Requirements for a Hooks Tool

Garry Froehlich, H. James Hoover, Luyuan Liu, Paul Sorenson

Department of Computing Science

University of Alberta

Edmonton, AB. T6G 2H1

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

November 16, 1998

Abstract

Object-oriented frameworks can be complex and difficult to use. Tool support can greatly aid developers in using these frameworks. However, existing tools, such as graphical user interface builders, only support one framework and are not easily customized. Hooks, which describe the intended use of a framework, can form the basis of a general, flexible tool that supports many different frameworks. This paper describes the requirements for such a tool.

1 Introduction

Graphical user interface builders represent one of the most successful applications of framework technology. Object-oriented frameworks consist of a generic design for a class of problems, such as user interfaces, and an object-oriented implementation of that design. GUI builders allow developers to quickly piece together a window-based user interface from predefined components such as buttons, windows and menus. The interactions of the components

are well-defined within the framework but the components are given user-defined functionality. However, frameworks exist in many different areas beyond user interfaces, such as manufacturing [14], communications [7] , operating systems [3] and engineering [13]. Tools for these kinds of frameworks can help, and in some cases are critical in, making their use more successful as well.

A tool for developing applications from frameworks needs to be both an aid to the user and flexible enough to be used in different ways. There are two primary ways in which we envision such a tool being used. First, application developers uses the tool to quickly develop applications from the framework without changing the framework. Second, framework maintainers will use the tool to evolve or modify the framework itself.

In order to aid application developers, support for exploring the documentation of the design and use of the framework should be provided, along with support for actually adding application specific classes. *Hooks* provide the basis for documenting the changes the framework builder intended to be made to the framework. The structured description of those changes within hooks can be enacted interactively with the user of the tool. Aid to framework maintainers is provided by allowing extensions to the framework and the hooks themselves to be made and incorporated back into the framework.

Existing tools, such as GUI builders, focus exclusively on a single framework, and do not support the integration of multiple frameworks. Users must learn multiple tools and do integration between frameworks by hand, but a tool based on hooks can be flexible enough to support many different frameworks. A general tool will also enable framework builders to adapt the tool to their framework instead of going through the expense of developing a custom tool for each framework, or not providing tool support at all. The flexibility to add new components, add new hooks, modify hooks or modify parts of the framework is required by the framework maintainer.

In this paper, we outline the key requirements for a tool we are building to aid in the use of frameworks to develop applications. Section 2 describes the concept of hooks which are at the heart of the tool. Section 3 describes in greater detail the requirements of the

tool for application developers and specifies how the tool can be used to enact the changes described in a hook. In Section 4, the focus is on requirements for the tool from a framework maintainer's perspective. Finally, a summary and future directions are given.

2 Hooks

A hook [4] 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, each focusing upon a smaller problem within the larger, more complex problem.

Each hook details the design/implementation actions that are required, the constraints that must be adhered to, and any effects on the framework that will result. The hook exposes only the framework details needed to solve the problem so that it can be quickly understood and used. Once the correct hook has been found, an application developer uses the hook simply by performing all of the activities within the changes section of the hook in the order given.

A 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 format in which hooks are described helps to organize the information and make the description precise and amenable to enaction within a tool. 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.
- Preconditions: constraints that must be satisfied before the hook can be applied.
- 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.
- Postconditions: constraints that must be satisfied after the hook has been applied.
- 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.

An 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.

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. 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. Since they are open-ended, the developer has to be more careful about the effects changes will have on the framework.

3 Requirements for Application Developers

When application developers start development using frameworks, our tool first must create a copy of the framework or multiple frameworks that they wish to include. The tool incorporates the assumption that the original design or implementation of the framework should not be modified. This assumption preserves the benefits of maintaining a common code base between a family of applications; otherwise, the application is no longer an extension of the framework, but an evolution of its code base.

The tool distinguishes between two types of classes:

- Framework classes are classes provided with the framework and should not be modified.
- Application classes are added to the framework classes by application developers in order to implement specific functionality and can be modified freely.

