Reusing Application Frameworks Through Hooks

Garry Froehlich, H. James Hoover, Ling Liu, Paul G. Sorenson

Department of Computing Science

University of Alberta

Edmonton, AB. T6G 2H1

{garry,hoover,lingliu,sorenson}@cs.ualberta.ca

Abstract

Object-oriented frameworks can be complex. Often, considerable time is needed to understand the framework, including not only what the framework does, but also how to use it. Techniques for documenting the design of a framework already exist, but less work has been done on documenting how to use the framework. We have developed the notion of hooks as a means of representing knowledge about the places in a framework that can be adapted by application developers to produce an application from the framework and have applied the concept to the SEAF (Size Engineering Application Framework) project. By describing the hooks, implicit knowledge about how to reuse the framework is made explicit and open to study, refinement or use. Having this knowledge available can make frameworks easier to reuse, since developers need not spend the time deriving the knowledge themselves. With hooks, a template is provided to describe the hooks and to capture all of the relevant knowledge in a form that provides guidance to application developers. This paper discusses the model of hooks as a means of capturing knowledge about how to reuse the framework, reports on our experience with hooks in the SEAF project and gives some initial assessment of the notion of hooks.

X.1 Introduction

An application framework provides a generic design within a given domain and a reusable implementation of that design. An object-oriented application framework presents its generic design and its reusable implementation of a system through a set of classes and their collaborations (Beck and Johnson 1994). Applications are built from frameworks by extending or customizing parts of the framework, while retaining the original design.

Frameworks solve larger-grained problems than individual components, making the effort of finding and reusing them much more cost effective than for small components. Due to the potentially large size and complexity of frameworks, even for well-designed frameworks, the time required to understand how to use the framework can be significant. For the use of a framework to be cost effective, it should take substantially less time to understand and reuse a framework than to build an equivalent application without the framework.

To aid in framework reuse, we have developed the notion of *hooks* as a means of representing knowledge about the places in a framework that can or should be changed or augmented by application developers to produce an application from the framework. Hooks focus on how the framework is intended to be used, and 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. Each hook captures the relevant knowledge of some potential use of a framework in

a form (template) that provides guidance to application developers. 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.

There are several novel contributions of the hooks approach. First, hooks present a structured and uniform method to allow the framework builder to be more precise about how the framework is intended to be used. By describing the hooks, implicit knowledge about how to use the framework is made explicit and open to study, refinement or reuse. Having this knowledge available can make frameworks easier to reuse, as developers need not spend the time deriving the knowledge themselves. Having this knowledge available can also lead to better designed frameworks, as it gives framework builders a means of examining and refining how the framework can be used. Second, the type system for hooks catalogues many of the ways in which changes can be made to a framework and how they can be supported.

Last, but not least, the hooks model provides a number of characterizations for each hook (e.g., the requirement, the use area, the adaption type, the participants), which makes it easier for application developers to find the hook they need from the database of all the hooks for a framework. This is especially useful for frameworks having many places of potential reuse or that are open for refinement. Application developers will not have to read through all of the hooks to begin to gain an understanding of the framework and which hooks to use.

Our notion of hooks is being evaluated in the context of the SEAF (Size Engineering Application Framework) project, a joint effort with Teledyne Fluid Systems - Farris Engineering, to develop an engineering tool for the sizing and selection of pressure relief valves (SEAF Project 1997). 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 rest of the paper proceeds as follows. In Section 2 we briefly overview the related work and the background research on techniques for documentation of application frameworks. We discuss the notion of hooks as a means of capturing knowledge about how to reuse the framework and overview the basics of the hooks model in Section 3. Section 4 presents our experience with hooks in the SEAF project and gives some initial assessment of applying the hooks model. We present conclusions in
Section 5.

X.2 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 (1992) proposes that framework documentation needs at least three parts: one describing the purpose of the framework, one describing how to use the framework, and one describing the design of the framework.

Several approaches to framework documentation have focused on the design of the framework. Some approaches that have been used are: providing multiple views of the design, using exemplars, and using design patterns. Campbell and Islam (1992) propose a six view approach to documenting the design which includes documenting the classes but also documenting the relationships between classes. Exemplars (Gangopadhyay and Mitra 1995), which consist of a concrete implementation provided for all of the abstract classes in the framework along with their interactions, provide another means of understanding the design of frameworks.

