The Risks Of Stopping Too Soon

David Lorge Parnas

Software Development suffers from an infirmity best described as "*Premature Termination"*; the symptoms are that developers begin to do something useful but stop too soon. The result is something that is not only not useful, but often harmful.

There are three obvious causes for this illness:

- The work that is done before termination is the easy part of the task; what remains undone would require tedious, detailed work.
- Those doing the work have not been taught how to do the job correctly or how to determine when the work is complete.
- Those who review the work, or purchase the product, do not insist on proper completion of the key tasks.

Premature termination can be observed throughout the development, deployment, and post-deployment improvement of software products. The problem is also present when experts advocate and describe a software development process. In this column, I will describe some of the disease's manifestations in requirements documentation, diagrammatic description of software, interface documentation, and quality control.

Requirements elicitation and documentation

Nowhere is premature termination more evident than in the field sometimes called "Requirements Engineering" (RE).

In RE, developers identify properties that they want a system to have and assemble a list of these wishes. Often, the wishes conflict; they are almost always too vague to tell the programming team what to build. Requirements lists include statements such as:

- "The system must be easy for clerks to use."
- "The system must contain an UNDO command."
- "The product shall allow the user to select a chosen language."

Each of these examples leaves many questions unanswered. Among them:

- What will be the training of the clerks that will use the system? What information will they have at hand? What characteristics of the interface would make the system easy for them to use?
- How many past actions should one be able to undo? Does the requirement apply to all commands or only to a proper subset of those commands? If a subset, which subset? Should you be able to undo the most recent command if the file was closed after it was executed and then reopened? Should you be able to undo a command without undoing the subsequent commands first?
- What languages (and variants) should be offered)? Should it be easy for a user to add languages? Should the hyphenation rules change with the language? Should the keyboard interpretation change when a user switches languages?

If the requirements document does not answer such questions, it is will be the programmers who determine what the system does. Programmers are chosen for their ability to design good algorithms and data structures as well as their knowledge of specific languages and support environments. They should not be expected to understand the needs and characteristics of the future users. Often, a programmer makes incorrect guesses about the detailed requirements and consequently, extensive revisions are required (either before deployment or after users complain).

A list of wishes like the examples listed above is not adequate as a requirements document. Even if the questions posed above are answered, there is no way to check that there are no other relevant questions. The completeness and consistency of such a list cannot be verified. Many things are always left unstated. Such lists are a good start but considerable work must be done before one has resolved the conflicts and ambiguity in such a list to produce a complete and precise software requirements document, one that tells the programmers what they must build to satisfy the agreed wishes. It isn't wrong to produce a wish list but it is wrong to hand it over to the programmers as if it were their requirements document.

Drawing pictures of programs

The debate over whether pictures are a useful way to document programs is an old one that never seems to get resolved. The issue came up again in a recent column by Grady Booch. His plea for better pictures reminded me of a presentation by the late Edsger Wybe Dijkstra around 1975. At a meeting of IFIP W.G.2.3, a working group on programming methods, Dijkstra advised against drawing pictures saying, “Every time someone draws a picture to explain a program, it is a sign that something is not understood.” I found this surprising; during my education in Electrical Engineering, we were often shown how to use a diagram when designing or analyzing a proposed design. Those diagrams were sufficiently meaningful that one could derive equations from them. On the other hand, I could not find a diagram describing a program that did not raise more questions than it answered. All were so vague that it was very likely that two people would look at a diagram and interpret it differently. Most raise more questions in my mind than they answer.

Dijkstra's observation struck me as so thought provoking that, when I returned to my group in Germany, I repeated it to them. One of my associates, Wolfram Bartussek, responded immediately with a German version. However, in his “translation” he changed the statement's emphasis. “Yes,” he said, “drawing a picture is what you do when you are trying to understand a program or trying to help someone else understand it.[1]” Reflecting on this, I found it was true. A picture is often very helpful when trying to understand a complex problem. Bartussek had not contradicted Dijkstra's observation but he had explained why Dijkstra's advice (not to draw diagrams) was wrong. Diagrams can be a good starting point.

Subsequent experience with software documents deepened my understanding of the problem with diagrams of software systems. A few years later, I was asked to review a project that had repeatedly missed deadlines. In a series of meetings with key designers, I began by asking each one to draw a diagram that explained the workings of the system. I used a Polaroid camera to take a picture of each drawing. When each meeting began, the new expert studied the diagram on the board from the previous meeting and said, “That's an interesting picture but its not our system,” I showed them the pictures that I had accumulated but they found those no better. Each then asked to erase some of the existing diagram so that they could draw another one.

Those photos are faded now, but it is still clear that the diagrams are all different, all are vague, and none of them contains enough information to allow someone to understand what its creator

[1] This is an informal translation, not Bartussek's exact words.

meant. In a few cases, an expert reused part of the previous picture but the discussion revealed that he was interpreting it differently from the person who had originally drawn it.