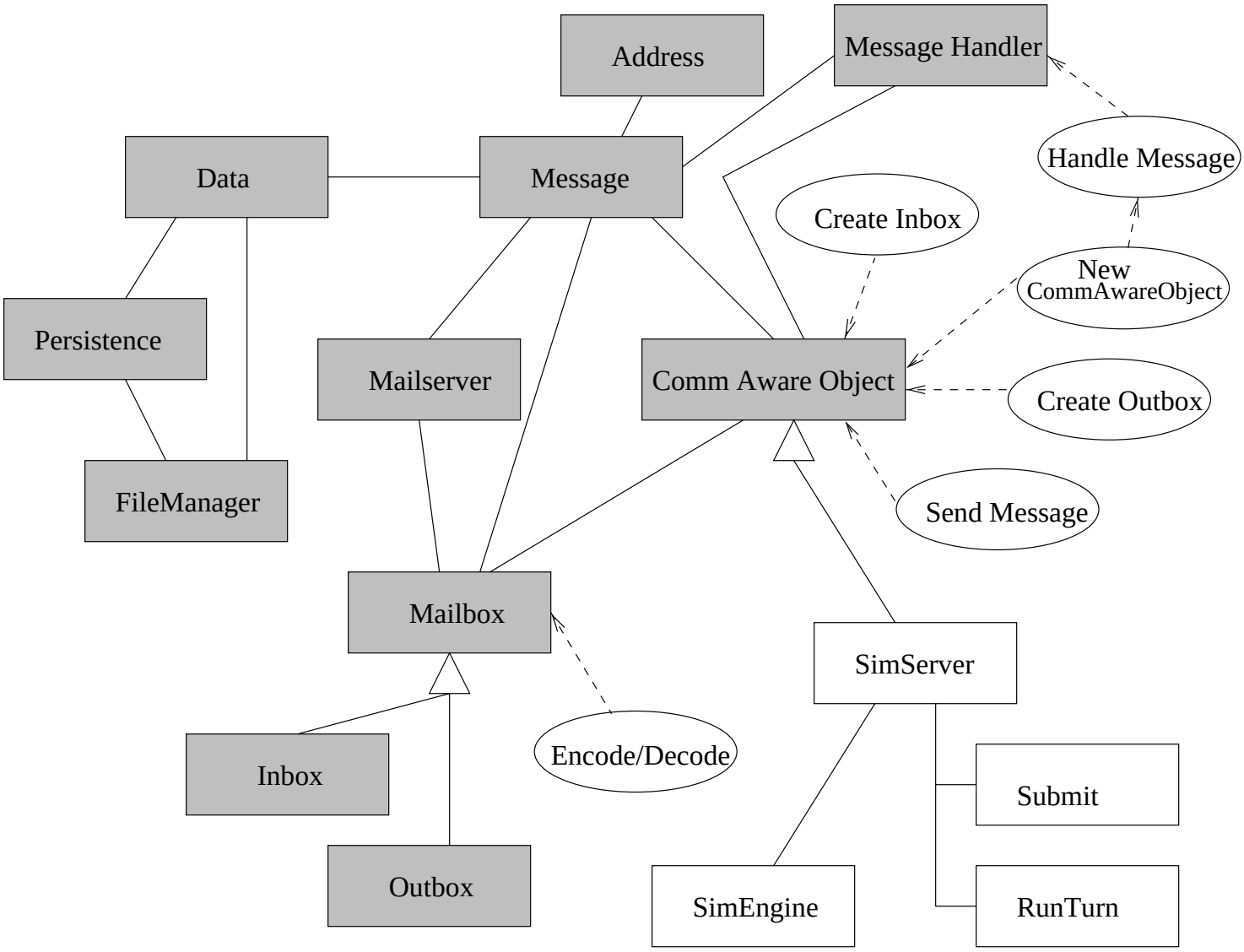

Figure 1: CSF: Framework View.

To protect the design of the framework it can be distinguished from application classes through color or shading. Figure 1 shows a simplified view of a client-server framework (CSF) using shading in UML. Being able to see the overall design of the framework is one of the main requirements for the frameworks tool. UML [1] provides a standard diagramming notation for both the structure and behavior of the framework through class and interaction diagrams. We are adding explicit support of the notion of hooks to UML for the tool.

The overall view, different than UML views, shown in Figure 1 is the main view of the framework, or the framework view. It contains all of the classes of the framework, along with any additional classes that have been added by application developers. For example, Data is a framework class while SimServer is an application class. Hooks such as Handle Message are shown as ovals which connect to the primary participant within the hook.

Whenever a user tries to modify or delete one of the framework classes such as CommAwareObject, a warning is given with an explanation of the potential danger of changing the framework. There will be cases in which the framework has to be modified to complete the development of an application because of some limitation of the framework or because the application developers are scavenging parts of the framework, so we provide the flexibility to override the warning. Modifying application classes such as SimServer do not cause any warnings.

Option hooks are a special case, since they include optional components to the application, but these components are provided as part of the framework library. Attempting to modify the components themselves produces a warning, but modifying the choice of components does not.

Users of the tool can develop applications in two main ways. In the first, application developers use the editing tool to add or modify application classes in any means desired. The second, preferred, method is choosing and enacting hooks within hook views.

