Software Documentation Guidelines

In addition to a working program and its source code, you must also author the documents discussed below to gain full credit for the programming project. The fundamental structure of these documents is entirely independent of project, programming language, and operating system. You will find a number of advantages when you pursue a rigid documentation approach to programming. First of all, you will have a firm understanding of the task at hand before you start coding. A good understand of the problem leads to a clean design that tends to have fewer bugs. Always make your goal to program it right the first time! The next advantage is that others will be able to use your documentation to test the program, fix bugs, and make enhancements. In the corporate world, these duties are normally performed by different people and often by different groups within a single company. Therefore, the more detailed, organized, and easy-to-read your documentation is, the more you help other people do their jobs. As you learn to *write* solid documentation, you will also come to appreciate *reading* solid documentation, and will eventually detest reading technical crap (the world is full of poorly written technical books and manuals). In other words, write simply and clearly. The way you write is just as important as the details you present. Always strive to spell correctly and use proper grammar. The campus Writing Center can aid you in this respect.

User Requirements Document (URD)

This document describes the problem from the user's point of view. It briefly describes the problem domain, e.g.. a psychology experiment or a small business accounting package. Then the document delivers a simple and exact description of the problem. After the problem description, the user states exactly what he/she would like the software system to do. While this may seem to indicate a user interface, it is better to focus on the tasks to be solved rather than the interface required to solve them. However the user may require a specific interface, e.g.. a GUI rather than a command line interface. The crux of this document is to identify what the user requires of the program, and not what the user requires of the programmer. This document furnishes the programmer with a formal description of the problem. Ideally, this document would be written by someone in marketing who has talked with a customer; not by a programmer. The most important thing to remember is that this document describes the functionality required of the program.

The URD has:

- user's view of the problem
- brief description of the problem domain
- complete description of the problem
- what is expected from a software solution
- what is not expected from a software solution

The URD does not have:

- programmer's point of view
- programming jargon or technical details
- description of programming languages or environments unless it is a specific user requirement
- description of the solution. This is not a design document. We only want requirements here.

Back to top

Requirements Analysis Document (RAD)

This document takes the URD as a starting point and looks at the problem from a designer's point of view. However, instead of diving directly to implementation details, the analysis focuses on the system and software requirements needed to implement the user requirements. This document gets detailed, but does not delve into programming details. Instead, take the user's requirements and clearly identify all of the details and mitigating factors that will affect the solution that the user wants. An analysis may indicate a preference for a particular programming language that best suits the problem domain rather than an algorithm to satisfy a particular requirement. The RAD looks at the URD as defining an entire system, and then breaks the URD down into bite-size chunks (divide and conquer). These chunks identify the subsystems of the overall solution, and the relationships between them. But the RAD also goes further and identifies the actual details of the problem that the user may not be aware of.

The RAD also maps the domain of software systems onto the user requirements. For example, the RAD may indicate that a database is needed for a particular subsystem, or that an expert system can satisfy certain other requirements. The RAD is written from the designer's perspective. An astute software designer is one who is aware of available software systems and paradigms. He/she should know what types of systems and solutions work best in different environments. The RAD, then, identifies the software systems and paradigms that will best fit the user requirements. The RAD doesn't design a solution; it merely identifies the most beneficial means for an implementation.

The RAD has:

- designer's interpretation of the user's requirements: identify the "real" problem(s)
- breakdown the problem into high level constituent parts
- deep analysis of these parts and identification of all relevant details
- identify existing solutions
- identify alternative technical solutions
- link these solutions to the problem(s), especially with respect to details
- suggest the best solution and break it into parts
- devise ways to test the solution

The RAD does not have:

- user's point of view

- implementation details
- algorithms
- user interface specification

Back to top

User Interface Specification (UIS)

This document describes exactly what the user interface is going to do, what it looks like, and how the user interacts with the program. The UIS does not describe how the interface is implemented. Nor does it describe what the program does behind the interface. Rather, the UIS focuses in detail specifically on the user interface itself. For a GUI, the UIS would define the components and all options on a MenuBar, all of the MenuBar headings, all of the submenus, and all of the options of those submenus. The UIS may describe the functionality of each of the mouse buttons, if appropriate. The UIS also describes the presentation of data, be it graphics, text, or a combination. The UIS should be understandable to the user. The UIS should contain drawings or screen captures of prototype interaces.

