A RATIONAL DESIGN PROCESS: HOW AND WHY TO FAKE IT

David L. Parnas
Computer Science Department
University of Victoria, Victoria BC V8W 2Y2 Canada

and

Computer Science and Systems Branch
Naval Research Laboratory
Washington DC 20375 USA

and

Paul C. Clements
Computer Science and Systems Branch
Naval Research Laboratory
Washington DC 20375 USA

I. THE SEARCH FOR THE PHILOSOPHER'S STONE: WHY DO WE WANT A RATIONAL DESIGN PROCESS?

A perfectly rational person is one who always has a good reason for what he does. Each step taken can be shown to be the best way to get to a well defined goal. Most of us like to think of ourselves as rational professionals. However, to many observers, the usual process of designing software appears quite irrational. Programmers start without a clear statement of desired behavior and implementation constraints. They make a long sequence of design decisions with no clear statement of why they do things the way they do. Their rationale is rarely explained.

Many of us are not satisfied with such a design process. That is why there is research in software design, programming methods, structured programming and related topics. Ideally, we would like to derive our programs from a statement of requirements in the same sense that theorems are derived from axioms in a published proof. All of the methodologies that can be considered "top down" are the result of our desire to have a rational, systematic way of designing software.

This paper brings a message with both bad news and good news. The bad news is that, in our opinion, we will never find the philosopher's stone. We will never find a process that allows us to design software in a perfectly rational way. The good news is that we can fake it. We can present our system to others as if we had been rational designers and it pays to pretend do so during development and maintenance.

II. WHY WILL A SOFTWARE DESIGN "PROCESS" ALWAYS BE AN IDEALISATION?

We will never see a software project that proceeds in the "rational" way. Some of the reasons are listed below:

(1) In most cases the people who commission the building of a software system do not know exactly what they want and are unable to tell us all that they know.

(2) Even if we knew the requirements, there are many other facts that we need to know to design the software. Many of the details only become known to us as we progress in the implementation. Some of the things that we learn invalidate our design and we must backtrack. Because we try to minimize lost work, the resulting design may be one that would not result from a rational design process.

(3) Even if we knew all of the relevant facts before we started, experience shows that human beings are unable to comprehend fully the plethora of details that must be taken into account in order to design and build a correct system. The process of designing the software is one in which we attempt to separate concerns so that we are working with a manageable amount of information. However, until we have separated the concerns, we are bound to make errors.

(4) Even if we could master all of the detail needed, all but the most trivial projects are subject to change for external reasons. Some of those changes may invalidate previous design decisions. The resulting design is not one that would have been produced by a rational design process.

(5) Human errors can only be avoided if one can avoid the use of humans. Even after the concerns are separated, errors will be made.

(6) We are often burdened by preconceived design ideas, ideas that we invented, acquired on related projects, or heard about in a class. Sometimes we undertake a project in order to try out or use a favourite idea. Such ideas may not be derived from our requirements by a rational process.

(7) Often we are encouraged, for economic reasons, to use software that was developed for some other project. In other situations, we may be encouraged to share our software with another ongoing project. The resulting software may not be the ideal software for either project, i.e., not the software that we would develop based on its requirements alone, but it is good enough and will save effort.

For all of these reasons, the picture of the software designer deriving his design in a rational, error-free, way from a statement of requirements is quite unrealistic. No system has ever been developed in that way, and probably none ever will. Even the small program developments shown in textbooks and papers are unreal. They have been revised and polished until the author has shown us what he wishes he had done, not what actually did happen.

III. WHY IS A DESCRIPTION OF A RATIONAL IDEALISED PROCESS USEFUL NONETHELESS?

What is said above is quite obvious, known to every careful thinker, and admitted by the honest ones. In spite of that we see conferences whose theme is the software design process, working groups on software design methods, and a lucrative market for courses purporting to describe logical ways to design software. What are these people trying to achieve?