Design patterns (Gamma et al. 1995) are a popular means of both building and describing frameworks. Beck and Johnson (1994) have used design patterns to help show how the architecture of the HotDraw framework

is derived. Since design patterns are often flexible and extensible, using them to design a framework can lead to more flexible and extensible frameworks. Design patterns are also useful for describing the design of the framework and showing the decisions that were made regarding the design. 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.

Pree (1995) uses metapatterns to both design and document the hotspots of a framework. The hotspots are the flexible or incomplete places within a framework. Each hotspot tends to have several hooks within it. Many design patterns can be broken down into a group of metapatterns. The metapatterns document the design of a particular hot spot in terms of the relationship between template classes that encapsulate some standard functionality within the framework and hook classes to be filled in by application developers that have methods for extending that functionality.

All of the design-oriented approaches provide excellent means of documenting the design. They were, however, not targeted for explicitly capturing the purpose of a framework, nor how the framework should be used. For example, they show classes and their interactions, but not the ways in which those classes can be used or adapted to build applications from the framework.

Less work has focused on the purpose and intended use of frameworks. Two such approaches are cookbooks and patterns. The cookbook in (Krasner and Pope 1988) 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 (Visualworks Cookbook 1995) 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. Johnson's patterns (Johnson 1992) 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. All of these techniques rely on narrative descriptions that may be imprecise or incomplete.

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, but they are meant to augment, rather than replace other types of documentation. Examples and design documentation are important in terms of learning and using a framework. Examples capture specific uses of the framework. Design documentation presents the design and 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.

X.3 The Hooks Model: An Overview

A hook is a point in the framework that is meant to be adapted in some way such as by filling in parameters or creating subclasses. Each hook description documents a problem or requirement that the framework builder anticipates an application developer will have and provides guidance about how to use the hook and fulfill the requirement. Each description typically focuses on a small requirement. For more complex problems a group 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 adhered to and any effects upon the framework that the hook will impose. Only the information needed to solve the problem is provided within the hook. Developers are then able to quickly understand and use the hook. 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.

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, refactoring an existing framework is beyond the scope of hooks.

X.3.1 Hook Descriptions

Each hook description is written in a specific format made up of several 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 and help to ensure that all relevant information is included. The format in which hooks are described helps to organize the information and make the description more precise and less ambiguous than pure English narrative. Each hook description consists of the following parts:

Name: a unique name, within the context of the framework, that identifies the hook.

Requirement: a textual description of the problem the hook is intended to help solve. The framework builder anticipates the requirements that an application will have and describes hooks for those requirements.

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 that outlines the changes to the interfaces, associations, and control flow amongst the components given in the participants section.

Constraints: limits imposed on the hook, or on the use of the hook.

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. Concrete examples of hook descriptions will be given in Section 4.

X.3.2 Characterizing Hooks

Hooks are currently categorized by the following two characterization types (recall the Type description of a hook in Section 3.1). The first characterization, the *method of adaption*, indicates the type of change that the hook uses, such as enabling a feature within the framework, removing a default feature, or adding a completely new feature. The second characterization, the *level of support*, indicates how much support the hook provides for the adaption within the framework. The method of adaption used is distinct from the level of support provided for the changes. For example, removing a feature, 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 the hook may be to use. Each type indicates the issues that both the hook writer and the application developer must consider. For example, removing a feature will often have an impact on other features of the framework.

X.3.2.1 Method of Adaption

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

Enabling a feature

Disabling a feature

Replacing a feature

Augmenting a feature

Adding a feature

The two most common methods are enabling a feature and adding a feature, and these are examined in this chapter. The others are more fully described in (Froehlich et al. 1997). *Enabling* a feature involves activating features that are a part of the framework but may not be a part of the default implementation. Hooks of this type often involve using pre-built components that come with the framework that may be parameterized. The hook needs to detail how to enable the feature, such as which components to select for inclusion in the application, which parameters to fill in, or how to configure a set of components. The constraints imposed by using the feature, such as excluding the use of another feature, are also contained in the hook.

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, 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 and may limit the interfaces that the new class can use to interact with the framework.

X.3.2.2 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.

Option

Supported Pattern

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 or more without requiring extensive knowledge about the framework. Most often, components are chosen to enable features within the framework or to replace default components. The hook describes 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.