For non-GUI applications, the user interface may be either a Command Line Interface (CLI) for which the UIS could be similar to a UNIX man page, or could be an Application Programmatic Interface (API). An API is really nothing more than a collection of library routines that allow you to link and/or layer software components. For an API, the UIS consists of a definition of the calling interfaces, names of routines or object methods, parameters, and descriptions of what they will do.

The UIS has:

- description of the complete user interface, CLI, or API
- what the user interface looks like to the user
- how the user interface behaves
- how the user interacts with the system
- if GUI, names for all of the interactive components, from the mouse to buttons to menus and scrollbars, and pictures of what the interface should look like
- if CLI, flags and arguments, inputs and outputs as in a UNIX man page
- if API, complete description of the link/call interface

The UIS does not have:

- implementation details
- coverage of the mechanics beneath the user interface

Back to top

Prototype

Once you know what the user interface (GUI, CLI, or API) is going to look like, go ahead and try to build a "shell" of the user interface as quickly as possible. This is called a prototype. You don't have to implement the core functionality - you just code the interface to see 1) if it's possible and 2) how it appears to the user. Ideally, you should end up throwing this prototype away. The prototype should be shown to the user to see if it's what the user had in mind. That's why you really shouldn't put too much effort into it because chances are the user is going to say, "No, I wanted it to look like this" or

"I like this screen, but that one sucks", etc. You want to ensure that the user sees what they want to see - and this applies to GUI, CLI, and API. It's much easier to change a prototype than it is to do all the work on the internals first and then have to change things. The prototype helps you design things in such a way that the interface is independent of the underlying implementation of the actual problem. The interface is not the solution to the problem. But the interface is the user's window to that functionality. That functionality, or solution to the problem, begins with the next step.

Object Oriented Analysis (OOA) or High Level Design (HLD)

The OOA applies an object-oriented view to the problem. The easiest way to do this is to pick out all of the nouns in the RAD. Each noun is usually an object. Through out nouns that aren't substantive objects. The next step is to write a detailed description of each object, no matter how trivial it may seem or how much you take this object for granted. Each object must be completely and succinctly documented. This is called a *data dictionary*. Next, look for overlap between objects and remove objects that are not important to the problem domain. For example, take out user interface components. You will deal with those separately. Once you have removed unnecessary objects, identify their attributes and methods. More often than not, you will find that some of the objects you have are merely attributes of other objects. Next, establish relationships between objects. For example, an employee works for a company. Here, both company and individual are objects. An individual *works for* a company. A company *employs* an individual. A company *has* many workers. An individual usually works for a single company. Here, we have both defining and numerical (cardinality) relationships. Find such relationships between your objects, and define the cardinality, eg. one-to-one, one-to-many, many-to-many.

The next step of the OOA is to do the same thing with your UIS. The user interface and underlying application subsystems should be completely independent of one another. In fact, you should be able to design and develop your interface and your underlying application independently of one another. You should make your user interface classes as generic as possible, and subclass off of them to get application-specific behavior. For example, to make a list of colors you would probably use a generic List object. A List has a series of text string labels for each choice, lets the user make a choice, activates some function when such a choice is made, allows the user to add an item to the list, delete one, and so on. So far, you could use this List object in almost any application that requires a List interface object. However, for a color selector, you may want to show the actual color in the rectangular slot where you usually show the textual name of the color. In this case, design the object hierarchy such that most of the functionality is encapsulated in the List object, but then subclass a ColorList class to present the colors rather than the textual labels. Design interactive objects in a similar way: allow an application to register actions with an object rather than defining that an object performs specific tasks. Most of the appearance and functionality is then abstracted away from the end user in a particular program. This way you can share an object between applications, and assign application-specific behavior with registration rather than explicit coding.

If you are not using an object oriented programming language like C++ or Java, you can still use object oriented design techniques. However, if you prefer, you can write a High Level Design document in place of the OOA. The HLD accomplishes many of the same goals, but from a non-OO approach.

