

Minimalist Documentation of Frameworks

Kasper Østerbye
Norwegian Computing Center &
Department of Computer Science
Aalborg University
Kasper.Osterbye@nr.no

Abstract

Programmers are like most other humans - they prefer to *act* rather than *read*. In the context of application frameworks, this means that a programmer is more inclined to “try it out” than to read its manual. Rather than trying to instruct programmers to stop this all too human habit, this paper examine ways in which documentation can be organized so that a programmer is supported in learning through doing rather than learning through reading. The programmer is most open to study documentation during a breakdown of his understanding of the framework, and must in that case be led to the appropriate pieces of documentation. In the context of a concrete framework, the paper discusses how reference material, tutorial and run-time error messages and warnings are combined into a hypertext-based documentation of frameworks.

Keywords: online documentation, frameworks, software documentation, and hypertext.

The theme of the workshop is “Experiences in Object-Oriented Reengineering” with one of the sub-themes being documentation. As will be clear in the paper, reengineering was not in my mind when I originally wrote this. However, I believe the notion of tying code and documentation together with runtime errors is also an interesting idea to pursue in the situation where a legacy system is reengineered to become a framework. New documentation must be written at that point in time, and the code must be made more robust. The points in the code in which to improve robustness are the same places where I tie run-time checks to documentation.

1 Introduction

In *The Nurnberg Funnel*, John Carroll presents what he call the “paradox of sense-making”:

To learn, they (*the users*) must interact meaningfully with the system, but to interact with the system, they must first learn. [Carroll, 1990](page. 77).

Traditionally, this paradox has been attacked by ignoring it, and putting learning (reading the manual) before interaction. This seems like a logical choice - we have a long tradition of learning through studying. However, Carroll gives plenty of evidence that given a choice, people prefer to interact before they learn.

In this paper, we will examine how framework documentation can be organized in a manner that will allow the programmer to be active within minutes of starting with the framework and its documentation. In addition, we will discuss how to apply Carroll’s ideas of minimalist instruction in the context of framework documentation.

The main points in our approach are:

- Example based tutorial based on full running examples.
- A fairly traditional reference manual

- Extensive run-time checks in the framework, as a designer you must assume all rules will be broken
- Hypertext linking between reference manual and examples, and from runtime errors to manual and examples.

The main thesis behind this work is that breakdowns are an opportunity where the programmer is prepared to switch from doing to learning. This has led us to propose a way out of the sense making paradox by integrating runtime error checks and documentation, providing a concrete and goal directed entry into the documentation.

Software documentation can be divided into two broad categories, user documentation and internal documentation. Here internal documentation is taken to mean all kinds of documentation that is needed to maintain and further develop the software, whereas end-user documentation concentrates on documenting how to use the software. User documentation often consists of reference documentation as well as some kind of “introduction” material, e.g. tutorials, guidelines, cookbooks or pattern languages. The prime purpose of internal documentation is to capture the intention and solutions (why and how) of a large piece of software.

The focus in this paper is on a special kind of end-user documentation, documentation of object-oriented frameworks. This should not be confused with internal documentation although the user in the case of frameworks is a programmer. Programmers who are using a framework need user documentation for that framework. This paper addresses how to produce and organize such framework documentation.

This paper is organized as follows. Section 2 discusses our concrete framework, and its documentation is organized. In section 3, we discuss tools and techniques to make it easier to produce this kind of documentation. We then look on how we have attempted to use Carroll’s ideas in our documentation, and we relate this style of documentation to framework documentation using pattern languages in section 4.

2 The BetaSIM documentation

The BetaSIM framework [Østerbye and Kreutzer, 1996] is a framework for discrete event simulation in the BETA programming language [Madsen *et al.*, 1993]. The BetaSIM framework differs from other such frameworks in that it provides a higher level of abstraction with regard to synchronization than is normally found in discrete event simulation frameworks [Kreutzer, 1996].

The main abstraction provided by the framework is the concept of an *entity*, which is used to model real-world entities with their own life cycle. The framework provides classes that realize resources with exclusive access, and most prominently, it provides servers that allow rendezvous style synchronization, where a client and a server are synchronized during calls to the server. The server abstraction also allows interrupts, and servers can be grouped to allow several servers to cooperate in providing a common set of services.

