# Documenting Framework Behavior

Neelam Soundarajan

Computer and Information Science
The Ohio State University
2015 Neil Avenue Mall
Columbus, OH 43210
USA

e-mail: neelam@cis.ohio-state.edu
Tel: (614) 292 1444. FAX: (614) 292 2911

Frameworks [Johnson, Foote, Sparks] promise to dramatically reduce the time and effort needed to develop complete applications. A framework $\mathcal{F}$ for a given application area typically provides the *control flow* among the various methods of the various classes. A developer who wants to use $\mathcal{F}$ to develop a complete application $\mathcal{A}$ need only provide the code for the various (pure) virtual methods[1] in the various abstract base classes of $\mathcal{F}$. But in order for the promise of frameworks to be truly realized, the framework $\mathcal{F}$ must include documentation that provides the application developer with suitable information about $\mathcal{F}$. In the absence of such documentation, the application developer will be forced to go through the code of $\mathcal{F}$ to extract the information, thereby substantially negating the advantages that the use of frameworks was supposed to provide. In the rest of this article we point out the need for a new approach to the formal specification[2] of the behavior of frameworks, sketch a possible approach, and briefly indicate how an application developer could combine such a specification of framework behavior with appropriate information about the code he supplies to arrive at a specification of the entire application.

What is different about frameworks that necessitates the development of a new approach to their specifications rather than using the standard approaches, such as those discussed in, for example, [Meyer, Guttag, Leavens??] etc., that are used in dealing with 'normal' OO programs? The key problem is the critical role that virtual (and pure virtual) functions play in frameworks and applications built using them. While virtual functions can also, of course, be used in other OO programs, the standard approaches to OO specifications in a sense downplay the contribution that the definitions of the bodies of the virtual functions in the derived classes (which, in the case of frameworks, would be the code designed by the application developer) make to the overall behavior of the system. More precisely, suppose $f$ is a virtual function; in effect what the usual approaches to OO specifications do is to require that the designer of the base class of which $f$

[1]For concreteness we use *C++* terminology but the ideas in this paper are language independent; further, we will often use the terms 'method' and 'function' interchangeably.

[2]Formal specifications are often considered, perhaps with some justification, too hard to understand, and even harder to create, and many designers prefer *informal* documentation. Nevertheless, it is useful to develop the formal approach since insights gained in the formal approach can often be used effectively in informal specifications; thus, for example, loop invariants were developed as part of the work on formal specification and verification of procedural code and are used extensively in informal documentation. We will briefly return to this point towards the end of the article.

is a member provide a *sufficiently general* characterization of the behavior of $f$, perhaps in the form of pre- and post-conditions, such that redefinitions (or definitions if $f$ is pure virtual) of $f$ in derived classes satisfy this characterization. Further, and this is what makes these approaches unsuitable for use with frameworks, the *only knowledge* that clients have regarding $f$ is whatever is provided by this general characterization; in other words, the differences between the different definitions of the virtual functions is abstracted away. In the case of frameworks this would mean that *all* applications built on a given framework $\mathcal{F}$ would be equivalent to each other since the only differences between these applications is in how they define the various virtual functions of the abstract base classes of $\mathcal{F}$! Clearly we need a new approach, one that will allow us to distinguish between these applications depending upon the differences in the behaviors implemented by the various application developers in their respective definitions for the virtual functions.[3]

