an excellent means of documenting the design, they do not explicitly describe the purpose of a framework, nor how it should be used.

Exemplars [7] provide a different means of understanding frameworks. An exemplar consists of a concrete implementation provided for all of the abstract classes in the framework, and their interactions. The interactions between classes can be explored through the exemplar using a special tool to provide a deeper understanding of the framework. Class hierarchies can also be browsed by the tool to make choosing the correct classes easier. While this approach allows for the understanding of the design of a framework, it is not as well suited for providing the purpose of the framework, or how the framework is meant to be used.

Less work has focused on the purpose and intended use of frameworks. Two such approaches are cookbooks and patterns. The cookbook in [11] consists of a general description of the purpose of the Smalltalk Model-View-Controller (MVC) framework, the major components of the framework and their roles, and follows with a number of examples to illustrate how the components can be used. It is presented as a tutorial to learn the framework. A different type of cookbook found in [14] focuses on specific issues such as how to create an active view in MVC. Each entry in this cookbook defines a problem to solve and then gives a set of steps to follow along with some examples for solving the problem. However, the steps are narrative descriptions and are not well-structured or uniform.

Johnson's patterns [10] fall roughly into the same category as a cookbook, documenting the purpose and use of a framework as well as a little of the design. Each pattern describes a problem that application developers will face when using the framework, gives general narrative advice and examples about ways to solve the problem, summarizes the solution, and refers to related patterns. The collection of patterns makes up a directed graph indicating the order in which to read them, starting with the main pattern which describes the purpose of the framework.

Lajoie and Keller combine the idea of design patterns with Johnson's patterns, which they rename motifs, to provide a more complete description of a framework [12]. In their strategy, motifs point to design patterns, contracts [8] and micro-architectures to help provide the developer with an understanding of the architecture of the framework in the context of the problems it was meant to solve. Motifs give advice and examples on how to design solutions to problems using the framework and help to show the purpose of the framework, but they are no more structured than patterns.

Hooks focus on the intended use of the framework much like cookbooks or motifs but do not focus on the design like design patterns or exemplars. Hooks provide an alternative view to design documentation. Rather than presenting the design or even the reasons why a particular design was chosen, hooks show how and where a design can be changed. They present knowledge about the usage of the framework.

Hooks are more structured and uniform than motifs or cookbooks and help the framework builder to be more precise about how the framework is intended to be used. Further, the type system introduced for hooks in this paper shows many of the ways in which changes can be made to a framework and how they can be supported, and provides a basis for application developers to find the hook they need.

HOOKS

Hooks provide solutions to focused problems. For more complex problems, groups of hooks can be provided with each focusing upon a smaller problem within the larger, more complex problem. Each hook details the changes to the design that are required, the constraints that must be followed and any effects upon the framework that the hook will impose, such as configuration constraints. Only the information needed to solve the problem is provided within the hook. Developers are then able to quickly understand and use the hook.

Hooks are meant to be used for developing applications from a framework, not developing new frameworks from old ones. Selecting options, filling in parameters or extending the framework for a particular application are all hooks. However, modifying or refactoring an existing framework is beyond the scope of this paper and are not considered to be valid hooks.

Each valid hook is written in a specific format made up of sections. The sections detail different aspects of the hook, such as the components that take part in the hook (participants) or the steps that should be followed to use the hook (changes). The sections serve as a guide to the people writing the hooks by showing the aspects that should be considered about the hook, such as how using it affects the rest of the framework (constraints). The format helps to organize the information, helps to prompt for the required information and makes the description more precise and uniform, which aids in the analysis of hooks and the provision of tool support for them. Each hook description consists of the following parts:

- Name: a unique name, within the context of the framework, given to each hook.
- Requirement: the problem the hook is intended to help solve.
- Type: an ordered pair consisting of the method of adaption used and the amount of support provided for the problem within the framework.
- Area: the parts of the framework that are affected by the hook.
- Uses: the other hooks required to use this hook. The use of a single hook may not be enough to completely fulfill a requirement that has several aspects to it, so this section states the other hooks that are needed to help fulfill the requirement.
- Participants: the components that participate in the hook. These are both existing and new components.
- Changes: the main section of the hook which outlines the changes to the interfaces, associations, control flow and synchronization amongst the components given in the participants section. All changes, including those involving the use of other hooks, are intended to be made in the order they are given within this section.
- Constraints: limits imposed by using the hook, such as configuration constraints.
- Comments: any additional description needed.

