A Framework for Framework Documentation

Article *in* ACM Computing Surveys · March 2000

DOI: 10.1145/351936.351951 · Source: DBLP

CITATIONS
27

READS
668

3 authors, including:

Hafedh Mili
Université du Québec à Montréal
193 PUBLICATIONS **4,214** CITATIONS

SEE PROFILE

A Framework for Framework Documentation

Greg Butler
Department of Computer Science
Concordia University
and
Rudolf K. Keller
Département d'Informatique et de Recherche Opérationnelle
Université de Montréal
and
Hafedh Mili
Département d'Informatique
Université du Québec à Montréal

Frameworks are quite difficult to understand when one first uses them: the design is very abstract, to factor out commonality; the design is incomplete, requiring additional subclasses to create an application; the design provides flexibility for several hotspots, not all of which are needed in the application at hand; and the collaborations and the resulting dependencies between classes can be indirect and obscure. Many approaches to documenting frameworks have been tried, though with different aims and audiences in mind. In this paper, we present a task-oriented framework for framework documentation. Our framework is based on first identifying a set of framework (re-)use cases, which we then decompose into a set of elementary engineering tasks. Each such task requires a set of *documentation primitives*, enabling us to specify a minimal set of documentation primitives for each framework usage scenario. We study some major framework documentation approaches in light of this framework, identifying possible deficiences and highlighting directions for further research.

Categories and Subject Descriptors: D.2.m [**Software**]: Miscellaneous—*Reusable Software*; D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms: Documentation, Design

Additional Key Words and Phrases: application frameworks,design patterns,use case,CASE

Name: G. Butler
Address: 1455 de Maisonneuve Blvd West, Montréal H3G 1M8 Canada
Name: R.K. Keller
Address: C.P. 6128, succursale Centre-ville, Montréal H3C 3J7 Canada
Name: H. Mili
Address: B.P. 8888, succursale "A", Montréal H3C 3P8 Canada

1. INTRODUCTION

Frameworks are introducted to support the development of families of applications through reuse [Fayad and Schmidt 1997; Mili et al. 1995]. The design of a framework defines and fixes certain roles and responsibilities amongst the classes, as well as standard protocols for their collaboration. The variability within the family of applications at hand is factored into so-called "hotspots" [Pree 1995], and the framework provides simple mechanisms to customize each hotspot. A framework is not an easy thing to understand when one first uses it: the design is very *abstract*, to factor out commonality; the design is *incomplete*, requiring additional subclasses to create an application; the design provides *(too much) flexibility*, not all of which may be needed in the application at hand; and the collaborations and the resulting dependencies between classes can be *indirect* and *obscure*. Many approaches to documenting frameworks have been tried [Butler and Dénommée 1997; Campbell and Islam 1993], though with different aims and audiences in mind. These different audiences — application developer, framework maintainer, and framework developer — have quite differing needs from documentation.

In this paper, we propose a framework for specifying the documentation needed for each kind of framework usage, and we study existing approaches for the extent to which they address those needs. Our approach is based on an analysis of the various framework (re-)use cases, ranging from simple instantiation to the case of mining a framework for design ideas to use in other frameworks. For each such (re-)use case, we identify a set of elementary (software) engineering tasks, each with its own documentation needs; such needs are expressed in terms of documentation primitives. This setting is shown in Figure 1 using OMT notation.