At the *pattern* level, the developer supplies parameters to components and/or follows a well-supported pattern of behavior. 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. 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.

In the next section we report our experience with the hooks model in the SEAF project and illustrate the roles of hook characterization types in guiding the reuse of application frameworks by examples taken from our experience with SEAF.

X.4 Hooking into SEAF

The Size Engineering Application Framework (SEAF) project is focused in part on the re-engineering and re-development of the Size Master-Plus engineering software package for sizing and selecting pressure relief valves. The project involves the collaboration of Teledyne Fluid Systems - Farris Engineering in Edmonton and researchers in the Software Engineering Research Laboratory at the University of Alberta. To support current and future product development, two frameworks have been built. The first is the user interface and persistence framework which is responsible for coordinating the interaction between an application's user interface, the underlying database of information, and actual calculations embedded in the application. The second is the engineering framework which provides a worksheet model that guides the engineer through a complex calculation, reminding them of important steps, ensuring that key decisions and sub-calculations are made in the proper order, and recording the process for future review. The two frameworks are fully integrated and together form the basis for developing future size engineering packages. The hooks model was developed in parallel with the SEAF application frameworks. As a test of the capabilities of the hooks model, we chose an interesting portion of the engineering framework in which to retroactively apply the model.

X.4.1 The Engineering Application Framework

The core of the SEAF engineering framework is a hierarchically organized set of calculations called a *worksheet*. Each worksheet captures all of the possible calculations that are relevant to the problem domain. The purpose of the framework is to help the user fill in the parameters of the worksheet and then perform the calculations. In this sense, each engineering tool is a very sophisticated spreadsheet, where each cell can contain an entire spreadsheet. But the framework provides more, in the form of *wizards* that encapsulate common design workflows through the worksheet.

Each wizard provides a view on the worksheet that exposes only the data that is necessary to the task at hand. However a worksheet can have a number of wizards active concurrently, and they could share common data. Thus, one function of the engineering framework is to support concurrent access to common data. In other words, we have a small database manager inside each application that manages access to the worksheet contents. Because it is simple, yet rich, we will use this aspect of the framework to illustrate hooks.

X.4.1.1 Worksheets and Data

A worksheet uses *calculations* to manipulate *data elements*. No calculation is allowed to directly manipulate a data element, it must always go through an *accessor*. Accessors allow controlled reading and writing of properties of the data element. Accessors also mediate database activities like write locking, and the attachment of handlers for *change events*.

The relationship between data elements and accessors is essentially the same as between the subject and observer in the observer design pattern described in Gamma et al. (1995). The difference is that observers must communicate with the data through the accessors. This controlled interface enforces certain framework policies such as no fetching if invalid data, no write access to locked data, etc.