To make the discussion more concrete, let us introduce a simple model of frameworks. Although most real frameworks are too complex to fit this model, the model does include the essential aspects of frameworks. A framework $\mathcal{F}$ in our model will consist of a concrete *controller* class called $C$, zero or more other concrete classes $C_1, \ldots, C_m$, and one or more abstract classes[4] $A_1, \ldots, A_n$. The *controller* class $C$ will have a distinguished method run() which, as the name suggests, will determine how control flows among the various methods of the various classes of $\mathcal{F}$. In order to develop an application using this framework, we must define one or more concrete derived classes corresponding to each of $A_1, \ldots, A_n$. Let $CAi^j, j = 1, \ldots$ be the concrete classes corresponding to $A_i$.[5] We will denote by $\mathcal{A}'$ this set of concrete classes; the entire application $\mathcal{A}$ is thus made up of $\mathcal{F}$ and $\mathcal{A}'$. Finally, in order to use the application $\mathcal{A}$, a client will create an object d that is an instance of the controller class $C$, and apply the run function to it: d.run(). We should note that run() is usually a non-terminating function. *** something about member variables of type ?? in order to be able to invoke the other functions? Or maybe this can be explained as part of the next para when explaining the example?

A simple example may make our model clearer: The example is the diagram editor framework of Horstmann [Horst]. This is a fairly standard, if simplified, framework for editing diagrams consisting of *nodes* and *edges*. The framework provides the usual functionalities such as tracking the mouse movement, interpreting mouse clicks, etc. The framework contains two abstract classes, node and edge. An example virtual function in the node class, isIn(), allows us to determine if the current mouse position (which is supplied as a parameter to isIn()) is inside the specified node; the reason this is a (pure) virtual function is that the procedure for determining whether a given point is inside a node is something that very much depends on the type of the node in question (such as circle, smiley-face, etc.), and hence must be defined in the corresponding derived class;[6] another is, of course, the display() function; etc. The edge class of the framework has similar virtual functions. The main concrete class of the framework (our *controller* class) is called diagramEditor; the main

[3]This problem will likely show up in any OO system in which there is even just one virtual function; for ways of dealing with the problem that work in limited situations, but not for general frameworks, see [Stata, Tools].

[4]A class is abstract if it has one or more pure virtual functions; for simplicity, we will assume that all our virtual functions are *pure*.

[5]If for one or more of the abstract classes we did not have any concrete classes, we would have another framework –rather than an application– but one that would be more concrete than $\mathcal{F}$; we will ignore this possibility here.

[6]The only aspect of a node that is specified in the base class node is the coordinates of an *anchor* point of the node.

functionality provided by the framework is encoded in the run() function of this class. To build an actual diagram editor, all the application builder does is to define the concrete node and edge classes, in particular providing the definitions for the various virtual functions of the corresponding base classes. The client of this application applies the run() function on a diagramEditor object; when the run() function termiantes, that also terminates the application.[7]

Our approach, which we will summarize next, to documenting the behavior of frameworks has an important property: given an appropriate specification of $\mathcal{F}$, and an appropriate specification of $\mathcal{A}'$, the application developer will be able to combine them to obtain the specification of $\mathcal{A}$. This ensures that application developers do not have to reanalyze the code of $\mathcal{F}$ everytime a new application is developed using $\mathcal{F}$.

The key contribution that $\mathcal{F}$ makes to the application $\mathcal{A}$ is the *control flow* provided by the run() function. In a sense, this control flow acts as a *behavioral skeleton*, transferring control to the appropriate (pure) virtual functions of the various abstract classes $A_1, \ldots, A_n$ at appropriate times. What these functions will do in turn is decided not by $\mathcal{F}$ but by $\mathcal{A}'$. Effectively, the application developer, by providing specific bodies for these functions, has *refined* the behavior(al skeleton) of $\mathcal{F}$ to obtain $\mathcal{A}$. How do we specify $\mathcal{F}$? Let us introduce a *trace* variable $\tau_{run}$ to record the various calls that run makes and the corresponding returns; in other words, $\tau_{run}$ is a sequence whose initial value is the empty sequence, and each of whose elements represents either a call to a virtual function or a return. Suppose $f$ is such a function, and there is a call $x.f(\ldots)$ to this function. The call will be recorded as an element that specifies that $f$ is being called at this point, and the values of all the parameters, including the state of the object $x$; the return similarly is recorded as an element that specifies the values returned by $f$, including the state of $x$ when $f$ returns. Of course, we need some way to tell the call-element from a return-element; we will assume that the notation used for these elements allows for that. More important is the question, given that it is a pure virtual function, how can we know what values $f$ will return? Indeed we do not know; that is something that the *application developer* will decide when he provides the body of $f$ (as part of $\mathcal{A}'$). What we need to do when specifying the behavior of $\mathcal{F}$ is to make sure that we allow for *all* possible values that $f$ might return; unless we do this, each application designer will have to reanalyze the behavior of $\mathcal{F}$ and that is precisely what we want to avoid. To achieve this, we will specify the behavior of $\mathcal{F}$ as an *invariant* $I_{\mathcal{F}}$ involving the state of the object $d$ (recall that this is the 'main' object used by the client, and the run function is applied to it), *and* the trace $\tau_{run}$.