The documentation of the BetaSIM framework is online only (through WWW), and consists of two different “documents”, the tutorial and the reference manual. The tutorial is six pages, which through annotated examples take you through the main aspects of the framework, and the reference manual is a rather traditional description of all classes and methods of the framework. The rest of this section addresses these two documents in detail, and concludes with how run-time checks are integrated into the overall documentation of the framework.

2.1 Tutorial

The main point of Carroll's work is that users, in this case programmers do not want to read documentation. The BetaSIM tutorial starts out with an example that can be compiled and run, and indicates what can be changed without the program breaking down.

Only then does the tutorial spend some time explaining what the program does. To keep the tutorial terse, it does not give the exact meaning of all aspects of the framework, but focuses on overall effects. This is deliberate - to support a more detailed understanding, all usage of framework classes and methods are linked to the reference manual. This makes it easy to seek more information, or not to do so if one understands the program as it is.

As an example, consider the following extract from the tutorial:

The Gate is an EntitySource. This means that it will produce other entities.

```
<Gate declaration>=  
Gate : @ | EntitySource  
  (# EntityType :: Dignitary;  
   productionFrequency::Exponential  
   (# init::(# do 5 -> mean #) #);  
  #);
```

We specify that the EntityType to be produced is Dignitaries.
The production frequency is exponential with mean 5.

Here a component called a "Gate" is discussed. It is briefly said what it does. The important thing, that it is an EntitySource, is stressed by making links to the reference manual both in the code and in the running text. In a typical web browser, such links will stand out as well. In order to guide the user to the important aspects first, there are no links from say *init* or *EntityType*. To find out what these are, the user must follow the link to EntitySource, and then go on from there.

The tutorial is build as a number of increasingly complex variations on the same simulation scenario. That way we avoid having to spend time introducing the scenarios, and can focus on the modeling issues involved.

2.2 Reference manual

A reference manual must satisfy several important criteria. It must be complete, correct, up to date, and not the least, it must be well indexed.

The reference manual for BetaSIM is organized with one web page per class and has separate entries for each method and local class (a BETA specific construct).

WaitQueue

Class

Origin:

BetaSim

Super:

NamedObject

Queue for suspending entities for later resumption.

While it is safe to use this queue directly at the simulation level, it is often more preferable to use a resource than a Queue.

Attributes

init

Further bound virtual Procedure. Inner: Voluntary HOOK.
Initialize Queue

empty

Procedure. Super: BooleanValue. Exit: Boolean.
returns true if queue is empty

wait

Procedure.
Insert executing entity in queue.

start

Procedure.
Starts first element in the queue.

The above page illustrates the documentation page for a class named `WaitQueue`. In BETA, there is no syntactical difference between a class and a method, both are described using a construct called a *pattern*. One of the important aspects of the documentation is to clarify the intended usage of a given pattern, as class, procedure, process, abstract class, etc. This is the second piece of information given.

Besides this classification, an entry will describe the input and output values for methods, and for virtual and abstract procedures¹, a description of their intended specialization is given. Because `WaitQueue` is a class, such information is not given here, but when available it will be in brief form together with the origin (link to outer block in the block structure), and super class information.

All patterns have a one-line description, and many have a more elaborate description. If the pattern (class, method, or process) has local public patterns, these are listed in a short form as well. Each local pattern (or public variable) are listed with an abbreviated description, listing the “tabular” facts, classification, parameters, super etc, and the one line description. If the local pattern has local attributes it self, or more extensive documentation, it will be described on a page by itself, which is why *wait* and *start* are linked, and *init* and *empty* are not. Under *init*, the phrase “Further bound virtual Procedure” is underlined to indicate a hyperlink. As *init* is a virtual procedure, there might be information relating to the semantics in the *init* procedure of the super pattern, which is then made accessible through this link.

Besides linking to other pages to give the inheritance structure, links are also provided to allow navigation in the block structure. Some classes have other classes declared local to them, and this is reflected in providing a short description, and a link to a full-page description.