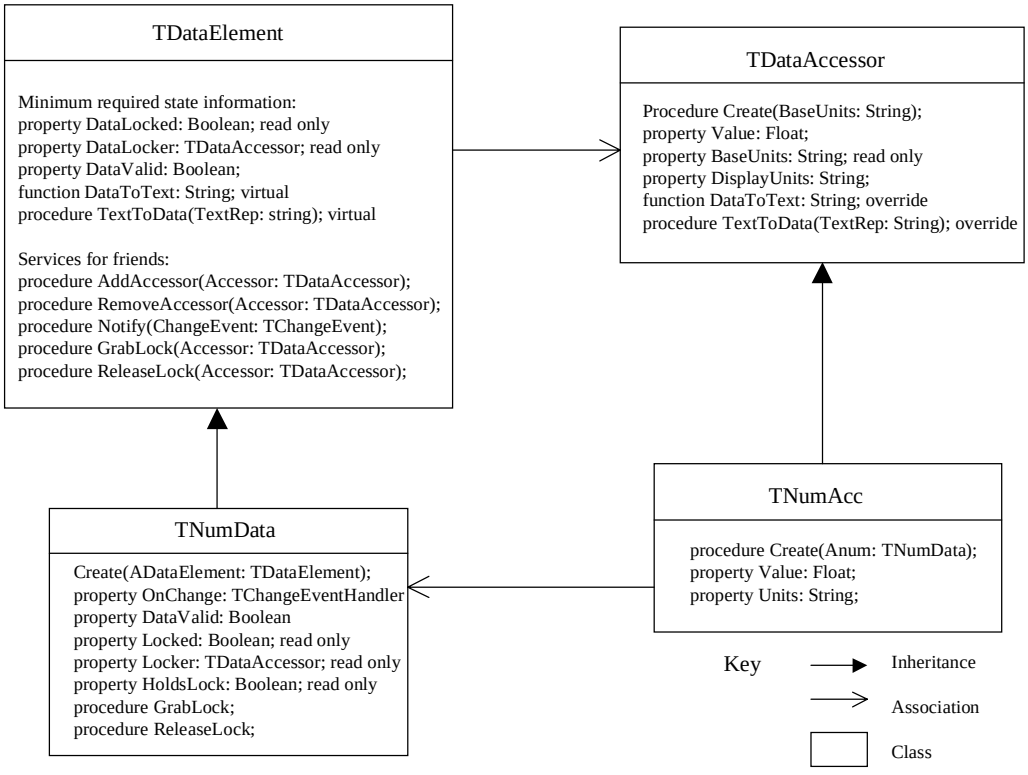

Figure X.1 Data element and accessor pattern.

Figure X.1 gives a 'heavily sanitized' UML (Booch et al. 1996) class diagram for this part of the engineering framework. It contains the two base classes TDATAELEMENT and TDATAACCESSOR. The data

element and accessor classes are always friends. The class diagram contains a simple dimensioned numeric data element, TNUMDATA, and its accessor TNUMACC. A TNUMDATA is a floating point number plus a base dimensional unit (e.g. meters). It also has the notion of display units that are compatible with the base units (e.g. feet). TNUMDATA and TNUMACC are produced by using the hooks given later in this section.

Although not illustrated, there can be many different types of accessor to the same data element, perhaps for enforcing restricted access, or increased functionality. For example one might want an accessor that did conversions from one unit to another.

We use *properties* (as in Eiffel or Object Pascal) to encapsulate the get and set methods for attributes. Unless qualified, properties are read/write. We have omitted many features from the class diagram, such as basic aspects like the identity and structural context of the data element, and the callbacks that lock holders get when another accessor wants them to release the lock.

All data elements support basic locking services, conversions to and from text representations, a change notification mechanism, and the important engineering notion of whether the data element actually contains valid data. This is all provided by TDATAELEMENT. Access to these basic services is provided by TDATAACCESSOR.

Change events are propagated via the ONCHANGE property of a TDATAACCESSOR. The engineering framework can pass change information to the handler, for example the magnitude of the change, or a list of all the components in an aggregate that changed.

Orthogonal to the description of the design, the description of the hooks shows how the classes and collaborations within the design can be extended or modified to produce an application from the framework. The hooks show how the framework is intended to be used. One of the requirements that we anticipate application developers using the framework will have is the need to create a new type of data element and its associated accessor, such as TNUMDATA and TNUMACC in Figure X.1. There are actually three hooks involved in defining a new data element. The main hook, NEW DATA ELEMENT uses two other hooks, NEW STATE and NEW ACCESSOR, to detail all of the actions required to define the new type.

```
Name:  New Data Element
Requirement:  A new type of data is required for calculations.
Type:  Enabling, Parameter Pattern
Area:  DataElement
Uses:  New Accessor, New State
Participants:  TDataElement, // The data type base class.
     NewDataElement  // The new data type to be created.
Changes:
     new subclass NewDataElement of TDataElement
     NewDataElement.Create() extends TDataElement.Create()
     New State[DataElement = NewDataElement]
     New Accessor[DataElement = NewDataElement]
     NewDataElement.DataToText overrides TDataElement.DataToText
          returns String
     NewDataElement.TextToData overrides TDataElement.TextToData
```

There are only two participants in the NEW DATA ELEMENT hook: TDATAELEMENT that is the base class from which the new type will be derived, and NEWDATAELEMENT that represents the new data element. NEWDATAELEMENT does not correspond to any existing class within the framework, but merely represents the new class to be created. When the hook is used, NEWDATAELEMENT is replaced with the actual name of the new class. These participants are used in the changes section of the hook. First, the new subclass is created and a construction method CREATE defined for it. Next, the NEW STATE and NEW ACCESSOR hooks are used to create the actual data

fields within the data element and the accessor for the data element. Finally, the methods DATATOTEXT and TEXTTODATA are overridden to provide a means of serializing the data element for the purpose of storing and retrieving the data type in persistent storage.