3.1 Hook Views

When a user clicks on one of the hooks within the main view, or another hook view, a hook view opens. A hook view contains a subset of the overall view of the framework. Both class and interaction diagrams are contained within the view. The view forms the context within which the hook is used, containing the participants of the hook and related classes. Since determining the complete context of a hook is difficult, hook views are defined by the framework builders rather than generated dynamically. Hooks may share the same context and thus the same view. The hook view shown in Figure 2 shows the context of the Handle

Handle Message Hook

Preconditions

- ☐ NewCOA subclass of Comm_Aware_Object

Changes

- ☐ new subclass NewMH of MessageHandler
- ☐ NewMH.handleMessage(Message m, CommAwareObject coa) extends MessageHandler.handleMessage(Message m, CommAwareObject coa)
- ☐ new operation NewCOA.register
- ☐ NewCOA.register -> NewCOA.registerHandler(message.type, NewMH)

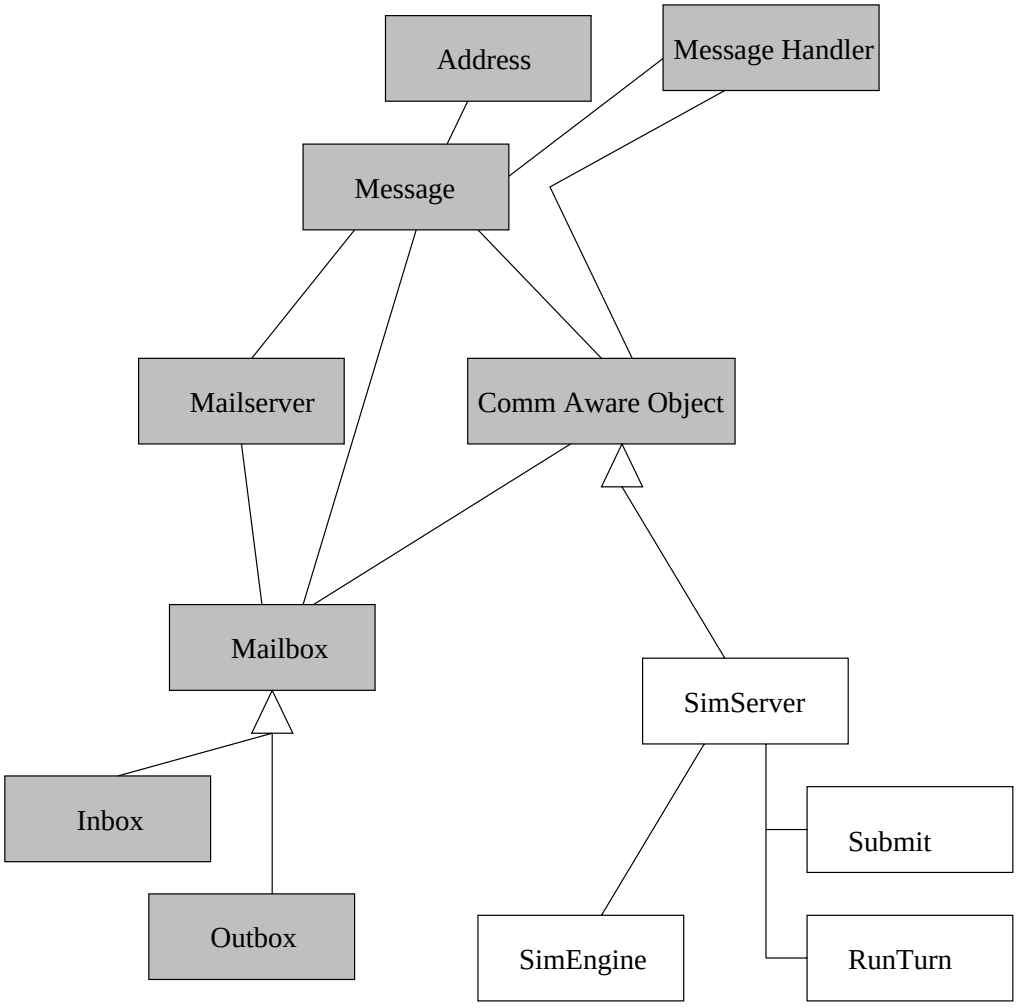

Figure 2: CSF: Handle Message Hook View.

Message Hook. It contains not only the diagrams but also an additional window which lists the pre and post conditions of the hook that was requested by the user along with the changes that can be made using the hook.

Within a hook view, application developers can enact the changes within the hook, semi-automatically with the tool's help. The allowable conditions and change statements are defined by a grammar and can be interpreted by the tool. That is, the tool interactively goes through the activities contained in the hook with the user, performing as much as it can automatically and requesting any required information from the user. Section 3.2 provides a more detailed example of how this happens. As the first stage in the process of enacting the hook, any preconditions of the hook must be satisfied. Next each of the changes within the hook can be made, or ignored if not needed. Finally, all the postconditions of the hooks should be satisfied. Checking a single precondition or postcondition, or enacting a single change statement is called a step.

