REALIZING REQUIREMENTS IN PRODUCT-LINE DEVELOPMENT USING O-O FRAMEWORKS

Garry Froehlich, H. James Hoover, Paul Sorenson

Software Engineering Research Laboratory
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
{garry, hoover, sorenson}@cs.ualberta.ca

ABSTRACT

An existing, rapidly evolving strategy for realizing requirements in product line development is described. The position taken and the proposed new directions are based on our substantial experience in building and using O-O frameworks. Important new directions include the use of hooks to encapsulate product-line requirements and the management of requirements using a hook evolution model and experience base construction. It is also proposed that requirements traceability can be supported in an obvious manner by following prescribed processes for product development that are based on enacting hooks.

INTRODUCTION

The paper begins with an overview of product-line development, O-O frameworks and the relationship between the two. The importance of requirement capture and management in software product-lines is identified and strategies for determining product-line requirements are reviewed. A form of prescriptive documentation, embodied in the hooks model [Froehlich97), is introduced as a mechanism for encapsulating requirements. Hooks use script style descriptions to assist the developer in extending a framework into a product using a set of intentional requirements that are anticipated by the O-O framework developers. The issues of how hooks are changed to support evolving requirements is discussed. A proposal for how requirements traceability can be supported using hook enactment is presented. This approach to traceability can be supported through a tool like HookMaster [Liu99).

Although much of this work is still "work-in-progress," it is based on our substantial experience in building and using O-O frameworks in areas such as engineered products [Froehlich99a) and personal assistants [Liew99) for the web.

PRODUCT LINES AND O-O FRAMEWORKS

Product-line architectures, or product families [Parnas79), move beyond the traditional notion of building a single product in isolation, and instead focus on building whole suites of related products within a single domain. The concept takes the view that most development is actually redevelopment. Most products have been built before at some level. Industries such as telecommunications build the same or similar products over and over again. Although the core requirements evolve slowly, there may be significant feature changes from one product to another.

Domain analysis techniques, such as FAST [Weis99) or FODA [Kang90), capitalize on the similarities between programs within a domain. Using commonality analysis techniques, they examine products within and across a domain or rely on the knowledge of experts to discover the core requirements that apply to the entire domain. These are divided into two basic, but important, types. *Commonalties* are the same across the products, such as the event model in a window based user interface. *Variations* are the parts that differ between products, such as the actual menus or buttons within that user interface.

Similarly, analysis patterns [Roberson93, Fowler97) capture common requirements and their solutions within a particular domain such as accounting. They document problems and solutions, and capture the experience of developers within the domain. Analysis patterns are at an abstract enough level to capture both the commonalties and variations of the problem.

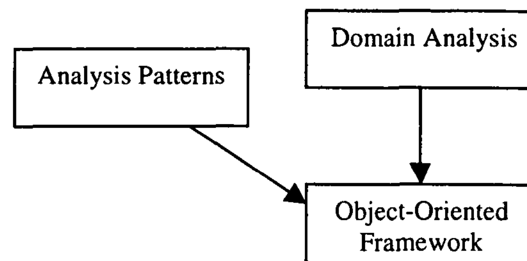

Figure 1: Relationship of Analysis Techniques to Frameworks

These domain analysis techniques and patterns are used to produce the domain specific representations of the product-line. These can take the form of tools that support domain specific languages such as JTS [Batory98), or domain 'jargons' as in FAST [Nakatani99). As shown in Figure 1, the products of domain analysis and analysis patterns can also be used as the basis for building object-oriented frameworks. O-O frameworks consist of an architecture that covers an entire domain, and an implementation of that architecture. The commonalties are built into the architecture and implementation so that they need not be developed over and over again. The variations are included as hooks into the framework. Product developers can easily attach their own components through the hooks. The framework forms a solid basis for quickly developing new products within the product line.

Strategies for Determining Product-line Requirements

Product-line requirements differ from the requirements of a single product in their breadth and their generality. In order to produce a successful product-line, the requirements must cover an entire domain and they must be general enough to capture both the commonalties and variations within that domain. New strategies are needed and these are explored in the next section.