Not all sections will be applicable to all hooks, in which case the entry not required is simply left out. For example, a hook that does not use any others will have no Uses declaration.

Our first example of a hook shown in Figure 1 is derived from our experience using HotDraw to construct a graphical display application for the Responsive Corporate Information Model (RCIM) [18] manager prototype. HotDraw is a framework from the University of Illinois at Urbana-Champaign for developing structured graphical editors written in Smalltalk. A view of a HotDraw application is shown in Figure 2. The major elements of HotDraw on which we focus are the tools (shown in the toolbar on the left) which are used to manipulate figures (the polygons within the window on the right) within a drawing (represented by the entire drawing window on the right). The tools provided

Name: New Tool Type
Requirement: A new type of tool is needed which is not already provided within the framework.
Type: Enable a Feature, Parameter Pattern
Area: Tools
Participants: icon, cursor, manipulatingCursor, eventTable, toolName, DefaultTools, Reader, Figure, Command
Uses: Incorporate Tools, Choose Figure, Choose Reader, Choose Command, DefaultEvents
Changes:
fill in icon, cursor, manipulatingCursor, toolName
DefaultEvents [eventTable]
repeat as needed
Figure = Choose Figure[ChosenFigure]
Reader = Choose Reader[ChosenReader]
Command = Choose Command [ChosenCommand]
eventTable add (Figure, Reader, Command)
DefaultTools add (toolName, eventTable, icon, cursor, manipulatingCursor)
Incorporate Tools[ChosenTools = (Tool, toolName)]
Comments:
When the tool is used, it finds the Figure it was used on and a Reader for the type of input received, and then executes the associated Command.
If there is no entry for a particular type of Figure, an entry for that Figure's superclass will be consulted.

Figure 1: New Tool Type Hook

in HotDraw include the selection tool for moving and manipulating figures, the deletion tool for removing figures, and many tools for drawing different types of basic figures such as circles or rectangles.

RCIM is an aid to corporate strategic planning. It manages several alternative and/or interrelated plans consisting of the goals, objectives and action plans of the business. The plans can be represented as diagrams. Since HotDraw is a framework for manipulating and displaying diagrams it proved to be a good platform upon which to build a graphical display application for the diagrams. These diagrams are intended to be integrated with the existing forms-based editor for RCIM which is also written in Smalltalk. We used the capabilities for displaying figures and animation while adding support for limited automated layout of diagrams. While the display application does not make use of the tools, HotDraw's support for tools and figure manipulation allows us to extend the display application into a graphical editor with little difficulty.

One common problem in HotDraw is choosing which tools to include in an application. While HotDraw includes a number of pre-defined tools that can be included in any application, it is often necessary to build custom tools. The `New Tool Type` hook in Figure 1 shows how a new tool can be created by simply filling in a set of parameters. The parameters are the name of the tool (`toolName`), the `icon` used to represent the tool in the toolbar, the `cursor` for the tool, the `manipulatingCursor` that the tool uses when a figure is selected and being manipulated, and the `eventTable` which defines the actions of the tool. Most of the parameters are filled in by simply supplying appropriate values. To fill in the `eventTable`, the hook indicates that the `Reader`, which is a class responsible for user input, the `Figure` to operate on, and the `Command`, which is a separate class that performs some action on the `Figure`, have to be chosen for each entry using other hooks. For example, to get the `Figure`, the `Choose Figure` hook is invoked and the class that the user selects as `ChosenFigure` in the `Choose Figure` hook becomes the `Figure` participant in the `New Tool Type` hook. After the entries are complete, all of the parameters are put into the dictionary of `DefaultTools` as a new type of tool. Finally, the `Incorporate Tools` hook is used to ensure that an instance of the new tool is integrated into the application. Since it affects tools, the hook is defined to be in the area of `Tools` for reference purposes. Since the hook involves filling in parameters and does not involve specifically adding any new classes or services to the framework, its type is defined to be enabling a feature with a parameter pattern.

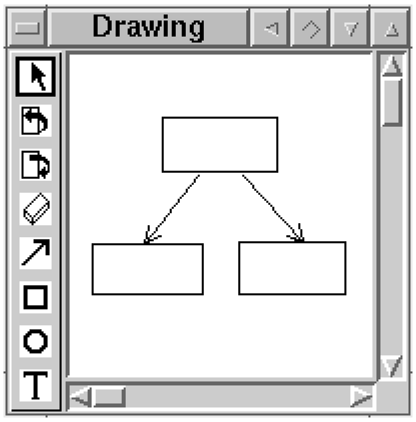

Figure 2: Default Implementation of the HotDraw Application Framework [10]

The problem of providing the needed tools for an application built from HotDraw is not solved completely by the `New Tool Type` hook. As shown in Figure 3, the problem is associated with a number of hooks which depict different ways in which the problem can be solved. The `Select Existing Tools` hook allows the selection of tools that are supplied with the HotDraw framework. The `New Tool Type` hook shows how new tools can be constructed simply by filling in a set of parameters.