The application developer has several actions that can be performed within a hook view:

- *Start enactment.*
- *Do a step.* Check one precondition, one postcondition or perform one of the changes within the hook.
- *Undo a step.*
- *Undo all steps and start over.*
- *Suspend enactment.* This action is required in order to perform some other tasks or leave the work for a later time.
- *Invoke another hook.* Hooks which use other hooks will place the current hook view in a suspended state and open up a new hook view.
- *Resume with playback.* If modifications to the application have been made within the context of the hook view, then preconditions must be checked again to ensure that none

of the modifications have violated them. After the preconditions have been checked, changes that were previously made by the developer can be 'played back' meaning they are automatically invoked by the tool.

- *Resume without playback.* If no modifications have been made, then the application developer can resume stepping through the hook immediately.
- *Commit with all conditions met.* Changes made within the hook view are propagated to the main framework view.
- *Commit without all conditions met.* Changes should only be propagated when all the postconditions of the hook are satisfied, but the application developer has the option of overriding any conditions.
- *Throw away changes.* Any changes within the hook view are simply discarded and the view closed.

3.2 Example Hook Enactment

As an example, when a user selects the New CommAwareObject hook show below, the communication hook view is displayed (see Figure 3). The changes window has been removed for simplicity. In this example, we are using the client server framework to create a simple internet-based simulation, such as a turn-based farming simulator. Users of the simulator enter a turn's worth of information using a client program or web-browser and submit it to the server, such as decisions about which types of crops to grow for a given year and how much fertilizer to use on the fields. The server then invokes the simulation engine and returns the results of the simulation to the user. Here we focus on the creating part of the server side. In order to create a server class which can communicate with clients using CSF, the New CommAwareObject hook is used.

```
Name: New CommAwareObject
Requrement: An object needs to communicate across the network.
```

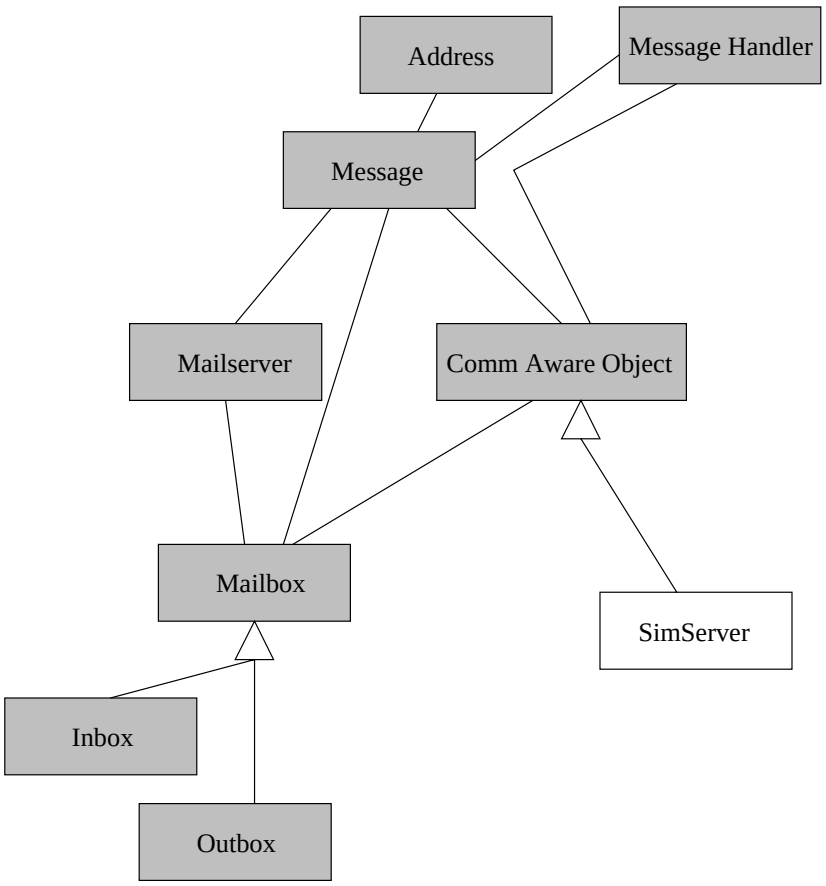

Figure 3: CSF: New CommAwareObject Hook View.

```
Type: Enabling Pattern
Area: Communication
Participants: NewCOA, Comm_Aware_Object (provided), Message
Uses: Handle Message
Preconditions: none
Changes:

// First create a new subclass of Comm_Aware_Object
new sublcass NewCOA of Comm_Aware_Object
repeat as necessary
        fill in Message.type  // a string name of the message (corresponding
                // to the type) that the COA should respond to.
        // invoke the Handle Message hook
        Handle Message[NewCOA = NewCOA, message = Message]
```