The index contains all public identifiers². If the same identifier is used in several classes, it is listed several times, and the class name is appended. E.g., the identifier *Finish* is used for three different procedures:

[Finish; .BetaSIM](#)
[Finish; .BetaSIM.Entity](#)
[Finish; .BetaSIM.EntitySource](#)

The first occurrence states that *Finish* is a procedure available in the framework itself, and the next two that *Finish* is declared in *Entity* and *EntitySource* as well. The entries in the index are hypertext links to the appropriate places in the reference manual.

¹ Because procedures, like classes, are described using patterns, procedures can be specializes as well. This is a BETA only. See [Østerbye and Kreutzer, 1996] for how that is used in the BetaSIM framework

² BETA has no concept of public or private modifiers, encapsulation is handled using other means. In the documentation, public identifiers refer to those documented.

2.3 Runtime errors

The main observation behind the paradox of sense making is that people prefer to act rather than to study. In the context of frameworks, this means that programmers prefer to “try things out” rather than reading a manual. This will most likely lead the programmers using a framework to violate some of its design decisions and assumptions. In the BetaSIM framework, this situation has been seen as an opportunity to instruct the programmer as to what has gone wrong, and to suggest possible solutions.

This is done by performing extensive error checking in the framework. Two levels of error message are given: a terse description of the problem, and an elaboration of the error, together with a list of possible solutions and common misunderstandings.

The solutions and explanations are kept short, but with links to the reference manual and to the tutorial where possible.

As an example, let us follow the error in the next piece of BETA program written using the BetaSIM framework.

```
Friend: Entity
  (#
  do cycle(#
    do Family.accept;
    Mingle.HOLD
  #)
#)
```

It is beyond the scope of this paper to explain the syntax of BETA. However, the “(# ” and “#) ” can be read as “begin” and “end”. “do” is a BETA keyword.

The example is an erroneous declaration of a Friend entity. A friend is an element in a simulation scenario of a garden party held by the royal family. A number of dignitaries and personal friends compete for the attention of the royal. The error is that the Friend tries to *accept*. The family is the server here, and friends are clients, only the members of the royal family will be able to *accept* requests to either chat or shake hand.

When the program is run, it will produce an error message stating:

Simulation error - see file “BetaSIM-error.html”.

When the “BetaSIM-error.html” is loaded into a WWW browser, the user will see the message below.

<p>BetaSIM-error</p> <p>Friend-1 tried to “accept” a call for Family.</p> <p>Friend-1 is not a server, and only servers are allowed to accept calls.</p>
<p>More information</p>

The error message is specific in that it uses names from the application rather than the framework. It says that a friend — here friend-1 — has tried to accept a call for Family. Emphasis has been given on providing concrete error messages, that is, we use the scenario terms Friend and Family as opposed to entity and server, which are framework level concepts.

The “More information” link leads to the following page:

<p>Accept from non-server</p> <p><u>Accept</u> is part of rendezvous communication. The <u>Bored Queen</u> shows an example of proper usage. If you got this error, it was because you tried to:</p>
--

- Use remote access, as in “SomeServer.accept” which is not legal under any circumstances.
- You have declared an entity local to a server, and that entity does an “accept”. That entity should most likely have been declared a local server. The Queens family shows an example of local servers.

The problem is shortly described, and two common sources for the error is mentioned, where the first one corresponds to our case. From this page, there are links to the tutorial material as well as to the reference-manual entry for *accept*.

Both the brief tutorial and the runtime error messages encourage the programmer take control of the learning process. The tutorial by being example driven, and it is highly unlikely that the example will be directly applicable to any another scenario, but it does show one way to use the framework. The error messages are concrete and at the level of the scenario program, with terse error messages and links to only those part of the documentation that the programmer is likely to have misunderstood.

3 Tools and techniques

This section will briefly discuss the techniques used to produce the documentation for BetaSIM. The section does not contribute at the scientific level, but provides some insight into practical and technical issues in producing this kind of documentation.