One strategy is to examine or to build products within a domain in order to gain experience in that domain (Taligent 1995). Obviously several products must be built or analyzed to determine the commonalties and variations across them. Typically, at least three such products are needed.

Another strategy is to employ a domain expert. Key to our success in constructing a product-line framework for engineering sizing and selection tools (Froehlich 1999a) was the close collaboration with the product line champion: a domain expert in pressure safety relief valve sizing and selection His familiarity with not only the relief-valve domain but the broader domain of engineering design was indispensable in the development of our first product SizeMaster Mark IV (see www.avrasoft.com/sizemaster) and the current EAF (Engineering Application Framework). We were also influenced by the expert users, practicing chemical engineers, who raised requirements that neither the domain export nor we anticipated. Building products and using domain experts complement one another, as we have discovered in the development of our frameworks.

A third approach that supports both the determination and elaboration of requirements is the creation of an experience base. In this approach, product-line development is treated as a continuous case study in which results from using the framework in several product developments is captured. These results are later used to improve the framework and the product development process that is based on the framework. The requirements for our CSF (Client Server Framework - see http://www.cs.ualberta.ca/~garry) are undergoing this type of refinement as part of the ongoing FrameScan case study (Froehlich 1999b). The CSF supports the development of simple web-based client server applications in our fourth-year software engineering class CMPUT 401 - see http://www.cs.ualberta.ca/courses/Cmput401.html. The ultimate goal of this approach is to provide an experience base that complements and links to the requirements (encapsulated in hooks as described below), examples and other design documentation of a framework. Our work to date in this area is preliminary but promising.

Documenting Product-line Requirements

In this section, we describe how the requirements that are common across products are encapsulated, and how those that vary within the product-line architecture are documented. The variations, which are particularly important to product developers, are documented prescriptively using hooks. Finally, we examine how

product-line requirements evolve as the domain of the family of products evolve and why they must be managed carefully.

Hooks

One means of encapsulating these requirements is through the use of hooks (Froehlich 1997). Each hook documents a narrowly focused means of using the framework to fulfill a particular requirement. For example, a hook will indicate how to create a menu within a user interface framework, or which packages to choose in a command and control framework. The suggested hook documentation consists of a structured template that elicits all of the required information needed to select and use the hook, including a section on the requirement fulfilled, the parts of the framework it uses, and any conditions on the use of the hook.
Since the product-line architecture and implementation already exists, the requirements must be mapped onto that implementation. As shown in figure 2, each hook maps its requirement back to the place in the framework that it affects. In addition, the hook defines in a systematic manner, how the design and code base of the framework should be extended.

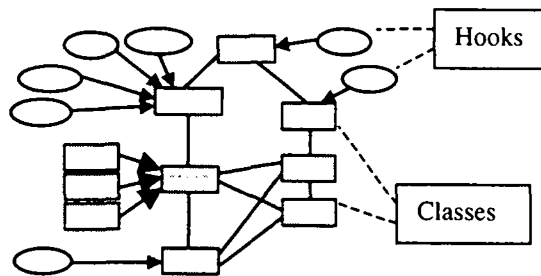

Figure 2: Hooks and Classes in CSF

Figure 3 shows the space of possible variations, that is, the changes, modifications or additions needed to produce a product within a particular domain based on an O-O framework. It is divided into four overlapping regions.

Option hooks are provided when the variations of a particular aspect of a domain can be explicitly catalogued. They are the narrowest in scope, but are also the easiest for the user. Option requirements are fulfilled by selecting one or more items from a pre-defined set of components and linking them into the object-oriented framework. They can be completely automated and tested to ensure conformance with the framework.

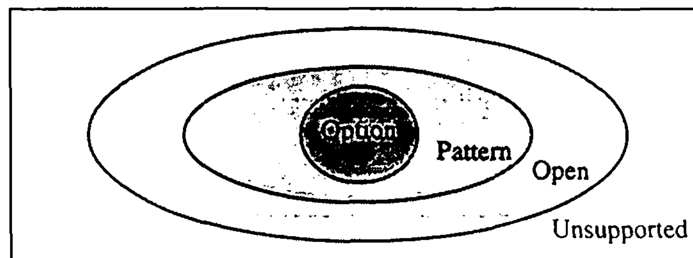

Figure 3: The Space of Possible Extensions to a Framework

Pattern *hooks capture variations that can be parameterized and/or restricted. They may include options, but they also present the user with a restricted set of operations that the product developer is allowed to perform. Patterns can be partially automated and tested.*

Open hooks use a much less restricted set of operations. Open hooks allow entirely new functionality to be added to the framework, and represent wide variations within a domain.