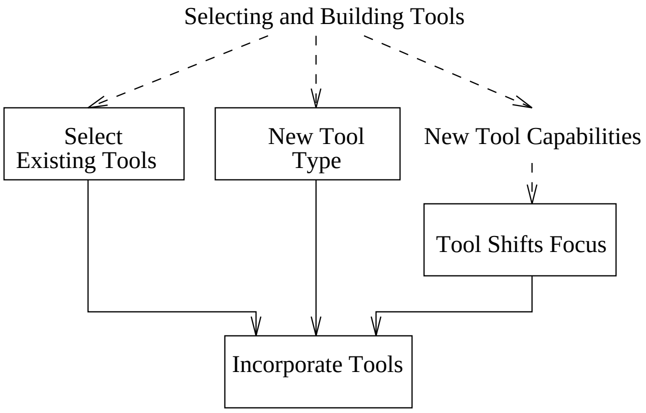

Figure 3: Relationship Amongst Some Hooks for Tools in HotDraw

There may be occasions when the previous two hooks are not sufficient and tools with new capabilities are needed. One such hook is shown for allowing the tool to perform some action when its focus shifts between figures. All of these hooks are not exclusive; all of them can be used together for the same application. Finally, all three hooks rely on the `Incorporate Tools` hook to actually link the new tools into the application being developed.

CHARACTERIZING HOOKS

There are currently two axes along which we have chosen to categorize hooks: the method of adaption used and the level of support provided. Each hook has a method of adaption for fulfilling the requirement, such as enabling a feature within the framework, removing a default feature, or adding a completely new feature. The method of adaption used is distinct from the level of support provided for the changes. For example, removing a feature, such as the use of tools in HotDraw, may be well-supported and simply require turning off a switch in the framework, or may be less well-supported and require the modification of one or more methods.

A hook's method of adaption quickly gives an application developer an idea of what the hook does. Its support type indicates how difficult it may be to use. The types can also serve as a basis for locating a desired hook. For example, if developers need to extend the framework in a particular area, they could retrieve all of the hooks that add a feature to that part of the framework. Each type indicates the issues that both the hook writer and the application developer must consider. For example, removing a feature such as the use of tools in HotDraw will often have an impact on other features of the framework, such as the ability for the user to manipulate the figures in a diagram.

Method of Adaption

There are several ways that a developer can adapt a framework and each hook uses at least one of these methods.

1. Enabling a feature
2. Disabling a feature
3. Replacing a feature
4. Augmenting a feature
5. Adding a feature

Enabling a feature that exists within the framework but is not part of the default implementation is one common means of adapting a framework. Hooks described by framework builders will typically be for enabling features. All structural and behavioral changes and obligations required to enable the feature such as the type of parameters to fill in, or the set of options to be chosen from will be provided within the hook. The constraints imposed by using the feature, such as excluding the use of another feature, are also contained in the hook.

Disabling a feature may be required if the default implementation of the framework has some unwanted properties. This is different than simply not choosing to enable a feature. If the removal is supported cleanly, then there may be little difficulty, although any configuration constraints imposed by the removal should still be noted.

Figure 4 shows an example hook that disables a feature. No manipulation of figures is required in the RCIM graphical display application so there is no need for the incorporation of tools. However, the tools are a default feature of the HotDraw framework and so have to be actively removed. For the purposes of this simple example, the hook shows one means of preventing the tools

Name: Disable All Tools
Requirement: The application does not require the use of tools for any purpose.
Type: Disable a Feature, Open-Ended
Area: Tools
Participants: DrawingEditor, NewDrawingEditor
Changes:
subclass NewDrawingEditor of DrawingEditor
NewDrawingEditor.openOn:withLabel: copies
DrawingEditor.openOn:withLabel:
remove code
'aToolPalleteView := ...' ,
'container add: aToolPaletteView ...'
replace 'leftOffset: 4 + ...'
with 'leftOffset: 0'
Constraints:
SelectionTool is required for movement of figures using the mouse

Figure 4: Disable Tools Hook

from being incorporated into an application, although not necessarily the best way. As shown in the hook, a subclass of `DrawingEditor` called `NewDrawingEditor` is created. It does not have to be a new class, any existing subclass of `DrawingEditor` can participate as `NewDrawingEditor`. A copy of the initialization method `openOn:withLabel:` from `DrawingEditor` is put into `NewDrawingEditor` and is modified by removing two lines of code, such as the one beginning with `container add: aToolPaletteView`, and replacing another line. A sample dependency is given in the constraints section, showing that the user will not be able to move the figures using the mouse if the tools are disabled.