Along with the view, a changes window displaying the preconditions and changes sections of the hook is shown. When the user is ready to start enacting the hook, they select the first step within the changes window. As this hook is a creation hook, it has no specific preconditions, so the first step becomes the first change, the creation of a new subclass of the framework class CommAwareObject. NewCOA is a variable representing an application class, which is called SimServer in Figure 3. The next step is a repeat loop which simply repeats all of the steps inside of it until the user declares that they are finished. The 'fill in' step brings up a dialog with the user that requests a string representing the message type. We want to create a message for submitting a turns worth of information to the servers, so the message type is SubmitTurn (not shown). CommAwareObjects communicate through messages represented by the Message class, each of which has a unique type. The second step within the loop then invokes the Handle Message hook. To do this, the current hook view is suspended and a view for the Handle Message hook is opened.

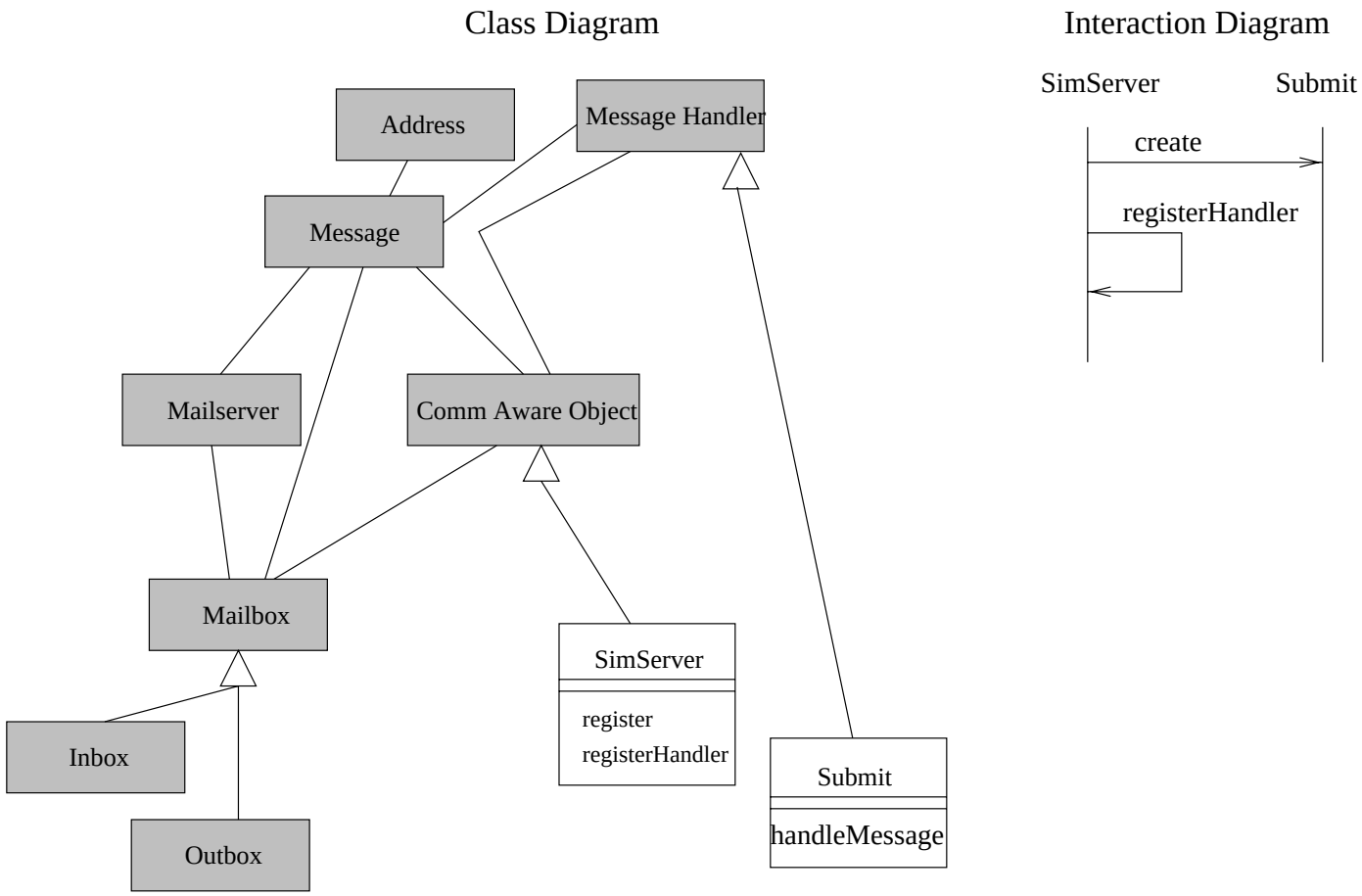

Figure 4: CSF: Handle Message Hook View.

```
Name: Handle Message
Requrement: When an object receives a message, it needs to respond to
it in some way.
Type: Enabling Pattern
Area: Communication
Participants: message, NewCOA, MessageHandler (provided), NewMH
Uses: none
Preconditions:
NewCOA subclass of Comm_Aware_Object
Changes:

// First, create a new MessageHandler subclass
new subclass NewMH of MessageHandler
// Specialize the handleMessage method and fill in the appropriate code.
// 'extends' infers that handleMessage must call it's superclass
// ie. code 'super.handleMessage(message,coa);'
NewMH.handleMessage(Message m, CommAwareObject coa) extends
    MessageHandler.handleMessage(Message m, CommAwareObject coa)
// The next line is for 'hooking' up the handler.
// Registering can occur within the initialization method of the object
// (the name 'register' is just a placeholder).
new operation NewCOA.register
NewCOA.register -> NewCOA.registerHandler(message.type, NewMH)
```

The Handle Message hook has the same context as the previous hook, although a new window is opened on the screen. Before the hook can be enacted, the participants are mapped to the parameters given with the hook invocation from the New CommAwareObject hook.

As shown in the hook call, the NewCOA participant (SimServer) is mapped to the NewCOA participant in Handle Message. The Message participant is similarly mapped to message in Handle Message. The first step in the enactment of the hook is to check the only precondition which checks automatically to see if NewCOA is a subclass of CommAwareObject. In this case it is, so enaction can proceed to the first change step. The first step creates a new subclass NewMH of MessageHandler. NewMH is another application participant. In this case we are creating a handler for the SubmitTurn message, so NewMH corresponds to Submit in Figure 4. Next the handlemessage method on NewMH is filled in by the user. A dialog then asks the user for a new operation, or a link to an existing method, within NewCOA (SimServer) from which registration of the handler can occur. The last step in the changes section then invokes the callback method regiseterHandler within NewCOA.