When developers use this hook, they are essentially filling in a template, or in other words, providing parameters to customize a standard pattern. In this case, the parameters are methods to be overridden and the results of other hooks. For this reason, the hook is said to have the type pattern. The method of adaption is enabling as this particular hook uses the existing capabilities of the framework. The actual extensions are encapsulated in another hook, NEW STATE.

```
Name: New State
Requirement:
     New state information is needed for a type of data element.
Type: Adding, Open-Ended
Area: DataElement
Participants: DataElement
Changes:
     repeat as necessary
          new property DataElement.newproperty
```

The NEW STATE hook is used by the NEW DATA TYPE hook to actually define the data fields for the new data element type. The fields are called properties that can be either base (simple variables) or calculated values. As many of these properties can be added as desired. As with the NEW DATA ELEMENT hook, this hook operates on data elements and so refers to the DATAELEMENT area of the framework. The hook is open-ended as there is no limit to the number or type of properties that can be added to the class. Since new properties are being added, the method of adaption is adding.

For every new data element that a developer creates, one or more new accessor must be created as well. The NEW DATA ELEMENT hook uses the NEW ACCESSOR hook to show how to create the new accessor.

```
Name: New Accessor
Requirement:
     A new accessor is needed for a new type of data element.
Type: Adding, Pattern
Area: DataElement, Accessor
Participants: TDataAccessor, NewAccessor, DataElement, TDataElement
Changes:
     new subclass NewAccessor of TDataAccessor
     NewAccessor.Create(DataElement) extends
     TDataAccessor.Create(TDataElement)
     repeat as necessary
          new property NewAccessor.newproperty where
          read of newproperty maps from set of DataElement.property
          write of newproperty maps into set of DataElement.property
```

The NEW ACCESSOR hook also uses the adding method of adaption, since it adds new properties to the accessor class. However, the choices of properties are limited to ones reflecting the actual properties within the data element that the accessor is being created for. The statement for adding the properties is given in repeat statement of the changes section. New properties of NEWACCESSOR must map to and from the properties that are being accessed in DATAELEMENT.

X.4.1.2 Using Accessors

Figure X.2 is a collaboration style diagram that shows a typical use of data elements and accessors. The worksheet contains a local data element TEMP, and a subcalculation that contains a local data element X. An accessor is needed for every scope in which we wish to use the data. Thus some accessors cross structural boundaries. For example, TEMP has three accessors: one locally in the main body of the worksheet, one in a subcalculation, and one external to the worksheet in a user interface form. The notes on the figure illustrate how the various services are used.

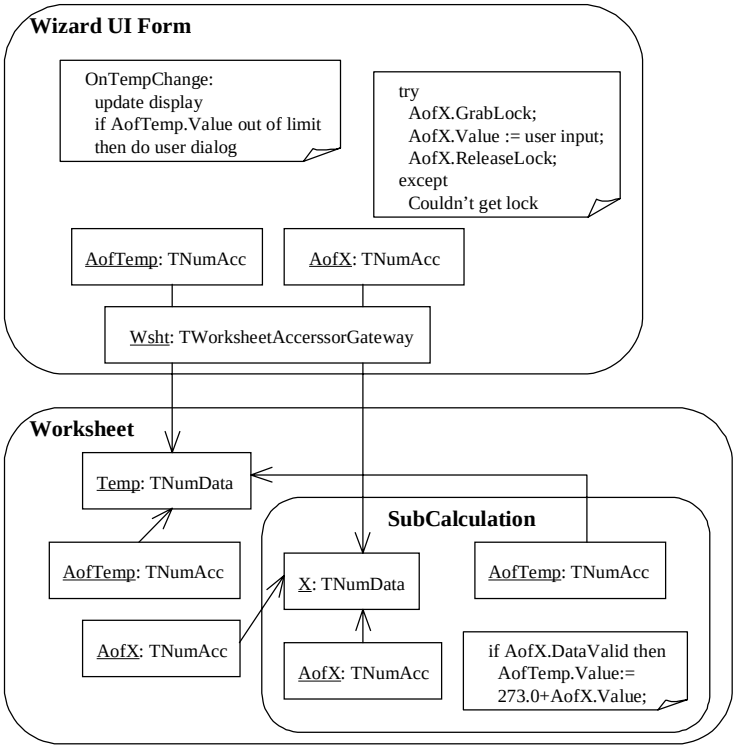

Figure X.2 Example of using the data element and accessor pattern.