Replacing or overriding an existing feature is related to disabling a feature, with the addition that something is provided in its place. Here it is important to describe the interface and behavioral obligations that any replacement must fulfill.

Augmenting existing behavior involves intercepting the existing flow of control, performing some needed actions, and returning control back to the framework. Unlike replacing behavior, augmenting simply adds to the behavior without redefining it. The framework builder can point to places in the control flow where a change to fulfill a particular requirement might be made. The dependencies, if any, and synchronization between existing steps in the flow of control and the new behavior should be described.

The `Tool Shifts Focus` hook in Figure 5 is an example of a hook that augments existing behavior within the framework rather than replacing it. The hook shows how a new `Tool` can perform some action every time a mouse command is issued while it is active. It has to be done in a new subclass since there is no support for the operation within the event tables of the `New Tool Instance` hook. As shown in the changes section, `NewTool` becomes a subclass of `Tool` which adds a new operation called `shiftFocus` and specializes the `mouseCommandFrom:figure:` method to use `shiftFocus`. The synchronization statement shows that `shiftFocus` should be invoked before calling the default implementation of `mouseCommandFrom:figure:` within the `Tool` class which returns control to the framework. The new operation does not have to be a separate method, but can be included directly within `NewTool`'s specialization of `mouseCommandFrom:figure:`. The `Provide Tool Initialization` hook is used to provide a means of instantiating the tool of class `NewTool` and type `toolName`. Finally, the `Incorporate Tools` hook is used to put the new tool into the application.

Adding a new feature or service to the framework is another common adaptation and probably the most difficult to support. Unlike enabling a feature, where the developer is using existing services, possibly in new ways, adding a feature involves adding something that the framework wasn't capable of before. These additions are often done by extending existing classes with new services or adding new classes, and adding new paths of control with the new services. The hook shows what new classes or operations are needed, and indicates where to integrate them into the framework and how they interact with old classes and services. The framework builder may also provide constraints that must be met by the new class or service.

Name: Tool Shifts Focus
Requirement: A new type of tool is needed that performs some operation when the selected figure has possibly changed.
Type: Augmenting Behavior, Open-Ended
Area: Tools
Uses: Incorporate Tools, Provide Tool Initialization
Participants: Tool, NewTool, toolName
Changes:
subclass NewTool of Tool
NewTool.mouseCommandFrom:figure: specializes
    Tool.mouseCommandFrom:figure:
new operation NewTool.shiftFocus
synchronization
    (NewTool.shiftFocus,
        Tool.mouseCommandFrom:figure:)
    in NewTool.mouseCommandFrom:figure:
fill in toolName
Provide Tool Initialization[NewTool, toolName]
Incorporate Tools[ChosenTools =
    (NewTool, toolName)]
Comments:
NewTool is not guaranteed to be focused on a new figure after every mouseCommandFrom:figure: message. The current figure may regain focus immediately. This hook is useful for text manipulation tools.

Figure 5: Tool Shifts Focus Hook

Level of Support

Another important aspect of hooks is the level of support provided for the adaption within the framework. There are three main levels of support types for hooks.

1. Option
2. Supported Pattern
3. Open-Ended

The *option* level provides the most support, and is generally the easiest for the application developer to use. A number of pre-built components are provided within the framework and the developer simply chooses one without requiring extensive knowledge about the framework. At the *pattern* level, the developer supplies parameters to components and/or follows a well-supported pattern of behavior. The parameters may include instance variables, or abstract object methods that need to be made concrete. Using a pattern hook requires more knowledge about the framework than does using an option hook, but since it is well-supported within the framework, the developer does not usually need to worry about unwanted interactions, or require a deep understanding of the design of the framework. It is at the *open-ended* level that hooks are provided to fulfill requirements without being well-supported within the framework. Open-ended hooks involve adding new properties to classes, new components to the framework, new interactions among components or sometimes the modification of existing code. The knowledge about the framework required to use open-ended hooks is generally greater than with the other two support types. Since they are open-ended, the developer has to be more careful about the effects changes will have on the framework.