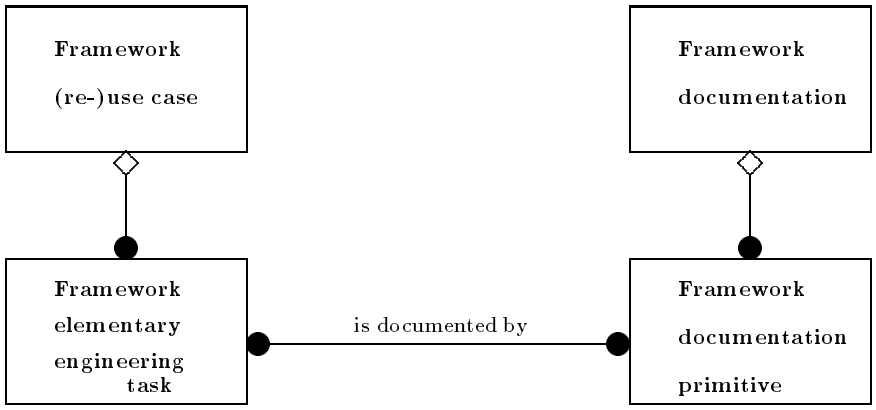

Fig. 1. Relating Elementary Engineering Tasks and Documentation Primitives

2. A DOCUMENTATION FRAMEWORK

Researchers have long recognized that different kinds of descriptions are required for different purposes. We have identified six categories of framework (re-)use [Butler 1997], each with their own demands on documentation.

Regular use.

—*Selecting*: A framework is selected as being appropriate for the intended application.

—*Instantiating*: A framework is instantiated for a particular problem by selecting concrete classes for the hotspots from an existing library, or by extending partially abstract classes in one of the ways planned by the framework designer.

Extended use.

—*Flexing*: A hotspot is customized in a way that was not planned for by the framework designer, yet the customization is consistent with the obligations and constraints of the design.

—*Composing*: Two frameworks are used in conjunction, possibly sharing participants and messages or message sequences [Mili et al. 1997].

Maintenance and reuse.

—*Evolving*: A framework is maintained to either increase the flexibility of existing hotspots, or to add new hotspots.

—*Mining*: A framework is mined for ideas that might be applicable in other contexts, such as a framework or a system for another domain [Keller et al. 1998].

2.1 Decomposing (Re-)use Cases into Elementary Engineering Tasks

The six kinds of (re-)uses identified above require increasingly more intimate knowledge of the structure and behavior of the framework, and are described in more detail below in terms of elementary engineering tasks.

Selecting: The selection process requires matching a description of the problem at hand to a description of the framework. If the framework is domain-specific, an enumeration of the participant classes may be sufficient to ascertain the appropriateness of the framework; we refer to this task as *data matching*. If the framework is domain independent and offers a service (e.g. persistence, graphical interfaces), or a service in a way that optimizes a design criterion, the matching is more complex and involves analyzing structure manipulation; we refer to it as *computational matching*.

Instantiating: The instantiation process requires filling in the hotspots. This may involve three tasks: i) *interface matching*, ii) *behavioral/computational matching*, and iii) *coding behavior*. Interface matching is used to ascertain that a concrete class supports all the methods expected by the other framework participants, and involves comparing signatures. It is similar to data matching, although far more detailed. Behavioral matching is concerned with the actual processing done by the individual methods to ensure that when a concrete class is used, the proper message sequence is triggered. Behavioral matching is similar to *computational matching*, although the latter is concerned with domain independent structure manipulation while the former may carry domain semantics. Coding behavior is concerned with implementing behavior according to some specifications.

Flexing: Flexing occurs when a concrete class matches the requirements of a framework participant, without the match being anticipated by the designer. This often means that the concrete class has the required computational structure, but does not carry the domain semantics envisioned by the designer. Hence, flexing substitutes domain semantics by generic structure manipulation before doing the matching. We call this translation step *domain abstraction*.

Composing: We limit our discussion to two clear-cut cases, *superposition* and *functional composition* [Mili 1996]. *Superposition* refers to the case where two frameworks share one or more participants, and where the same initial message triggers two message sequences, one within each framework. *Functional composition* refers to the case where two frameworks share participants in such a way that two message sequences are concatenated into a single serial one [Mili et al. 1997].

Evolving: The key to framework flexibility is the explicit separation (and implementation) of the variable parts of an application family from the fixed parts. That separation is never clear cut, and is usually refined through iterative *decomposition*, whereby a framework participant is split into two, a fixed part and a variable part. Thus, evolving often requires *domain abstraction* to identify a fixed part. From that point on, it involves almost the reverse operations of the composing case.

Mining: Mining consists of identifying subsets of the framework which may be reused within a different framework. It is useful to think of mining as a sequence of 1) selecting a subset of the participants, 2) performing *behavioral slicing*, and 3) performing *domain abstraction*. The second step isolates the message subsequences involving only the subset of participants identified in step one. The third step is used when moving to a different application domain.