When I am presented with "box and line" diagrams that are open to many interpretations, I ask that the picture be completed by adding a legend, i.e. an explanation of what property something must have to be represented by a box and what relation must exist between two boxes if there is a line connecting them. When the pictures come back, they have been altered to have several distinct box shapes and types of arrows. When the authors try to create the legend, they realize that they were using one symbol to represent several different kinds of objects or relations. The original picture had been a "buzz-diagram"[2]; the new one is always better than the first but usually not good enough. It takes many iterations before a reviewer can understand what the author is trying to convey. Only then, can we begin to discuss the design. When we do discuss it, more changes are made to the diagram. Some of those changes are design changes but others are adding arrows and boxes that had been overlooked.

Often, when the necessary information is added, the diagram becomes so cluttered and complex that it no longer helps people to understand the system. In such situations, the authors convert the diagram to some other representation of the information (often tables). The resulting document becomes an essential basis for subsequent work on the product.

The lesson should be clear; a picture is a good way to begin to understand something but most software developers stop too soon. They work on the picture until it means something to <u>them</u> but then stop even if it (a) does not contain all of the needed information or (b) does not communicate clearly to others. Often when they add a legend, it becomes clear that the diagram is not an accurate description of the design and that the design can be improved. In the end, another form of representation may replace the picture.

Talking about "the architecture"

Related to the problem of pictures is to talk about "the architecture" of the system. The phrase is often used as if there were a single structure that can be so identified. In fact, as has been discussed in [2, 3] there are many distinct, separately designable, structures and each should be described separately.

Software documentation

Problems caused by premature termination are found in all kinds of software documentation. For example, if you read the descriptions of methods in a Java library, you can usually pose questions that can only be answered by experimenting with an implementation or reading the code. So called "odd cases" are either not covered or ambiguously covered. For example, one often finds a list of exceptions that a class will "throw" but it is not clear if cases such as zero length string arguments are exceptions or what happens if two of the listed exceptions could be "thrown". Generally, it appears that the documenter started to write down a description but quit when they got to the tedious parts.

Testing and inspection

We can recognize the problem of premature termination in testing and inspection. The constant stream of "updates" (usually a euphemism for corrections) that we are asked to install is evidence that the developers stopped testing and inspecting too soon. Inspections are often considered finished when the participants are able to estimate the number of remaining errors and the

[2] A buzz-diagram, like a buzzword, has the property that most people think they understand it but are unable explain it clearly when asked to do so.

estimate is small. One honest tester answered my question, “How do you know when to stop testing?” with “When the schedule says there is no more time.” Another replied, “When my boss says to stop.” These events can occur long before the software is trustworthy enough to merit release. Testing is continued by the users.

Incomplete advice

It is particularly vexing that the problem of premature termination is also found in the writings and talks by people who claim to be telling practitioners how to do their job better. Many books and papers describe development processes with good advice such as “determine the characteristics of the users”, but they stop without telling you how to do it, how to document that information, or how to know when you have completed the task. Most of the books and papers contain buzz-diagrams describing the process and examples of work products that are themselves incomplete and unclear. Many of these “gurus” sell their methods by claiming that software design will be easy. Good software design is never easy and stopping too soon is easy in the short run but makes the job harder in the long run.

“Stopping too soon” never stops

The disease of premature termination seems to be immune to itself; it never stops. There are some obvious reasons for this.

- It is often hard to detect premature termination in a review because the work is not wrong; it is just not enough. There is an old joke that is applicable, “This is so bad its not even wrong!”. It applies to things that do not say much.
- Many managers don’t demand the disciplined, careful, complete work that they are entitled to demand. Cost and schedule are their primary concern. Short-term cost is easy to measure; long-term cost is unknown and wont affect the next pay raise.
- When selling methods, gurus are so eager to gain converts (and customers) that they try to make their approach seem easy and fun. They eschew anything that looks like tedious “dog work” or has even a hint of mathematics. The work that is left undone has both of those characteristics.
- Educators in Universities are so eager to give a complete survey of the available methods that they are (unavoidably) shallow. Rather than pick a strong method and teach it thoroughly they teach a little about a lot. Some also avoid anything that looks like “theory”. Some make a point of teaching students how things are currently done in industry and avoid teaching improved methods.

References

1. Booch, Grady, “Draw me a Picture”, IEEE Software, February 2011.02
2. Klein, J., Weiss, D., “What Is Architecture?”, Chapter 1 of Beautiful Architecture, Spinellis & Gousios, eds., O’Reilly, January 2009
3. Parnas, D.L., “On a 'Buzzword': Hierarchical Structure”, *IFIP Congress '74*, North Holland Publishing Company, 1974, pp. 336-339.
    - Reprinted as Chapter 8 in [4]
    - Reprinted in *Software Pioneers: Contributions to Software Engineering*, Manfred Broy and Ernst Denert (Eds.), Springer Verlag, Berlin - Heidelberg, 2002, pp. 501 - 513, ISBN 3-540-43081-4.
4. Hoffman, D.M., Weiss, D.M. (eds.), “*Software Fundamentals: Collected Papers by David L. Parnas*”, Addison-Wesley, 2001, 664 pgs., ISBN 0-201-70369-6