Option Hooks
With option hooks the developer has to choose from a set of pre-built components without worrying about their internal workings. This is the black-box approach to frameworks [9]. Most often, components are chosen to enable features within the framework or to replace default components.

The framework may already have several alternative solutions for a requirement which are mutually exclusive. This is a *single option* hook. The alternatives are presented to the developer who must choose a single solution. If the framework has several optional components of the same type which are not mutually exclusive, then the hook is said to be *multi-option*. Several of the components can be chosen and used within a single application. In either case, the hook should describe the options and how the chosen option(s) can be inserted into the framework. Often, this insertion should be obvious and can potentially be handled automatically. If none of the choices are suitable, another hook of a different support type may be provided to allow the developer to produce his/her own solution.

Option hooks are selection oriented. The changes section outlines, in an obvious way, how to choose the components. The changes section also needs to indicate how to incorporate the chosen components into the application. The incorporation can often be handled by another hook, such as a pattern which takes the set of components as a parameter. The constraints section details how the selected components affect the rest of the framework.

An example of a multi-option hook is shown in Figure 6. HotDraw comes with several existing tools which may be suitable for any application being developed from the framework. The existing tools are represented by the set `ExistingTools` which contains the class, name and description of each tool. The `ChosenTools` sequence holds the tools the developer chooses. The process for using the hook is simply to choose the tools desired and then incorporate them using the `Incorporate Tools` hook. Some sample configuration constraints are also provided in this example, showing that the `SelectionTool` is required to allow the user to move figures using the mouse, and that the `BringToFrontTool` and `SendToBackTool` tools should be included as a pair.

Name: Select Existing Tools
Requirement: The application needs a particular tool or set of tools which is already provided as a part of HotDraw.
Type: Enable a Feature, Multi-Option
Area: Tools
Uses: Incorporate Tools
Participants:
  ExistingTools set of (toolClass, toolName, description),
  ChosenTools sequence of (toolClass, toolName)
Changes:
  repeat as needed
    choose t from ExistingTools
    ChosenTools add (t.toolClass, t.toolName)
  Incorporate Tools[ChosenTools]
Constraints:
  SelectionTool is required for movement of figures using the mouse
  Set(BringToFrontTool, SendToBackTool)

Figure 6: Select Existing Tools Hook

Supported Pattern Hooks
If the framework has a pre-defined method of fulfilling a requirement, but the details of the method are application dependent and so have to be filled-in by the application developer, then the hook is a pattern hook. Unlike option hooks, there are no complete pre-defined components to choose from, but support is provided for the feature through parameters to components. Pattern hooks often enable, augment, remove or replace features.

The simplest patterns occur when the developer needs to supply values or parameters to a single class within the framework. The parameters themselves may be as simple as base variables, or as complex as methods or component classes. Some common tasks may require the collaboration of multiple classes, and may also have application specific details. For these, a collaboration pattern is provided which the developer follows to realize the task. The flow of control is already defined within the hook and the developers should not deviate from it or they are no longer using the hook.

Name: Incorporate Tools
Requirement: A custom set of tools is to be used in the application.
Type: Replace a Feature, Parameter Pattern
Area: Tools
Participants: DrawingEditor, NewDrawingEditor, ChosenTools sequence of (toolClass, toolName)
Changes:
subclass NewDrawingEditor of DrawingEditor
NewDrawingEditor.defaultTools overrides DrawingEditor.defaultTools
NewDrawingEditor.defaultTools returns sequence of
forall t in ChosenTools
instance of t.toolClass of type t.toolName
Comments:
The placement and invocation of all of the chosen tools is handled by the framework. The order in which tools appear in the tool bar is the same as the order they appear in the ChosenTools collection.

Figure 7: Incorporate Tools Hook

New subclasses, or methods to be overridden or specialized, or parameters to be filled in can be included in the changes section of a pattern hook. However, the creation of entirely new classes or new operations is generally not allowed. Some behavior of methods may also be included. The constraints section will give details on special conditions placed on the parameters or methods, such as values which are not allowed, as well as the effects that using the pattern has on the rest of the framework.