Within the framework, accessors have two distinct roles. Inside a worksheet the accessors and change events drive the evaluation mechanism. We use spreadsheet semantics, that is, when a data element changes it triggers the reevaluation of all downstream data elements. Unlike spreadsheets, worksheet calculations cannot contain cycles, so their updates always terminate.

The ACCESS DATA ELEMENT hook is provided for connecting the users of data (such as the form) to the actual data elements. The hook involves the collaboration of several classes, the DATAELEMENT, the actual ACCESSOR and the CALLER that wishes to access the data. The collaboration is a standard pattern built into the framework and so the hook is an enabling pattern.

```
Name: Access Data Element
Requirement:
     A class needs access to an internal data element and needs to be
notified when the data element changes.
Type: Enabling, Collaboration Pattern
Area: Accessor
Participants:
     DataElement, // The data to be accessed.
     Accessor,    // The mediator between the DataElement and the caller.
     Caller       // The class that needs notification of changes.
Changes:
     Caller -> Accessor.Create(DataElement)
          // Create accessor and add to accessor list.
     new operation Caller.Update  // Do actions upon notification here.
     Accessor.Onchange -$>$ Caller.Update <provided>
     Caller -> read and write Accessor.Property
Constraints:
     Accessor and DataElement are of compatible types.
```

The actual changes specified by the hook description involve the CALLER creating the new ACCESSOR for the DATAELEMENT which automatically adds the ACCESSOR to the DATAELEMENT'S accessor list. To be notified of changes, the CALLER must provide an UPDATE method that will be called by the ACCESSOR. The UPDATE method can be given whatever name desired (i.e. ONTEMPCHANGE as in Figure X.2). To retrieve or modify values within the DATAELEMENT, the CALLER must use the property access methods provided by the ACCESSOR. One constraint is placed on the hook, stating that the actual ACCESSOR and DATAELEMENT classes used in the hook must be of compatible types (i.e. TNUMDATA and TNUMACC must be used together).

Outside the worksheet, the accessors mediate the views and controllers of the worksheet. Thus in Figure X.2, the wizard GUI form is notified of changes in TEMP, and also attempts to alter the value of X. Accessors from outside the worksheet need to first access the worksheet before they can access its components. In this case, we do not want them to have the same change notification behavior as the accessors inside the aggregate. They should only have changes signaled when the aggregate as a whole has completed changing and is in a consistent state. For example, once the worksheet change events have ceased causing calculation reevaluation, then the change event is propagated to the accessor outside the worksheet. (In fact, there may be no change event propagated because the data element returned back to its starting value.)

The worksheet is itself an aggregate data element, and so has an accessor, WSHT. All external accessors must pass through WSHT that acts as a gateway that defers their change events until the aggregate has stabilized into a consistent state.

The hook for external accessors is nearly identical to the hook for internal accessors, reflecting the fact that to application developers the two mechanisms appear the same. The key difference is in the requirement that external accessors have different change notification behavior. Matching the requirement of the application to the service provided by the hook is crucial to the successful use of frameworks. In this case, using an internal accessor where an external one is required will cause incorrect behavior in the application. There is an additional a hook, not detailed here, for creating new data aggregates which is similar to the NEW DATA ELEMENT hook but also requires the gateway accessor policy for propagating change events outside of the aggregate to be defined.

```
Name: Access Data Aggregate
Requirement:
     Some class needs access to a data element within an aggregation of
elements and needs to be notified of a change to that element after all
internal changes to the aggregate have been made.
Type: Enabling, Collaboration Pattern
Area: Data Management
Participants:
     DataAggregate    // The container data element
     DataElement,     // The data within the aggregate to be accessed.
     GatewayAccessor, // An accessor to DataAggregage
     Accessor,    // The mediator between the DataElement and the caller.
     Caller       // The class that needs notification of changes.
Changes:
     Caller -> GatewayAccessor.CreateComponentAccessor(DataElement)
     new operation Caller.Update  // Do actions upon notification here.
     Accessor.OnChange -> Caller.Update <provided>
     Caller -> read and write Accessor.Property
Constraints:
     Accessor and DataElement are of compatible types.
     Caller.Update called only when all data elements within the aggregate
have changed.
```

The ACCESS DATA AGGREGATE hook provides the same mechanisms as the ACCESS DATA ELEMENT for notifying the CALLER of changes and for allowing the CALLER to get and set values within the DATAELEMENT. However, the GATEWAYACCESSOR is called to create the external accessor. The constraint that the ACCESSOR and DATAELEMENT are of compatible types is still present, but one additional constraint is added stating that the UPDATE method of the CALLER is only invoked when the aggregate as a whole has completed changing.