One issue to consider when producing documentation for any software artifact, is how to keep the documentation and the artifact consistent. Donald Knuth has proposed literate programming [Knuth, 1984] as a way to solve this problem. From the maintenance point of view, literate programming is interesting because it builds on the proximity-principle, that is, documentation and code are written together in a single file, an documentation is kept textually together with the aspect is documents. Either a literate programming tool can extract a program or produce a document containing both source code and documentation nicely typeset. Documentation extractors that extract documentation extracted from comments in the source code itself to produce reference manuals is an other kind of documentation tool that follows the proximity-principle. Examples of this include the Eiffel short [Meyer, 1988](page 205 ff), JaveDoc [Friendly, 1995], and the George system (See URL 1). Both literate programming and document extraction has been used in the development of the BetaSIM documentation.

The reference manual is extracted from the source code of the framework by a document extractor we have developed for the BETA programming language. Like most other document extractors, it is able to extract documentation from special comments, and will extract some information from the framework itself, to reduce redundant entry of such things as class and method names and super class relation ships.

To produce the tutorial we have used Norman Ramsey’s literate programming tool *noweb* [Ramsey, 1994]. This tool is particularly well suited for our purpose as it is programming language independent and can produce HTML output.

In using *noweb* to produce the tutorial, we have encountered two problems:

- The tutorial is made up by a single scenario, which is elaborated into an increasingly complicated synchronization scenario. We would therefore like to be able to extract several different programs from the tutorial, without having to state each in full. We have found no good method of doing this in *noweb*, and have in the tutorial a full example program for each variant. This is not satisfactory, and we believe that a specialized literate tool is necessary to produce tutorials and other example based instruction material.

- To allow links from the example code to the reference manual, it is necessary to place HTML anchors inlined in the BETA source code which makes the BETA compiler choke quite a lot. To address this problem, we do not use HTML anchors directly, but use macros which are expanded to anchors when documentation is generated, and to empty strings when BETA source code is generated.

Birtwistle has produced one of the first example based introductions to an object oriented discrete event simulation framework [Birtwistle, 1979]. In his book, variations are used a lot as well, but no attempt at using literate programming is done, as literate programming was not developed at the time.

It is important to use identifier names from the application program when reporting errors. We have here exploited a special BETA idiosyncrasy, which enables us to find the application name of objects and procedure invocations so we do not need to resort to using framework terms. Notice that if no error checks were done, error messages would neither be in application nor framework terms, but in terms of the underlying implementation language BETA.

The framework does substantial amounts of runtime checking, to capture situations that should not arise in a debugged simulation program. For reasons of efficiency, it must be possible to disable such checking later in the development process.

4 Discussion and related work

This paper has focused on the documentation of the external interface for an object-oriented framework. According to [Sametinger and Stritzinger, 1993] such documentation must have four parts --- overview and interface each addressing both static and dynamic aspects. However, their distinction between documentation of static and dynamic is unconvincing. The paper calls the dynamic interface description “the task interface description”, and the accompanying discussion focuses on cookbooks as a documentation technique. This leads us to believe that the fundamental distinction is not between *static* and *dynamic* aspects, but between Aristotelian and prototypical descriptions.

At a very fundamental level, there are two schools for how to describe concepts, e.g. an elephant. Elephants can be defined as being huge, with thick legs, a trunk, big tusks and large ears. Alternatively, you can define elephants as being all those that are similar to Dumbo. Defining a concept by its properties is sometimes called the Aristotelian perspective and defining a concept through examples is called the prototypical perspective.

In relation to documentation of frameworks, we believe that a framework must be documented from both perspectives. The Aristotelian perspective will result in a reference manual, where all properties of the framework are documented. The prototypical perspective gives examples of what concrete examples look like, i.e. show at least one whole elephant, or how to go about solving concrete problems, i.e. how to understand your particular problem as a trunk or a leg. The reference manual for BetaSIM satisfies the Aristotelian perspective, while our example based tutorial is prototypical. The runtime error messages relate the user's example to both the reference manual and the tutorial, encouraging the programmer to investigate both aspects to seek a solution.