The changes are then committed to the parent view, the New CommAwareObject hook view. Once the changes are committed, the New CommAwareObject hook view resumes and the repeat loop is invoked again, unless the user is finished. Here, we have no more messages to add, so the loop is stopped and the changes are committed to the parent view again, that is, the main framework view.

3.3 View Consistency

When changes are committed, consistency must be maintained between the hook views and the main framework view. Consistency is maintained by propagating the changes from the hook view to the parent view. All of the changes are logged by the tool and essentially they can be played back in the parent view. The framework classes that participate within the hook serve as the anchors that exist in both the framework and the hook view and on which the changes can be played back.

In general, propagating an arbitrary set of changes from one view to another is a very difficult problem. Due to the nature of the changes that can be made within a hook, the problem becomes much simpler. Only application classes are modified, so the two views will always have a set of framework classes in common. Further, the changes involve some

modification, but mostly additions of classes, methods or properties to a class which can be easily added to the main view. However, in some cases, there will be interference between hooks that must be caught.

3.4 Hook Interference

Interference occurs when two hooks enact changes which conflict with each other, or when general changes are made to an application that may conflict with an active or suspended hook view. Interference can occur when:

- A participant of a hook is deleted.
- A participant of a hook is modified.
- Namespace conflicts, such as two classes being given the same name.

In the general case, whenever arbitrary changes are made to a class in any view, all hook views currently open with that participant must have their preconditions checked and potentially undergo a resume with playback operation.

If the changes only involve the enactment of hooks, then the tool can help to prevent interference by defining sets of mutually exclusive hooks. Hooks with the same participants, particularly those which are application classes have the potential to interfere with one another. Two hooks within the same set cannot be enacted at the same time. As shown in Figure 5, hooks A and B cannot be enacted at the same time, but hooks A and C can. Since hooks do not allow the deletion of participants, only hooks which modify the potential participants of hook A need to be considered for belonging to the set of mutual exclusion for hook A. Currently, the framework builder defines the sets of mutual exclusion. By default, all hooks are considered to be in the same set. Namespace conflicts can occur for example when two new subclasses of a single class have the same name and can be checked prior to committing the changes to the hook view.