Unsupported changes go beyond the uses the framework builders envisioned for the framework. They may arise because the framework does not completely cover the domain or that the requirements of the product-line have changed. Typically, the discovery of unsupported changes is an indication that a framework needs to be updated or evolved.

Examples of all four kinds of hooks can be found in our Engineering Application Framework (EAF) which is used to build engineering worksheets. Each worksheet is a collection of data elements and calculations that represent the contents of an engineering code or standard. Each engineering application that we build has one or more worksheets that form the core of the application. The user interface of the application, provided by another framework, navigates through the worksheet but does not do any engineering computations. In this way we keep the robust core engineering part of an application relatively isolated from the more fragile user interface.

An engineering standard is embodied into a worksheet by following the EAF hooks. Option hooks are provided for adding data elements to the worksheet. Each item of data in the standard is identified, and then classified according to whether it is numeric or text, and whether it is an external input from the user or an internal computed value, and whether its range of values is fixed or user modifiable. For example, one combination of options can be used to add a text data element that has a predetermined set of user-selectable values. For numeric data, there is the additional option of what basis units it has, such as length or temperature, and what its normally displayed style should be (for example, English or metric).

Pattern hooks are provided for adding new data element types, basis units and calculations. The hooks for new elements and units involve constructing new classes directly by writing code. The pattern for adding new calculations is embodied in a custom calculation language that hides many of the details. Calculations are written in this higher-level custom language and then compiled into the class definitions that implement the calculation. This allows us to alter the implementation of the calculation hooks without having to rewrite the calculations. We just recompile them.

Open hooks are provided for calculations that cannot be described by our calculation language. Worksheets are by nature acyclic in order to enforce a consistent evaluation order. This means that iterative components of the calculations must be individually programmed, and then attached to the framework through open hooks.

Unsupported changes are more an issue of good framework architecture. If the architecture is well-factored, we find that adding hooks for unanticipated requirements is a natural, although non-trivial, activity.

EVOLVING REQUIREMENTS

When the requirements of the product-line evolve, it can have several different effects on the underlying framework:

Changes to the underlying framework. These occur when the common requirements change, new ones arise, or the framework design needs to be made more robust. Such changes typically include refactoring (Fowler 1999) the framework classes without affecting the hooks. The variable requirements themselves don't change so the hooks don't change either; the only changes are in the common parts of the framework and their underlying implementation.

Changing hooks to make them more flexible. Sometimes the variable requirements supported by the framework are not flexible enough to accommodate the requirements of the products within the domain. Generally, one or more hooks need to be evolved along with parts of the underlying framework in order to support the new flexibility. As shown in figure 4, to make a hook more flexible, it must be made more open as well.

Changing hooks to provide more support for the product developer. As a framework matures, the exact variations between products within a domain can be catalogued and support for the different variations built directly into the framework. Building in variations makes the framework easier to use as it contains more support for the product developer. As shown in figure 4, this is the opposite effect of increasing the flexibility of the framework.

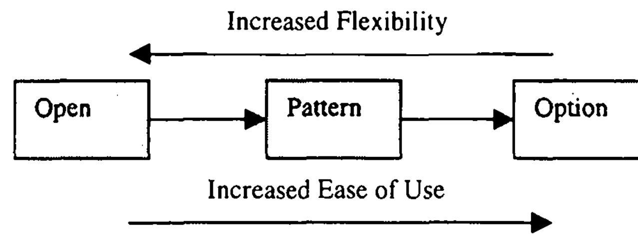

Figure 4: Evolving from One Type of Hook to Another

Adding new hooks. New hooks can be added in two ways:

- As users gain more experience with a framework they may discover ways to support a requirement that was not documented by the original framework developers. These experiences can be documented in a hook description, usually an open hook, and are supported in new releases of the framework. Existing products need not be affected.
- New functionality can be added to the framework itself, and this functionality accessed through hooks. This type of change can have pervasive effects on the framework and can affect existing products that rely on new releases of the framework.