The drawing editor framework HotDraw is documented using a pattern language in [Johnson, 1992]. Pattern languages focus on a set of problems usually encountered during a development in some particular area, in this case Smalltalk programs that use a drawing editor. The pattern language starts with general problems and points out solutions using the framework and then states what are the typical issues to be addressed once this problem has been solved. Typically, pattern languages do not include full examples, though they might include example fragments. Compared to our approach, it is worth noticing that neither our tutorial nor Johnson's pattern languages are meant to stand by themselves. Where we depend on the reference manual, Johnson depends on the programmer having access to the source code of the HotDraw framework, allowing documen-

tation of the classes and methods that make up the framework to be browsed on-line. Where we have chosen an example-based approach to providing prototypical documentation, Johnson has chosen pattern languages. Pattern languages are broader than an example based documentation, and can cover more aspects. However, this makes the pattern language harder to grasp if one does not have a good overall understanding first.

Continuing with the elephant metaphor - if we have a "build your own animal kit", with bodies, legs, etc. An example-based tutorial might say, here is an elephant, and this is how it is made. The pattern language will say that all animals have a body, here are the different kinds of body, and we have to choose one. Then proceed with how to put legs on a body, etc. The example-based approach is good if you have no clue as to what an animal is in the first place, but seeing an elephant might not help you build a giraffe.

To address this issue, we have included run-time errors as part of the documentation. If you build an animal with tail at both ends, you get an error message that tells you why that will not work.

An interesting aspect of documentation that has become more popular with the spreading of the Internet is documentation through FAQ (Frequently asked questions). Almost all aspects of programming now have a FAQ available somewhere on the Internet. The typical FAQ are organized as a question answer pair, with the question being a real problem faced by some person, and the answer the actual piece of information that helped to solve the problem. As there are more and more problems arising, the FAQ can evolve from shorter FAQ like that of Grant's CGI Framework (See URL 2) into rather large documents like the FAQ for visual basic (See URL 3).

One fundamental issue is that it can be very hard to predict what documentation is needed. It is all too easy to over-document a framework, writing lots and lots of documentation nobody will read. From a practical point of view, it is not possible to write a good FAQ before the framework has been used by real users. The same is true for a pattern language approach, if the patterns are to be fully useful, they must be based on experience of what problems arises in a real work situation. It is easier to develop an example based documentation, but it might fail because it off target. However, an example-based documentation can be devised based on the designers feeling for what the framework is supposed to do. As real users start to use the system, the example-based documentation can be supplemented by a FAQ. The use of runtime checks in BetaSIM can be seen as an attempt to make a build-in basic FAQ.

In his book on object-oriented software construction [Meyer, 1988], Bertrand Meyer makes a strong case for the usage of contracts in the form of pre- and post-conditions and class invariants. His argument is that proper usage of these semantic mechanisms will prevent extensive tests to see if parameters are as expected. If you develop a real valued square root function, is it the function or the user of the function that must check to ensure that the argument is not negative? If both do, the program becomes inefficient.

From a learning perspective, however, it is a good idea to have a way to inform the novice programmer that the framework will not do any checking for negative arguments once runtime checking is turned off.

In Carroll's work on minimalist documentation [Carroll, 1990], he present several ideas that we have adopted or which are similar to what we have done. The similarities include the focus on error recovery, getting started fast, encouragement of self-exploration, training on real tasks, and on attempting to coordinate between messages and training. Therefore, it is our feeling that there is indeed things that can carry over from his work, although most of it focuses on documenting an application where we are focussing on documenting a framework. Some differences do arise though. In our framework we have not been able to divide the functions into novice and advanced, which is a prerequisite to using the blocking technique Carroll discuss in connection with the "training wheel" system. In the BetaSIM framework, the users must suffer their errors before the framework can provide any help. Despite Carroll's recommendations, the error-checking messages

do try to be intelligent, in that they include common reasons for the error to occur. Carroll emphasizes the technique of exploiting prior knowledge. This has most likely been taken a step to far in our documentation, in that we expect the user to be both familiar with the programming language (BETA) and with discrete event simulation. Perhaps it is a property of all frameworks, that the documentation will assume either knowledge of the programming language, or of the application domain of the framework, but asking both is most likely wrong.