Conclusion: Need to work out details; but there are some other important problems too; we will mention two: generalize model – not too difficult; work on 'composing frameworks' – much more difficult and interesting.

[7]This is obviously a simplified description of Horstmann's framework; in particular we have left out initialization issues.

What is the key contribution that $\mathcal{F}$ makes to $\mathcal{A}$? If we can identify this, and if we can tailor our approach to the specification of $\mathcal{F}$ so that this aspect of $\mathcal{F}$ plays a central and explicit role in it, then the approach, as well as informal versions of it, are likely to be useful. As we said at the start of this article, the fundamental contribution that $\mathcal{F}$ makes to applications built on it is the *flow-of-control* among the various functions, especially the (pure) virtual functions. How do we capture this aspect of $\mathcal{F}$ without getting into other internal details of $\mathcal{F}$? Let us introduce a *trace* variable $\tau_{\mathcal{F}}$ that records the various calls to and returns from

Old:

What type of documentation? We can consider three kinds: Informal documentation tailor-made by $\mathcal{F}$'s designer for the particular framework; informal but *structured* documentation in the style, for instance, of [John92]; formal documentation along the lines of the literature on formal specification [see, for example, ??]. The first is probably the easiest, at least from the point of view of $\mathcal{F}$'s designer, but it is also the one that is least likely to be useful to the application developer. The potential dangers of informal documentation arising from possible ambiguities, incompleteness, etc. are well known, although often ignored.[8] In the case of frameworks the potential problems are even more serious because a misunderstanding, on the part of the application developer, of the behavior of the framework will make it almost impossible to build a correct application. The reason for this is the so-called 'Hollywood principle' [John97], i.e., which parts of the framework code and which parts of application code are invoked at what points is decided by the framework – not by the application code. Thus, while a software developer using a library can simply avoid using the portions of the library that he doesn't fully understand, such an option is not available to an application developer building an application on the framework $\mathcal{F}$.

What about the question of difficulty of creating formal specifications and of understanding them? There is indeed some validity to the claim that many, perhaps most, software designers find it hard to read, and even harder to create, formal specifications for systems of any degree of complexity. We believe the right approach is to seek middle ground: First develop an approach and the underlying formalism that *could* be used to formally specify and verify the behavior of frameworks and applications built on them; next, use the insights gained by the formalism in informal[9] but precisely structured ways such as proposed by Johnson [John92, John97]. In the rest of this article we will sketch a possible approach to the precise specifications of the behaviors of frameworks and applications built on them; in the last paragraph of the article we will briefly indicate how the main ideas underlying the approach can be used in informal specifications.

[8]The main reason why these dangers are ignored and why informal documentation is popular is, of course, the not entirely unfounded concern that *formal* specifications are hard to provide and hard to understand; we will return to this issue shortly.

[9]This would be similar to the use of loop invariants in informal specifications of procedural code; the notion of loop invariant is formal and precise –indeed the idea was developed as a key part of the axiomatic approach to program verification– but loop invariants, even those stated informally in English rather than in predicate calculus, can be extremely useful in informal specifications of such code.