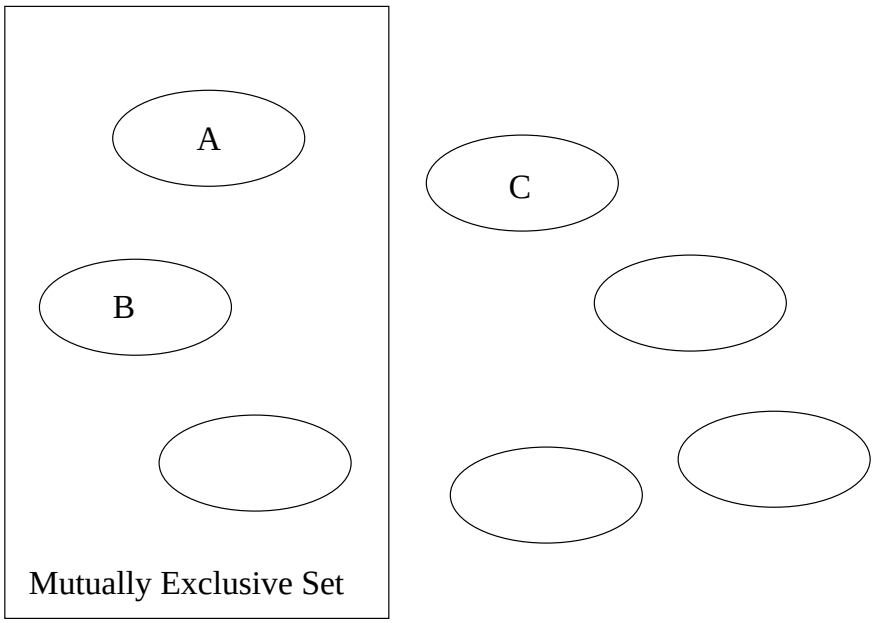

Figure 5: Mutually Exclusive Set of Hooks.

Hooks are automatically considered to interfere with themselves, but this is not always the case. A hook may not conflict with itself if it only adds things, unless the hook is meant to be used only once. Two separate views (two different windows) of the same hook might not have the same application classes as participants. The application participants, as opposed to the framework participants, are the important ones, since framework participants shouldn't be modified by the hooks. So two views of the same hook may not have the same application participants and therefore they shouldn't conflict.

Two hooks might also invoke incompatible options within option hooks. The options should be specified in pre or post conditions to ensure that doesn't happen.

The tool does not consider implicit conflicts such as those created by a method or class which calls a participant during normal framework operations. A distinction can be made between interference during the enactment of hooks and logical errors within the application. Changes might cause some subtle conflict within the application, but this is a logical error in relation to the framework or even the application extensions and not a direct result of the hooks. A logical error invalidates the framework, but does not invalidate the hook. Detection

of logical errors is a much larger and more complex problem outside the scope of hooks.

3.5 Hook Books, Examples and Use Cases

Design diagrams and hook views should not be the only means of presenting the framework. Other information such as use cases, examples and class descriptions give additional information to the user to aid in application development. Use cases provide a means of describing typical scenarios, sets of actions involving framework classes, in which the framework is used. They are valuable to people first learning to use the framework, or learning to use it in a different way.

Examples are equally valuable to show how the framework can be used. Users typically grasp concrete examples more quickly than abstract descriptions. The examples should be both of the framework in general (sample applications) and of individual hooks and use cases.

Descriptions of methods and classes are also necessary. These help application developers to understand the purpose of a class or method as these descriptions are not contained within the hook descriptions.

The *hook book* is a listing of all hooks within the framework. The book lists the name and the requirement sections of the hook. The hook book can be used to browse the list of hooks, or it can be searched for particular keywords or matching requirements. Additionally, the hook book can be used to monitor which hooks have been used, how many times they have been used, and which hooks views are currently open among all of the multiple users of the tool.

All of these things, the use cases, the examples, the class and method descriptions, and the hook book, are linked together to form a web of information about the framework which can be easily browsed. Use cases link to a series of hooks that are used within the use case. They and hooks also point to examples of the use of the framework, and conversely, examples point to the hooks that have been used within them. Hooks also point to the descriptions of the methods and classes that participate in them.

Finally, a log of all of the changes made through the hooks is kept which can then be reviewed when errors are detected. The log is also an invaluable tool when changes to the framework itself are made (ie. a new version is released). If hooks have changed in the framework, the log will show which parts of the application have to be modified to work with the new version of the framework.

4 Maintainers and Developers

One of the key advantages of the tool is its ability to be used for many different frameworks rather than focusing exclusively on one. Once the initial development of a framework has been completed, the framework builders only have to import their design into the tool and then define the hooks for it. However, the tool is not appropriate for the initial development phases of a framework. Development typically happens in a tight spiral and constantly updating the framework model for the tool would require too much unnecessary overhead. Once the code base has become stable, it can be imported into the tool to aid in the use and future maintenance of the framework.