Finally, the usage of a framework is a very open ended task, and in Simulation, the difficult aspect is often to model rather than to program. Though we have not discussed this aspect in this paper, key parts of the reference manual have links to a set of discrete event simulation patterns which specifically addresses the issues of modeling for simulation [Kreutzer, 1996]. In [Carroll, 1994] Carroll discusses the usage of end user scenarios as a means for guiding design. As mentioned in the above discussion on using FAQ's for documentation, one particular documentation problem is to anticipate what the users will find difficult. The usage of scenarios can help to point out what is often done, and should therefore be easy to do. We see the example-based tutorial as going one step further and actually using scenarios/examples as part of the documentation.

The paper [Østerbye, 1995] addressed how hypertext could be used for internal documentation, and showed how hypertext was an especially useful tool in documenting solutions that involved more than one class. The main difference between that work, and the work reported here is that the present work addresses issues relating to user documentation of frameworks, where the [Østerbye, 1995] paper focussed on using hypertext for internal documentation.

5 Conclusion and Further work

The main outstanding issue is to perform a validation test on this approach to documentation, in particular how runtime errors are perceived and used by programmers. We have argued that the documentation follows almost all the principles of minimalist instruction, a principle which reportedly does improve learning time as well as understanding [Carroll, 1990].

Our documentation has been created using a language specific tool for the reference manual part, and a general literate programming tool for the tutorial part [Ramsey, 1994]. The BETA language [Madsen *et al.*, 1993] is hardly mainstream. Can the approach be generalized to other languages and tools. As already said, the technique of extracting reference manual from the source code is well known. To fully use literate programming for example based tutorials, a specialized literate tool is needed that has better support for variations. In summary, there is nothing particularly specific about the way our documentation we have created that will make it impossible to do in other languages. The reference manual layout and structure is specific to the individual programming language, but that is all.

6 References

- [Birtwistle, 1979] G. M. Birtwistle. *A System for Discrete Event Modeling on Simula*. Macmillian Press, 1979.
- [Carroll, 1990] J. M. Carroll. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. The MIT Press, 1990.
- [Carroll, 1994] J. M. Carroll. *Making Use a design representation*. Communications of the ACM, 37(12):29-35, December 1994.
- [Friendly, 1995] Lisa Friendly. The design of distributed hyperlinked programming documentation. In *International Workshop on Hypermedia Design*, 1995, page 159, June 1995. Proceedings to be republished summer 95 by Springer Verlag.

- [Johnson, 1992] R. E. Johnson. *Documenting frameworks using patterns*. In OOPSLA'92 Proceedings, 1992.
- [Knuth, 1984] Donald E. Knuth. *Literate programming*. The Computer Journal, 27(2):97-111, May 1984.
- [Kreutzer and Østerbye, 1996] Wolfgang Kreutzer and Kasper Østerbye. *BetaSIM - a framework for discrete event modeling & simulation*. Under review.
- [Kreutzer, 1996] W. Kreutzer. *Foundations and patterns in discrete event modeling and simulation*. In Proceedings of PLOP workshop, September 1996.
- [Madsen *et al.*, 1993] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison Wesley, 1993.
- [Meyer, 1988] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall International, Inc., 1988.
- [Ramsey, 1994] Norman Ramsey. *Literate programming simplified*. IEEE Software, 11(5):97-105, 1994.
- [Sametinger and Stritzinger, 1993] J. Sametinger and A. Stritzinger. *A documentation scheme for object-oriented software systems*. OOPS Messenger, 4(3):6-17, July 1993.
- [Østerbye, 1995] Kasper Østerbye. *Literate Smalltalk Programming Using Hypertext*. IEEE Transactions on software engineering 21(2):138-145, February 1995.
- [Østerbye and Kreutzer, 1996] Kasper Østerbye and Wolfgang Kreutzer. *Synchronization abstraction in the BETA programming language*. Under review.

7 Referenced URLs

1. *George - Automatic Source Code Documentation*
<http://www.k2.co.uk/products/George/George.html>
2. *Grant's CGI Framework*
<http://arpp1.carleton.ca/cgi/framework/faq.html>
3. *Visual Basic*
<http://puta.gurunet.org/vb/>