REFINING REQUIREMENTS AND REQUIREMENTS TRACEABILITY

In our experience, we have found that one of the difficulties in using a product-line architecture is mapping the requirements of the product to the requirements supported by the architecture. When developers are familiar with the architecture and the framework, they can perform this mapping intuitively. But when developers are new to the framework, it can be difficult to see the relationships between the focused requirements represented by the hooks and the coarse requirements of the product.

We define a high level process consisting of a series of steps that guides the new product developer through the use of the framework. Figure 5 shows an example for the Client-Server Framework (CSF). Each of the steps is written in natural language and discusses a key concept of the framework, much like framework patterns (Johnson 1992). The steps link to the hooks themselves, which in turn link to the framework. These steps provide three benefits:

Mapping. The high level process guides the user in deciding how to map their requirements onto the requirements supported by the framework. For example, the first two steps, Discover Data and Identify Persistence, guide the framework user in deciding what they need to communicate across a network, and which information needs to be persistent (stored in files or in a database). They are then pointed to specific hooks in the framework, specifically New Data and Read/Write Data, which handle transmitting and storing information in the CSF.

Order. The steps of the process explain the order in which things should be done to successfully use the framework, which isn't clear from a simple list of supported requirements. Even in the case of the CSF, we found that when new framework users were presented with the unordered set of hooks and the dependencies between them (e.g. the Send Message hook depends on the New Data hook), they needed some coaching in where to get started and how to proceed further. The ordering shown in Figure 5 is somewhat simplistic, but appropriate for the CSF framework. However, in practice for larger and more complex frameworks, choices made later in the steps may affect earlier steps, so more complex orderings and mappings may be required.