The `Incorporate Tools` hook shown in Figure 7, indicates how tools are linked into an application. Its participants consist of the `DrawingEditor` class, which is responsible for instantiating the classes necessary to start the application, a `NewDrawingEditor`, and the `ChosenTools` consisting of the class and type of each tool to include. `NewDrawingEditor` becomes a subclass of `DrawingEditor` and overrides the `defaultTools` method. `NewDrawingEditor` does not have to be a new class; an existing subclass of `DrawingEditor` created by the developer can be used here. The parameter to be filled in is the `defaultTools` method, and because it overrides a default in the framework, the method of adaption used is replacing a feature. The hook shows the obligation of the replaced feature, namely that it must return a collection containing an instance of each element of `ChosenTools`. If an existing class is used, the new tools are simply added to those already incorporated into the application. The placement and invocation of the tools is taken care of by the framework.

Open-Ended Hooks
It is unlikely that a framework will provide components or parameters for every possible use. With open-ended hooks, the framework does not provide direct support for the required change, although it may not hinder it either. Open-ended hooks can be used to point to places in the framework that can be changed by a developer to fulfill certain requirements for which the framework builder decided not to provide support, or simply could not provide support. Typically, these changes will be for adding, replacing, removing or augmenting features within the framework, sometimes in ways that the original framework builder never anticipated. If the requirement fulfilled by the changes may be applicable to more than one application and no hook exists for it, then a new open-ended hook can be defined for the framework. New hooks document some of a developer's experience with the framework and show how the framework can be used in new ways.

The changes section of an open-ended hook will often include the introduction of new classes which are not subclasses, the introduction of new operations to classes, and sometimes the removal or replacement of code in addition to what is done in pattern hooks. Since the change is not supported within the framework, the hook often also includes explicit control flow information to show the new paths of control that the hook introduces. The constraints section should indicate the effects that using the hook has on the rest of the framework.

Figure 8 shows an open-ended hook which adds support for allowing individual figures to animate themselves, as was needed in the RCIM graphical display application. The new open-ended hook shows one way in which figures might be animated. The hook shows that `AnimDrawing` and `AnimFigure` are subclasses of `Drawing` and `Figure`. `AnimDrawing` overrides the existing `step` method of `Drawing`. A new method called `step` is then added to `AnimFigure`. The `AnimDrawing` sends the `step` message to all of the `AnimFigures` in order to perform the animation. Additional synchronization between methods is also required. The `changed` method should be called after the `step` method of `AnimFigure`, and the `solveConstraints` method, which updates all graphical constraints placed on figures, should be called after the `step` method of `AnimDrawing` to ensure that the display is updated properly. Finally, the hook indicates that the `Incorporate New Drawing` hook, not included here, should be used to put `AnimDrawing` into the application.

USING HOOKS

Once the correct hook has been found, an application developer uses the hook simply by performing all of the changes within the changes section of the hook in the order given. The problem exists of finding the correct hook to use from the database of all of the hooks for a framework. Developers should not have to read through

Name: Animating Figures
Requirement: Some form of constant animation which is dependent upon the internal state of each figure is needed.
Type: Adding a Feature, Open-Ended
Area: Figures, Animation
Uses: Incorporate New Drawing
Roles: Figure, AnimFigure, Drawing, AnimDrawing
Changes:
subclass AnimDrawing of Drawing
AnimDrawing.step overrides Drawing.step
subclass AnimFigure of Figure
new method AnimFigure.step
control flow
AnimDrawing.step − > all AnimFigure.step
synchronization
(AnimFigure.step, AnimFigure.changed),
(AnimDrawing.step,
AnimDrawing.solveConstraints)
Incorporate New Drawing[NewDrawing = AnimDrawing]
Constraints: All Figures used in AnimDrawing must derive from AnimFigure.
Comments:
What is done inside the step method of each AnimFigure is completely application dependent. The framework will periodically send the step message to AnimDrawing.

Figure 8: Animating Figures Hook

all of the hooks to begin to gain an understanding of the framework and which hooks to use.

There are a number of ways in which the correct hook could be found. First, the area and type characterization for a hook make it easier to search for the needed hook. If, for example, the developer wants to find all hooks related to tools in HotDraw, then the area specification of hooks could be searched to return all of those hooks. The search could be narrowed by including the type of hooks wanted, such as all tool hooks that add new features.

Second, the developer might try searching the requirements section for a specific requirement. However, the developer might want to search for a more general requirement that might be met by several hooks. A set of general requirements or problem descriptions, which can be searched or browsed, can be provided which point to the hooks that satisfy some aspect of that requirement.

If none of the hooks found are appropriate, they will at least point to places in the design of the framework where a change most likely can be made to fulfill the requirement. As a third alternative, each part of the design can be connected to the hooks that extend or modify it. Particularly when adding new features, the developer might start by looking at the part of the design to which the feature will be added, and then look to see if there is a hook which accomplishes the desired task.