During maintenance, changes are typically made to the main view and then the hook views are updated accordingly. Maintaining consistency is not as much a concern, since modifications should be made by few or one maintainer. Maintenance of the framework involves two main areas: modifications to the hooks and modifications to the design of the framework itself.

To allow for flexibility within the tool, new hooks can be added to the framework. Adding hooks involves the least amount of difficulty when refitting existing applications, since no change to the application is required. A hook design screen is used which presents the hook template. The framework maintainer fills in the appropriate fields which then can be checked for internal consistency by the tool. Then the hook can be placed in the framework view and a new hook view defined for it, or an existing hook view assigned to it.

Hooks may also be modified by the maintainer, either to correct errors, to improve the

hook or because the underlying framework has evolved in some way. The main difference between modifying and adding a hook is that the hook description and view already exists, otherwise the same process is followed. However, existing applications may themselves have to be modified when the hooks are changed.

The framework itself will evolve over time to add new functionality or to recast existing functionality in a new way. Changes to the existing framework can be accomplished within the tool (after turning the warnings off). Any changes to the participants of a hook may require changes to the hook and will likely require changes to existing applications as well.

5 Related Work

Other means of describing the intended use of a framework include cook books and framework patterns. Cook books can be presented as tutorials that describe the basis of the framework and examples of its use [9], or can be a listing of individual problems and their step by step solutions within the framework [12]. Similarly, framework patterns [8] document common problems and solutions with examples through a general narrative. These are called motifs in [10]. The information contained in cook books and framework patterns is valuable but cannot be easily interpreted by an automated tool, unlike hooks.

A tool for the exploration and use of frameworks through exemplars is described in [6]. A concrete instance is provided for all abstract classes in the framework. These can then be executed through the tool to learn the behavior of the framework. A tool also exists for examining or discovering the design patterns [5] used in a framework [11]. These tools allow exploration of the design of the framework, but do not explicitly describe how the framework can be used.

Parallels also exist between the hook tool and aids used in commercial tools such as the wizards or wizard-like dialogs in Borland Delphi [2] or Microsoft programming tools. These are dialogs that take a user step-by-step through some process and request information from the user much in the same way that the hook tool does. However, wizards are generally

coded for a particular tool and cannot be dynamically generated. A tool for enacting hooks is much more flexible in that the hook descriptions are interpreted and hook descriptions can be easily modified.

Graphical user interface (GUI) builders that come with many tools, including VisualWorks for Smalltalk [12], have already been mentioned. These tools allow users to visually position components on a screen and to adjust a list of parameters provided with the component. Typically, the user can also define or fill in methods that can respond to events within the system. Similarly, application developers can select components to include through option hooks and adjust parameters or fill in methods through pattern or open-ended hooks. However, the hook tool applies to many different frameworks whereas GUI builders focus on a single visual framework.

6 Conclusions and Future Work

Using the same basic ideas that exist in graphical user interface builders, tools can be constructed to aid in the use and evolution of object-oriented frameworks. The notion of hooks helps to form the basis of the tool by describing how the framework is intended to be used and showing where changes can be made. The hook tool aids users by extending the UML language to include hooks and by semi-automatically enacting the changes within hooks. The tool handles propagation of changes between views and helps to prevent inconsistencies. Additional support comes from extensive use of use cases, examples, class descriptions and hook books. To support evolution, the tool is flexible enough to allow hooks to be added or modified along with the framework itself. Finally, the tool is flexible enough to provide support for many different frameworks, or more than one framework at a time, which is not currently done in existing tools.

In the future we would like to increase support for application developers and framework maintainers. Incorporating new versions of a framework into an existing application can potentially be aided by the tool since all application extensions to the framework have been

logged and potential areas of conflict can be flagged. We would also like to provide additional support for framework developers who define the intended use of the framework through use cases by identifying hooks within the use cases. Finally, an interesting extension of the work with use cases would be to allow users to define use cases and then identify the hooks within the framework which will fulfill those use cases.

References

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

[2] *Borland Delphi for Windows.* Borland International, Inc., Scotts Valley, CA, 1995.

[3] 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.

[4] G. Froehlich, H.J. Hoover, L. Liu and P. Sorenson. Hooking into Object-Oriented Application Frameworks. In *Proceedings of the 1997 International Conference on Software Engineering* (Boston, Mass, 1997), pp. 491-501.

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

[6] 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), pp. 90-99.

[7] H. Hueni, R. Johnson and R. Engel. *A Framework for Network Protocol Software.*, In Proceedings of OOPSLA'95, Austin, TX, 1995.

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

[9] 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.

[10] 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.

[11] 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.

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

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

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