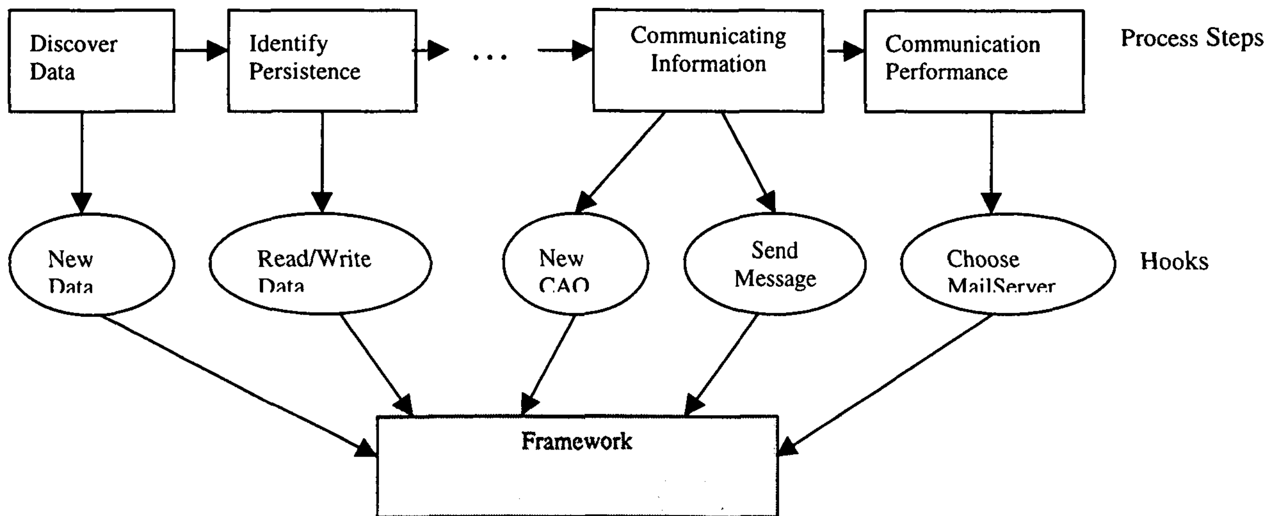

Figure 5: Process for Using CSF

Details. This approach is similar to work done by Lajoie and Keller (1994) in which high level descriptions called motifs are mapped to the design patterns of the framework. The process steps themselves are based on the hooks and point to specific hooks in order to make the changes. The hooks map into the framework itself, so the transition from requirements, to architecture to implementation can be made smoothly. For example, in the Communication Performance step, the product developer decides on the specific communication needs, such as performance. The Communication Performance step directs the developer to, in this case, the Choose MailServer hook, in which an 'engine' best matching the requirements is chosen. The hook itself points to the places in the framework that it affects.

Ideally, the steps will identify single requirements (or sets of similar requirements) which can then be mapped directly to a hook. The refinement performed potentially offers an additional benefit. Requirements mapped to the framework during the process steps allow for requirements tracability. That is, each requirement identified is mapped to a single hook, which then points to a specific part of the framework that fulfills that requirement. Additionally, support for this type of process can be incorporated into a tool, not just for a single framework as in the case of GUI builders, but for frameworks in general. We have constructed the prototype of such a tool called Hook Master (Liu 1999) for supporting the enaction of hooks on frameworks.

CONCLUSIONS

Using hooks to encapsulate requirements in a generic design and implementation is extremely promising for product-line development. It helps to localize the effects of requirement changes and focuses a developer's attention on those requirements that are unique for the product under development. Traceability can be supported in a straightforward manner and with the assistance of a tool such as HookMaster.

Our approach has been tried out and refined in the development and use of several O-O frameworks including the *EAF* (Engineering Product Framework), *Prothos* (web delivery of database-centric applications), *Sandwich* (personal web assistant proxy framework), and *CSF* (Client Server Framework). We are continuing to experiment and improve many aspects of our approach. Future work includes: i) formalizing the hooks model so that model checking can be applied to ensure consistency and completeness of a product-line architecture, ii) extending the HookMaster tool to more explicitly support requirements traceability, iii) investigating the use of generic processes for O-O framework evolution, and iv) developing support for the creation and evolution of experience bases that can be associated with O-O frameworks.

REFERENCES

D. Batory, B. Lofaso and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In Proceedings of the **5th International Conference on Software Reuse**. (Victoria, Canada, June 1998).

M. Fowler. **Analysis Patterns: Reusable Object Models**. Addison-Wesley, Menlo Park, California. 1997.

M. Fowler. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, Menlo Park, California. 1999.

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

G. Froehlich, H.J. Hoover, L. Liu and P. Sorenson. Reusing Hooks. In **Building Application Frameworks**. M. Fayad, D. Schmidt and R. Johnson, ed. Wiley Computer Publishing, New York. 1999, 219-235.

G. Froehlich, A. Kamal and P. Sorenson, FrameScan: Exploring O-O Framework Usage. Located at www.cs.ualberta.ca/frames/papers.

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

K. Kang, S. Cohen, J. Hess, W. Novak and A. Peterson. **Feature–Oriented Domain Analysis (FODA) Feasibility Study**. (CMU/SEI-90-TR-21, ADA 235785). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

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).

W. Liew. **A Personal Web Assistant Framework**. MSc. Thesis, University of Alberta, 1999.

L. Liu. **Hook Master: A Tool to Support the Enactment of Hooks**. MSc. Thesis, University of Alberta, 1999.

L. Nakatani, M. Ardis, R. Olsen and P. Pontrelli. Jargons for Domain Engineering. In **Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99).** (October, 1999).

D.L. Parnas. On the Design and Development of Program Families. **IEEE Transactions on Software Engineering**, SE-2:1-9, March 1976.

S. Robertson and K. Strunch. Reusing the Products of Analysis. In **Proceedings of the Second International Workshop on Software Reusability**. (Lucca, Italy, March 1993).

Taligent. **The Power of Frameworks**. Addison-Wesley Publishing Company, Reading, MA. 1995.

D. Weiss. Commonality Analysis: A Systematic Process for Defining Families. In Proceedings of the *Second* **International Workshop on Development and Evolution of Software Architectures for Product Families**. (February 1998).