X.4.2 Experience

We have found that applying the hooks model to a framework has several benefits. The precise and structured description provided by hooks can help clarify understanding and prevent incorrect use of the framework by application developers. For example, in an earlier version of the framework, a narrative description of how to use internal accessors was provided and used in developing the current SizeMaster application. However, since the description did not clearly state the requirement that the accessors to aggregates fulfilled, and the developers did not clearly state the application requirements, the accessors were used incorrectly and caused a flaw in the application. Discovery of the error also caused external accessors to be added to the framework.

Describing the hooks for the engineering framework before developing the application would have also caught the need for external as well as internal accessors. Writing the hooks forces the framework builder to state precisely how some part of the framework should be used. By forcing a walkthough of the intended use of the framework, defining hooks can help to expose deficiencies within the framework. When the hook for internal accessors was written, it became clearer that external accessors were needed.

Finally, writing hooks also exposes unnecessarily complex structure within the framework and forces the framework builders to either justify the complexity or to rethink it. While the proper hooks for using the framework may exist, those hooks may be complex or difficult to use. Often a complex hook or set of hooks is a reflection of needlessly complex structure within the framework. In an earlier version of the framework, the state (the properties) of a data element was part of a different class. Deriving a new type of data element then involved inheriting from three different classes (state, data element, and accessor) and creating a complex pattern of interactions between the three. After attempting to describe the hooks for it, the structure was streamlined to the current and easier to use version. Ease of use is one of the desirable features of frameworks and describing hooks

can help expose where the use of the framework can be simplified.

X.5 Conclusions

We have discussed the hooks model as a means of capturing knowledge about how to reuse the framework. The hooks model presents a structured approach to modeling the knowledge about the reuse properties of a framework. Each hook describes one aspect of the intended use of the framework and identifies the places in a framework that can or should be changed or augmented in the development of an application. By describing the hooks, implicit knowledge about how to use the framework is made explicit and open to study, refinement and reuse. Our experience with SEAF demonstrates that having knowledge about how a framework is intended to be reused is vital to the successful (cost-effective) use of a framework. Without it, application developers must synthesize the reuse knowledge implied in a framework design by examining architectural descriptions, source code and examples, before they can use the framework. Similarly, without a structured means of describing the properties of reuse of a framework, framework builders can only rely on informal narratives. These descriptions may be imprecise or incomplete, and inadequate for passing on to application developers the knowledge of 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 use the framework effectively.

We have chosen the SEAF frameworks to gain some initial experience in applying the idea of hooks developed in (Froehlich et al. 1997). This work has helped to validate our views that structured and precise descriptions are needed to communicate the intended use of the framework to application developers. We have also found that simply specifying the hooks helps to expose deficiencies or over-complexity in the design of the framework. The idea of hooks will continue to be refined based on our continued experience.

There are a number of areas of future work related to research involving hooks. First, tool support involves both finding the appropriate hook for a given application requirement, and then applying the framework extensions within the hook. Hooks at the option and pattern level are well-defined and can be partially automated by asking the application developer for the option desired or the parameters to fill in and then performing the appropriate actions automatically to incorporate those options into the framework. Second, hooks are currently text based, but a more graphical definition can be developed for incorporation into graphical object-oriented diagrams as well as the proposed tools. Third, operators on hooks can be defined. Structured means of adding new hooks by application developers, modifying existing hooks or even applying hooks will aid in the use and management of collections of hooks. Finally, we are interested in the area of framework evolution and how hooks affect evolution. The hooks themselves may indicate when a framework should evolve. If the hooks defined are not being used, then the framework may need to be rethought. Additionally, as more information and experience is gained about a framework and what it can be used for, the framework might evolve to change open hooks into pattern hooks or even option hooks.