The OOA has:

- data dictionary defining exactly what each object represents (in English)
- class diagrams showing the name, attributes, and methods of each class
- relationships between the classes, depicting graphically and textually
- a set of class diagrams and data dictionary for the application domain
- a set of class diagrams and data dictionary for the interface domain
- end-user readability

The HLD has:

- detailed breakdown of technical solution in subsystems
- descriptions of data structures and operations required for eachs subsystem
- detailed interaction between subsystems, including interface subsystem(s)

The OOA and HLD do not have:

- algorithms
- implementation details

Back to top

Object Oriented Design (OOD) or Low Level Design (LLD)

The OOD is as close to coding as you can get without actually coding. If you do this document correctly, the code will just fall out naturally. The OOD takes the classes in the OOA a level deeper into the realm of pseudo-code. The OOD defines the datatypes for the attributes. The OOD also defines the algorithms and implementation details of the class methods. However, you are not writing code yet. The OOD should be language-independent. You are merely specifying more exactly what the attributes and methods consist of. Like the OOA, the OOD addresses both the interface and application class hierarchies. Also like the OOA, it keeps the two domains separate and independent of one another.

The OOD goes another step beyond the OOA by identifying the *objects*, rather than the *classes*, required to implement the software system. While the user may be able to comprehend the OOA, the OOD goes very deep into software design. The OOD is much more detailed than the OOA and establishes instantiations from the class hierarchy and their relationships to one another, especially associations and cardinality.

The OOD also provides the algorithms for all class/object methods pseudo-code. For a graphical environment, the OOD would specify how the event loop dispatches events to specific objects. For example, a mouse-down in a drawing canvas would activate the pen object to start drawing, while a mouse-up would terminate the line. The OOD often makes use of state and event diagrams that define exactly what happens when the user interacts with the graphical components. It maps the user interactions with graphical components to underlying application subsystems. For the drawing example, the OOD maps the mouse down to the pen object that draws the line. The mouse doesn't draw the line, the pen object does, using the mouse coordinates as a guide.

For non-OO design, the LLD contains essentially the same contents as the OOD, namely, explicit detail of all datatypes and functions. You should provide psuedo-code for all algorithms and flesh out all aspects of the programming effort without yet resorting to actual code. If you were to

compare this documentation process to writing a research paper, you could regard the HLD as the chapter and section titles, the LLD as the bullet items for each section, and the code as the text of the paper. Each document just gets more and more detailed.

The OOD and LLD have:

- inner details of class attributes (datatypes) and methods (functions)
- detailed object (as opposed to class) diagrams for OOD
- state diagrams
- event diagrams
- pseudo-code
- algorithmic descriptions

The OOD and LLD do not have:

- code

Back to top

Code Documentation (CD)

You are expected to fully document your code. Every class and class method should have a name, a brief one-line description, and a detailed description of the algorithm. All methods also require descriptions of all inputs and outputs. If applicable, you should also note any caveats - things that could go wrong or things that the code doesn't address. Put assumptions in the caveats section. If you are coding in Java, you should use the documentation tags that work with the javadoc utility. This utility automatically generates web pages for your documentation. To make things consistent, simply cut and paste the textual descriptions of your classes, objects, and methods from your OOD directly into the code. Then let javadoc do the dirty work. If you are not coding in Java, you can still use the same tags and see if javadoc operates on your source files. Otherwise, you could write such a utility yourself!

Back to top

Testing Documentation (TD)

The TD describes how you tested your program to prove that it works sucessfully. You should include testbeds for both the user interface and application aspects of your program. You should also provide test datasets where applicable. For example, if you program does some form of text processing, you should provide example file inputs that test specific features of the program requirements. You should pay special attention to borderline values and bogus user input. The TD should also include evaluation criteria, so you know what you are actually testing for.

Back to top

User's Guide (UG)

This document tells a user how to use your system. The format is up to you. Pick your favorite User's Guide and copy its format. You may even want to use the UNIX man page system. Assume that the user knows nothing about your project. Provide an overview and then details of each

subsystem. Make sure you explain how the whole system works before delving into the details. This document is for the end-user or another programmer, so you have to explain the obvious, e.g.. Basically, you have to tell someone how to use your system.

[Back to top]