Module 7

Software Evolution; Documentation

Objectives:

1. Understand the interdependence of change and software usefulness.
2. Understand Lehman's laws of software evolution.
3. Know the types of software maintenance.
4. Understand what software rejuvenation is and what constitutes it.
5. Understand the importance of software documentation.
6. Know the difference between user documentation and system documentation.
7. Know some of the potential reasons why software documentation often does not serve its intended purpose, and what can be done to resolve those issues.

Reading:

Sommerville, chapter 9, Web Chapter 30 (Documentation, available on the companion website: http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/WebChapters/index.html).

Journal article: A Rational Design Process: How and Why to Fake It, by David Parnas and Paul Clements. (I've provided a copy on Blackboard, but you can also find it with a Google search. Interestingly, you can NOT get this paper from the IEEE Xplore digital library, because their archive for some reason does not contain any of the issues of IEEE Transactions on Software Engineering from the year 1986.)

Assignment:

No assignment for Module 7.

Quiz:

Take the quiz for Module 7.

The Maintenance Phase

Software maintenance consists of all the tasks that must be performed on software *after* it has been released. There will almost surely be a few bugs which must be fixed. As people use the software they will get ideas for new features that should be added, or existing features that should be changed, or removed entirely. During the time the software is used, a greater understanding may be developed about the problem the software was designed to solve. This understanding may necessitate changes in the software's functionality. At some point the operating system the software runs on, or other software with which it must interact may be changed, and the software must adapt. Business rules may change, which may require changes in all or portions of the software. It could be that the software that was released was only a rough version that was released quickly to make sure it got to market on time. Further refinements may be needed to ensure the software keeps its market share.

It's tempting to believe that once software hits the maintenance phase most of the effort required for the project has already been expended. After all, to get to the release stage developers had to go through requirements elicitation, analysis/design, implementation, and testing. The only phases left are maintenance and the death phase (when the software gets phased out). It may be surprising to hear that for many projects the effort expended from requirements to testing is only a drop in the bucket. Some estimates indicate as much as 80% or more of the total effort expended on a software project occurs during maintenance.

When software is changed during the maintenance, it is also called **software evolution**. The term evolution is used to demonstrate that software must adapt, or evolve in order to remain useful, much like living things must evolve to adapt in order to survive.

Lehman's Laws

Lehman and Belady have spent a great deal of time studying the concept of software evolution. The set of 8 laws they proposed are very insightful. The first law, the *Law of Continuing Change*, states that if software is used in some environment, it must change or it will become less useful in that environment over time. Intuitively, this makes sense. We've already discussed how difficult it can be to achieve a complete understanding of the problem a given software system was built to solve, so it's highly unlikely the first version of the software will actually be correct. It may satisfy all the requirements that were known at development time, but as the software is used greater understanding of the problem can be achieved, which may involve adding new requirements, or changing or removing existing requirements. Use of the software will also bring to light inefficiencies or other problems that could not have been foreseen when the software was developed. Even if the software is correct, what happens if its environment changes? Suppose the operating system changes, for example, and the changes are not compatible with the software system. The software must be changed to adapt to the new environment. What if business rules change, rendering a portion of the software obsolete? The software must be changed to function according to the new rules.

The *Law of Continuing Growth* goes hand in hand with the Law of Continuing Change. Continuing growth means that users will not be satisfied if the software's functionality does not change to meet the demands of the users. This law may also partially explain why many people are compelled to always buy the most recent version of a software application.

The *Law of Conservation of Familiarity* is especially interesting. This law states that on average, software changes at a pretty much constant rate. The reasoning is that every change has the potential to introduce new faults, and the number of new faults is directly proportional to the size of the change. If small changes are made, the faults can be fixed fairly quickly. Large changes generate enough bugs that no more functionality can be changed until all the bugs are fixed, which may mean that an entire new release may have to be dedicated to bug fixes. Another interesting side note to this law that I've seen proposed is that for many software systems, the vast majority of the functionality is contained in the first release. The functionality added over subsequent releases doesn't significantly change the software from a size standpoint. I'm sure this does not hold true for all software, but I believe it is accurate for some.

The *Law of Increasing Complexity* states that software becomes more complex as it is changed. The main reason complexity increases is because the original, finished software has been carefully designed and

thoroughly tested (we hope). Changes can corrupt a fine-tuned system much like impurities can clog a well-tuned engine. This occurs more frequently with software that has not been designed using good engineering practices such as modular design. When modules have a high degree of coupling with other modules, changing one of those modules can upset the balance, which will require making changes in other modules, which may upset the balance in yet other modules, and so on. Not every change will increase the complexity. For example, a change that alters the color of one part of the user interface is not likely to change the complexity. It will most likely amount to nothing more than replacing one color value with another. Additions to the software's functionality will increase the complexity simply by virtue of increasing the software's "mass"; that is, there will be more to it. Subtracting from the functionality will only increase complexity if other parts of the software depended on the functionality that is removed. In any event, as changes accumulate the software may eventually become unstable. At that point it is usually necessary to expend enormous effort overhauling the software.

Types of Software Maintenance

Sommerville talks about three categories of software maintenance:

1. Corrective maintenance, which involves fixing bugs (faults).
2. Adaptive maintenance, which must be performed when the software's environment changes.
3. Functionality maintenance, where the software's functionality is modified

Adaptive maintenance and functionality maintenance can also take other forms, where changes that are not strictly necessary are made. These changes can be perfective, meaning they are made to make the software function more efficiently. For example, a sequential search algorithm may be replaced by a more efficient binary search algorithm. The changes can also be preventive, meaning they are made in an effort to make the software more resistant to potential problems that may be anticipated to occur in the future.