The above descriptions are idealizations of what might be a sequence of messy and poorly structured cognitive tasks; they are useful nonetheless to help us identify the information needed for each elementary task and (re-)use case.

2.2 Framework documentation primitives

We now look at the various kinds of documentation primitives. We distinguish between two dimensions, "what" is being documented, and "how". Under the "how", we distinguish between two styles, *static descriptions* and *dynamic descriptions*.

—**Signature of the participants (SP)**: This consists of an enumeration of the interfaces of the participants. This kind of documentation is essentially *static*.

—**Behavioral specification of the participants (BSP)**: This is the description of the individual behaviors of the participants. A *static* description may involve pre- and post-conditions, or relations between inputs and outputs. A *dynamic* description may include the description of an algorithm or an execution trace.

—**Computational specification of the participants (CSP)**: This is a domain-independent and static behavioral specification of the participants. This might simply mean that the atoms of the description language are domain-independent.

—**Structural dependencies between the participants (SDP)**: This is described in terms of structural relations between the participants. A *static* description specifies the binding of the various participants to each other. A *dynamic description* typically shows an assembly sequence of the various participants—we call these *instantiation scenarios* in [Mili et al. 1997].

—**Behavioral dependencies between the participants (BDP)**: This describes inter-object behavior, which is often a consequence of the structural dependencies. A *static description* may be a set of behavioral axioms involving several participant methods. A *dynamic description* typically consists of so-called message sequences [Mili et al. 1997].

—**Computational dependencies between the participants (CDP)**: This is a domain-independent BDP.

Table 1 shows, for each elementary engineering task described earlier, the documentation primitives— and the specific style— needed to support the task.

Elementary engineering task	Documentation primitive					
	SP	BSP	CSP	SDP	BDP	CDP
Data matching	static					
Computational matching	static		static			
Interface matching	static					
Behavioral matching	static	static				
Coding behavior	static	dynamic		both	dynamic	dynamic
Domain abstraction	static	both			both	both
Superposition	static	both		static	both	both
Functional composition	static	both			both	both
Decomposition	static	both			both	both
Behavioral slicing	static	both		static	both	both

Table 1. Relating Elementary Engineering Tasks and Documentation Primitives

3. AN EVALUATION OF EXISTING DOCUMENTATION TECHNIQUES

In this section, we look at some major approaches that have been used to document frameworks. They may be goal-driven, or provide context or reference material, or provide descriptions or specifications of interfaces, interactions, architecture and micro-architecture. We first describe the various approaches, and then identify the documentation primitives they comprise. Based on this, we are able to determine which framework (re-)use cases each one of these approaches supports.

3.1 An overview of documentation approaches

The source code of *example applications* is often the first and only documentation provided to application developers. Documentation requires a graded set of training examples. Each should illustrate a single new hotspot, starting with the simplest and most common form of reuse for that hotspot.

A *recipe* describes how to perform a typical example of reuse during application development. The information is presented in informal natural language, perhaps with some pictures, and usually with sample source code. A *cookbook* is a collection of recipes. A guide to the contents of the recipes is generally provided, either as a table of contents, or by the first recipe acting as an overview for the cookbook.

The *interface contract* of a class provides a specification of the class interface and class invariants in isolation. A contract specifies the type constraints given by the signature of a method, and the interface semantics of the method.

An *interaction contract* deals with the co-operative behavior of several participants that interact to achieve a joint goal. It specifies preconditions on participants required to establish the contract, and the invariant to be maintained by these participants [Lajoie and Keller 1995].

A *design pattern* [Gamma et al. 1994] provides an abstraction above the level of classes and objects. Design patterns capture design experience at the micro-architecture level, by specifying the relationship between classes and objects involved in a particular design problem [Keller et al. 1998].

A *framework overview* defines the jargon of the domain and delineates the scope of the framework: just what is covered by the framework and what is not, as well as what is fixed and what is flexible in the framework. A simple application can be reviewed, and an overview of the documentation can be presented.