Clearly a browser which provides hooks as an alternative view to the design of the framework is needed. However, this paper focuses on introducing the concept of hooks and how they are described, and the exact nature of such a browser is an issue for future consideration.

As can be seen, the presence of hooks does not preclude the need for other types of documentation. Some means, such as motifs, of documenting the purpose of the framework and the general problems it solves is needed, which can then point to hooks that solve aspects of the problems. The hooks themselves describe how the framework should be adapted. Examples, both of applications and of the use of individual hooks are also important for understanding. Finally, hooks will never be able to describe all changes that might be made to a framework. An application developer will need to look at the design of the framework in order to obtain more information. The design could be represented by existing techniques such as design patterns or more traditional methods such as the Booch method [2] or OMT [15].

Hooks are an important part of this overall framework documentation. Without the documentation of the intended use of the framework provided by hooks, application developers are more likely to cause errors by incorrectly using the framework. They may also expend effort extending a framework to provide a solution to a problem that the framework already solves simply because they were unaware of the support.

INITIAL EXPERIENCE

Our work on hooks is grounded in real world experience. We are in the fortunate position of being able to validate the concept of hooks using the existing HotDraw framework and through developing frameworks as a part of the Size Engineering Application Framework (SEAF) project [16].

An investigation of the HotDraw framework and several example applications provided with the framework was made. We found that even with a well-designed framework, a large initial investment in time is needed to learn how to use the framework. A prototype application was then developed from the HotDraw framework to discover what useful hooks might exist and to gain some insight into how to describe hooks, particularly how much information should be provided within a hook. The RCIM graphical display application and the graphical editor program make use of many of the features of HotDraw such as Figure manipulation and animation and also extends the framework into areas such as automated layout.

Our notion of hooks is also being evaluated in the context of the SEAF project, a joint effort with Teledyne Fluid Systems, to develop an engineering tool for the sizing and selection of pressure relief valves. This project involves two orthogonal application frameworks. One deals with persistent object management and user interface support (as two sub-frameworks that collaborate extensively), while the other deals with support for engineering calculations. The frameworks total about twenty thousand lines of code, and required about a person/year of work by very experienced developers to reach the alpha stage.

The hook concept was developed in parallel with the work on the application frameworks. We find that the idea of a hook influences the framework architecture and implementation by forcing the framework designer to clearly articulate the interface to their framework. A difficult to document hook or one whose type is fuzzy is an indicator of a weakness in the framework architecture.

Hooks also provide a focus for the quality efforts of a project. Since hooks are important points of access to the framework, the evaluation of their design, implementation, and evolution should be among the high priority quality activities.

Finally, like others, we have observed that the initial effort of developing the framework must be done by the same group that is developing the first application. But because the SEAF project involves a number of related applications, we have an ideal experimental situation for validating the hook concept, especially in terms of the documentation requirements. Developers that are initially unfamiliar with the frameworks will be using them to build actual production applications.

SUMMARY AND FUTURE WORK

In this paper, we introduced the idea of a hook as a means of documenting the intended use of a framework and providing strong guidance for actually building applications from the framework. We also provide a characterization of hooks according to the method of adaption used to fulfill a requirement and the level of support provided in the framework.

By focusing on the intended use of a framework, rather than the design of the framework, hooks are able to provide a more structured and uniform specification than textual narrative of the changes needed to fulfill a requirement that an application might have. This can be an aid in the analysis and provision of tool support for hooks. The framework builders' knowledge and other application developers' experiences with the framework can be captured in hooks and made available to application developers. Short and understandable solutions are provided to small problems, with only sufficient detail to allow the changes to be made. More complex problems can be supported by linking and using several hooks together. However, hooks are not meant to be the only documentation needed for a framework. Examples, purpose and design documentation are still needed.

Five methods of adaption for fulfilling a requirement and three levels of support are used to categorize hooks and provide a two-dimensional type system. Framework builders can use this knowledge to help decide what types of hooks to use in a framework and what sort of information is needed to describe the hooks. Application developers can use this knowledge to gain a quick, general understanding of what a hook does and how much effort is required to use it. Further, the introduction of a type system for hooks helps to manage the potentially large number of hooks that a large framework may have. Developers can query the database of hooks using the type and area of the hook to better find the hooks they need.