References

Beck, K. and Johnson, R. 1994. "Patterns Generate Architectures." In *Proceedings of ECOOP'94* (Bologna, Italy): 139-149.

Booch, G., Jacobson, I. and Rumbaugh, J. 1996. *The Unified Modeling Language for Object-Oriented Development.* Rational Software Corporation http://www.rational.com/uml).

Campbell, R.H. and Islam, N. 1992. "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).

Froehlich, G., Hoover, H.J., Liu, L. and Sorenson, P. 1997. "Hooking into Object-Oriented Application Frameworks." In *Proceedings of the 1997 International Conference on Software Engineering* (Boston, Mass): 141-151.

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

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

Johnson, R. 1992. "Documenting Frameworks Using Patterns." In *Proceedings of OOPSLA'92* (Vancouver, Canada): 63-76.

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

Pree, W. 1995. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley Publishing Company, Reading, MA.

SEAF Project, unpublished (http://www.cs.ualberta.ca/softeng/SEAF/project.html).

VisualWorks Cookbook. 1995. Release 2.5, ParcPlace-Digitalk Inc., Sunnyvale, CA.

Appendix A: Grammar for Hook Descriptions

```
<hook> ::= <name>
           <requirement>
           <type>
           <area>
           [<uses>]
           <participants>
           <changes>
           [<constraints>]
           [<comments>]

<name> ::= Name: <string>
<requirement> ::= Requirement: <string>

<type> ::= Type: <method>, <level>
<method> ::= enabling | adding | replacing | augmenting | disabling
<level> ::= <option> | <pattern> | open
```

```
<option> ::= single option | multi-option
<pattern> ::= parameter pattern | collaboration pattern | pattern

<area> ::= Area: <string> [, .., <string>]

<participants> ::= Particpants: <identifier> [<type>] [<style>] [, ..,
     <identifier> [<type>] [<style>] ]
<type> ::= set of <identifier> [, .., <identifier>] |
     sequence of <identifier> [, .., <identifier>]
<style> ::= new | exists

<uses> ::= Uses: <hook name> [, .., <hook name>]

<changes> ::= <statement> [, .., <statement>]
<statement> ::= <loop statement> |
     <hook statement> |
     <new element statement> |
     <method statement> |
     <modify statement> |
     <participant statement> |
     <option statement> |
     <behavior statement>

<loop statement> ::= <loop id> <statement> [.. <statement>]
<loop id> ::= repeat [as necessary] | forall <var> in <set>

<hook statement> ::= <identifier> = <hook name> "[" <identifier> "]" |
     <hook name> "[" <identifier> = <rhs> [, .., <identifier> = <rhs>] "]"

<new element statement> ::= [new] subclass <identifier> of <identifier> |
     [new] property <qualified identifier> <whereclause> |
     [new] operation <qualified identifier>
<whereclause> ::=
     read of <identifier> maps from [set of] <qualified identifier> |
     write of <identifier> maps into [set of] <qualified identifier>

<method statement> ::= <qualified identifier> <method operation>
          <qualified identifier> [<return expression>] |
     <qualified identifier> <return expression>
<method operation> ::= copies | specializes | overrides | extends
<return expression> ::= returns <string> |
     returns [set of | sequence of] <rhs>

<modify statement> ::= remove code '<string>' [, .., '<string>'] |
     replace '<string>' with '<string>'

<participant statement> ::= fill in <identifier> [, .., <identifier>] |
     <identifier> add <set>

<option statement> ::= choose <identifier> from <set>

<behavior statement> ::=
     synchronization ( <qualified identifier>, <qualified identifier> [,
          .., <qualified identifier> ) [in <qualified identifier>]
          [provided]|
     [control flow] <qualified expression> -> [<read/write>]
          <qualified expression> [provided]
<read/write> ::= read | write | read and write
```

```
<rhs> ::= <identifier> |
      <set> |
      instance of <var> [[of | with] <attribute> <var>] |
      <loop id> <rhs>

<set> ::= <var> | ( <identifier>, <identifier> [, .., <identifier>] )
<var> ::= <identifier> | <qualified identifier>
<attribute> ::= <identifier>
<qualified identifier> ::= <identifier>.<identifier>

<constraints> ::= Constraints: <string> [, .., <string>]
<comments> ::= Comments: <string>
```