Software Rejuvenation

Maintaining software can be very expensive. What does an organization do if they can't afford to pay the costs of software maintenance? It may be possible to extend the life of software by "rejuvenating" it. Often the process of rejuvenating software is less than the cost of upgrading it. Software rejuvenation can occur in several forms:

<u>Redocumentation</u>

Here, the source code of the software is analyzed, and additional comments may be added to improve understanding. The goal is to enable the organization itself to make modifications to the code rather than buying a new version or paying for the vendor to maintain the software. Of course, this is only possible if the source code is available and the contract under which the software was acquired permits such modifications.

<u>Restructuring</u>

Restructuring goes one step beyond redocumentation. Code that is poorly written is rewritten to make it more structured or more efficient. Again, this is only possible if the code is available and such changes do not violate any software contracts.

<u>Reverse Engineering</u>

Reverse engineering essentially means engineering part or all of the software in reverse. For example, the byte code in a Java .jar file can be decompiled to produce Java source code. In many cases the source code produced is identical, or nearly so, to the original code. In some cases binary executables can be disassembled to yield the equivalent assembly language, but not the high level source code that was used originally. Source code can be reversed engineered to yield designs, and it is even possible to generate a set of requirements, although these will likely differ from the requirements and designs originally used

to build the software. Success is not guaranteed; rather, the degree of success depends on how much information can be extracted from the reverse engineering process. The whole process is like doing an archaeological dig, where bones are found, assembled to produce a skeleton, and then hypotheses are proposed as to what the creature may have actually looked like. In most cases developers prohibit reverse engineering of their software, for obvious reasons. Usually words to that effect can be found somewhere in the license agreement for the software.

Reverse engineering is not meant to be an acceptable method of building software. It should be reserved for software that must be modified, no designs or source code are available, and doing so will not violate any license agreements or other binding laws. Good candidates for reverse engineering are legacy systems that are no longer supported by their manufacturers, or whose manufacturers are no longer in business.

Re-engineering

Re-engineering is the natural consequence of reverse engineering. Once a system has been reverse engineered, changes can be made to modify, or re-engineer the system.

Documentation

I've harped about documentation with respect to the group project all semester so far. I will continue harping about it this week, and afterwards I will probably not talk about it much more. Chapter 30, found only as a web chapter on the companion website, is dedicated to the importance of documentation.

Documentation for a software project can be divided into two broad categories: 1) user documentation and 2) system documentation. User documentation consists of the user manual, instructions for installation of the software, documented system requirements, and known bugs and technical issues. In other words, user documentation consists of everything a potential user needs to know in order to install, use, and troubleshoot the software. For those cases where the end product is a component library targeted for software developers, an API is also typically provided, and in some cases even the source code may be made available.

System documentation consists of all documentation that is generated during the development process (sometimes I refer to this documentation as the programmer's manual). This includes the requirements document, the requirements specification, design documents, source code, test harness code, test suites, configuration management documentation, etc. Since the user documentation is one of the end products of the development process, a copy of the user documentation is usually included as part of the system documentation. In short, system documentation must contain everything a developer would need to know in order to maintain the software once it has been released. The engineers responsible for maintaining the software may not be the same engineers who developed the software in the first place, meaning they may have to depend solely on the system documentation to tell them what the software is supposed to do and how it is supposed to do it.

Common Problems with Software Documentation

A 1986 paper co-authored by David Parnas discusses some of the major problems the authors believed were present in software documentation of the time [1]. This is a very enlightening paper, underscoring many of the topics we've discussed so far in this course, and providing evidence against the use of the waterfall model. Among other things, the authors attempt to answer the questions, "What is wrong with most documentation today? Why is it hard to use? Why isn't it read?". The answers they came up with were:

- Poor organization

  Much documentation tends to be written haphazardly as "streams of consciousness", which unfortunately doesn't do much to help readers understand what is going on.

- It's boring to read

  If you've ever tried to really explain something in precise detail, accounting for all the possible cases, this probably makes perfect sense to you. As boring as it is to write such detailed, accurate documentation, it is even more boring to read it.

- Confusing terminology

  Often, those who document software fail to set up standard terminology before the documentation is actually written. This leads to inconsistent terms, confusing terms, and aliased terms in the final documentation, and this problem is compounded the more engineers there are on a team.

- Overdocumentation of minute details and underdocumentation of the bigger picture

  The authors refer to this as "myopia", meaning that as a project goes on, engineers tend to assume the big-picture details are common knowledge, and they don't document them. However, they do disproportionately document the little details right in front of them. This may work fine until someone who wasn't affiliated with the project needs to refer to the documentation, and are unable to find the answers to their questions because those answers all disappeared into the minds of the engineers who originally built the project.

The authors go on to suggest that these problems can be solved by:

- creating a set of standards that guide the creation of the terminology, the format of the documentation, and the types of content that should be present;

- adhering to an organizational scheme to prevent the documentation from de-evolving into streams of consciousness;

- using tables, diagrams, mathematical formulae, and formal notation to increase the density of the information.

Care must still be taken when implementing any of the above solutions. If the organizational scheme becomes too fine-grained, it will be too tedious to use, since too much time will be spent checking to make sure all the rules have been properly followed. This problem can be ameliorated by using templates to control formatting issues, and text generators to automatically produce whatever text can be auto-generated. Diagrams and tables, like most things, can be overdone, or done inappropriately. Flow charts, for example, used to be a staple of software design, but they have somewhat fallen out of popularity, since they tend to take significantly more time to generate than writing the equivalent idea in pseudocode.

References

1. Parnas, D. L. and Clements, P. C. (1986). A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering* 12:2, 251-257.