With their structure and type system, hooks provide a basis for future work in the area of tool support for the selection of hooks as well as the use of individual hooks, and in the area of evolution of frameworks. A browser for finding the most appropriate hook is needed. Such a tool will allow developers to query the database of hooks based on the requirement, type and area of hook to find the one needed. It would also be desirable to connect hooks to the design of the framework; however, how hooks can be connected to existing design notations is an open issue.

The use of individual hooks may be partially automated. The developer might select a hook and have the system guide them through the use of it. For example, if one of a number of options must be chosen, then the set of options can be presented to the developer. The developer chooses an option and the system performs any changes that it can to accommodate that option, prompting the developer to perform the rest of the changes as required. Some automated tools already exist for individual frameworks, such as the ToolBuilder program which aids in the construction of tools in HotDraw. Hooks provide a means of specifying changes to a level at which building such automated tools should be made easier, particularly for option and pattern hooks.

Hooks may provide an indicator for when a framework could evolve. If the hooks provided for the framework are not useful to developers then the framework may not be meeting its intended purpose and needs to be modified or evolved. A framework might also evolve to incorporate more support for commonly used hooks.

An open-ended hook may evolve into a pattern hook when enough restrictions are placed upon it. A pattern hook may evolve into an option hook if enough common instances of the pattern are included within the framework. In particular, if a number of open-ended hooks are commonly used, then the framework might evolve to incorporate the new features the open-ended hooks provide.

REFERENCES

[1] K. Beck and R. Johnson. *Patterns Generate Architectures.* In Proceedings of ECOOP'94, Bologna, Italy, 1994, 139-149.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[3] K. Brown, L. Kues and M. Lam. *HM3270: An Evolving Framework for Client-Server Communications.* In Proceedings of the 14th Annual TOOLS (Technology of Object-Oriented Languages and Systems) Conference, Santa Barbara, CA, 1995, 463-472.

[4] R. H. Campbell and N. Islam. *A Technique for Documenting the Framework of an Object-Oriented System.* In Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems, Paris, France, 1992.

[5] R. H. Campbell, N. Islam, D. Raila and P. Madany. *Designing and Implementing Choices: An Object-Oriented System in C++.* Communications of the ACM, 36(9), Sept. 1993, 117-126.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[7] D. Gangopadhyay and S. Mitra. *Understanding Frameworks by Exploration of Exemplars.* In Proceedings of 7th International Workshop on Computer Aided Software Engineering (CASE-95), Toronto, Canada, 1995, 90-99.

[8] R. Helm, I. M. Holland and D. Gangopadhyay. *Contracts: Specifying Behavioral Compositions in Object-Oriented Systems.* In Proceedings of OOPSLA'90, Ottawa, Canada, 1990, 169-180.

[9] R. Johnson and B. Foote. *Designing Reusable Classes.* Journal of Object-Oriented Programming, 2(1), June/July 1988, 22-35.

[10] R. Johnson. *Documenting Frameworks Using Patterns.* In Proceedings of OOPSLA'92, Vancouver, Canada, 1992, 63-76.

[11] G. E. Krasner and S. T. Pope. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, 1(3), August-September 1988, 26-49.

[12] R. Lajoie and R. K. Keller. *Design and Reuse in Object Oriented Frameworks: Patterns, Contracts, and Motifs in Concert.* In Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences, Montreal, Canada, 1994.

[13] D. B. Lange and Y. Nakamura. *Interactive Visualization of Design Patterns Can Help in Framework Understanding.* In Proceedings of OOPSLA'95 Austin, TX, 1995, 342-357.

[14] *VisualWorks Cookbook.* Release 2.5, ParcPlace-Digitalk Inc., Sunnyvale, CA, 1995.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[16] *SEAF Project*, unpublished (http://www.cs.ualberta.ca/~softeng/SEAF/project.html).

[17] H. A. Schmid. *Creating the Architecture of a Manufacturing Framework by Design Patterns.* In Proceedings of OOPSLA'95 Austin, TX, 1995, 370-384.

[18] L. Tse. *A Responsive Corporate Information Model.* Ph.D. Thesis, University of Alberta, Edmonton, AB, Canada, July 1996.

[19] M. Vlissides and M. A. Linton. *Unidraw: A Framework for Building Domain-Specific Graphical Editors.* ACM Transactions on Information Systems, 8(3), July 1990, 237-268.

[20] A. Weinand, E. Gamma and R. Marty. *ET++ - An Object-Oriented Application Framework in C++.* In Proceedings of OOPSLA'88, San Diego, CA, 1988, 46-57.