A *reference manual* for a framework consists of a description of each class, together with global variables, constants, and types. Typically, a class description presents the purpose or responsibility of the class, the role of each data member, and some information about each method.

3.2 An evaluation based on (re-)use cases

Table 2 shows the contents of the different documentation approaches in terms of the documentation primitives described earlier, and the style most likely used. The value "implicit" means that the documentation primitive at hand is embodied in the documentation available and may be extracted using a reasonable effort.

Documentation approach	Documentation primitive					
	SP	BSP	CSP	SDP	BDP	CDP
Example applications	static implicit			dynamic		
Recipe/cookbook	implicit	implicit	implicit	implicit	implicit	implicit
Interface contract	static	static		static		
Interaction contract	static			dynamic	dynamic	
Design pattern	static		both implicit	both		dynamic
Framework overview		static		static		
Reference manual	static	both implicit				

Table 2. Relating Existing Documentation Approaches to Documentation Primitives

Based on Table 2, we are able to summarize in Table 3, for each documentation approach, the kind of framework (re-)use cases supported by the approach. The entry "Y" indicates that the documentation approach strongly supports the framework (re-)use case, while an entry "y" indicates support (but not strong support).

4. DISCUSSION

The challenge is to build a documentation that is *orthogonal*, *cross-referenced*, and *economical*, both to build and to understand. The quality of documentation is an essential factor in the reusability of software. It is also the most difficult to formalize. The framework for framework documentation makes explicit the different kinds of framework use and reuse, the elementary engineering tasks involved, and the documentation primitives that support those tasks. We focus on the needs of development with reuse: how the existence of a framework modifies the software

Documentation approach	Framework (re-)use case					
	Select	Instantiate	Flex	Compose	Evolve	Mine
Example applications	y	y				
Recipe/cookbook	Y	Y				
Interface contract		y	y	y		
Interaction contract		y	Y	Y	Y	
Design pattern			Y		Y	Y
Framework overview	Y					y
Reference manual		y	y		y	

Table 3. Relating Documentation Approaches to Framework (Re-)use

process, what information is needed, how it is accessed and applied, and how the abstract descriptions map to concrete applications. This provides a critical assessment of documentation and support tools from the perspective of the reuser.

Research continues on tools that support the (re-)use of frameworks by an appropriate mix of documentation primitives. Perhaps our greatest challenge is the knowledge acquisition problem of obtaining the framework documentation from the framework developers.

ACKNOWLEDGMENTS

This work has been supported by the Natural Sciences and Engineering Research Council of Canada, and *Fonds pour la Formation de Chercheurs et l'Aide a la Recherche* of Québec.

REFERENCES

Butler, G. 1997. A reuse case perspective on documenting frameworks. *http://www.cs.concordia.ca/ faculty/gregb*.

Butler, G. and Dénommée, P. 1997. Documenting frameworks to assist application developers. In *Object-Oriented Application Frameworks* (New York, 1997). John Wiley and Sons.

Campbell, R. H. and Islam, N. 1993. A technique for documenting the framework of an object-oriented framework. *Computing Systems 6*, 4 (Fall).

Fayad, M. E. and Schmidt, D. C. 1997. Object-oriented application frameworks. *Communications of the ACM 40*, 10 (October), 32–38.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Keller, R. K., Tessier, J., and Bochmann, G. 1998. A pattern system for network management interfaces. *Communications of the ACM*. to appear.

Lajoie, R. and Keller, R. K. 1995. Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert. In *Object-Oriented Technology for Database and Software Systems*, pp. 295–312. World Scientific Publishing Co.

Mili, H. 1996. On behavioral descriptions in object oriented modeling. *Journal of Systems and Software 34*, 1 (July), 105–121.

Mili, H., Mili, F., and Mili, A. 1995. Reusing software: Issues and research directions. *ACM Trans. Soft. Eng. 21*, 6 (June), 528–561.

Mili, H., Sahraoui, H., and Benyahia, I. 1997. Representing and querying reusable object frameworks. In *Symposium on Software Reusability* (New York, 1997). ACM Press.

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