If we have identified an ideal process but cannot follow it completely, we can still follow it as closely as possible and we can write the documentation that we would have produced if we had followed the ideal process. This is what we mean by “faking a rational design process”.

Below are some of the reasons for such a pretense:

(1) Designers need guidance. When we undertake a large project we can easily be overwhelmed by the enormity of the task. We will be unsure about what to do first. A good understanding of the ideal process will help us to know how to proceed.

(2) We will come closer to a rational design, if we try to follow the process rather than proceed on an ad hoc basis. For example, even if we cannot know all of the facts necessary to design an ideal system, the effort to find those facts before we start to code will help us to design better and backtrack less.

(3) When an organisation undertakes many software projects there are advantages to having a standard procedure. It makes it easier to have good design reviews, to transfer people, ideas, and software from one project to another. If we are going to specify a standard process, it seems reasonable that it should be a rational one.

(4) If we have agreed on an ideal process, it becomes much easier to measure the progress that a project is making. We can compare the project's achievements with those that the ideal process calls for. We can identify areas in which we are behind (or ahead).

(5) Regular review of the project's progress by outsiders is essential to good management. If the project is attempting to follow a standard process, it will be easier to review.

IV. WHAT SHOULD THE DESCRIPTION OF THE DEVELOPMENT PROCESS TELL US?

The most useful form of a process description will be in terms of work products. For each stage of the process, this paper describes:

(1) What product we should work on next?

(2) What criteria must that work product satisfy?

(3) What kind of persons should do the work?

(4) What information they should use in their work?

Management of any process that is not described in terms of work products can only be done by mindreaders. Only if we know which work products are due and what criteria they must satisfy can we review the project and measure progress.

V. WHAT IS THE RATIONAL DESIGN PROCESS?

This section describes the rational, ideal software design process that we should try to follow. Each step is accompanied by a detailed description of the work product associated with that step.

The description of the process that follows includes neither testing nor review. This is not to suggest that one should ignore either of those. When the authors apply the process described in this paper, we include extensive and systematic reviews of each work product as well as testing of the executable code that is produced. The review process is discussed in [11,17].

A. Establish and document requirements.

If we are to be rational designers, we must begin knowing what we must do to succeed. That information should be recorded in a work product known as a requirements document. Completion of this document before we start would allow us to design with all the requirements in front of us.

1. Why do we need a requirements document?

(1) We need a place to record the desired behaviour of the system as described to us by the user; we need a document that the user, or his representative, can review.

(2) We want to avoid making requirements decisions accidentally while designing the program. Programmers working on a system are very often not familiar with the application. Having a complete reference on externally-visible behaviour relieves them of any need to decide what is best for the user.

(3) We want to avoid duplication and inconsistency. Without a requirements document, many of the questions it answered would be asked repeatedly throughout the development by designers, programmers and reviewers. This would be expensive and would often result in inconsistent answers.

(4) A complete requirements document is necessary (but not sufficient) for making good estimates of the amount of work and other resources that it will take to build the system.

(5) A requirements document is valuable insurance against the costs of personnel turnover. The knowledge that we gain about the requirements will not be lost when someone leaves the project.

(6) A requirements document provides a good basis for test plan development. Without it, we do not know what to test for.

(7) A requirements document can be used, long after the system is in use, to define the constraints for future changes.

(8) A requirements document can be used to settle arguments among programmers; once we have a complete and accurate requirements document, we no longer need to be, or consult, requirements experts.

Determining the detailed requirements may well be the most difficult part of the software design process because there are usually no well-organised sources of information.

2. What goes into the requirements document?

The definition of the ideal requirements document is simple: It should contain everything you need to know to write software that is acceptable to the customer, and no more. Of course, we may use references to existing information, if that information is accurate and well organised. Acceptance criteria for an ideal requirements document include:

(1) Every statement should be valid for all acceptable products; none should depend on implementation decisions.

(2) The document should be complete in the sense that if a product satisfies every statement, it should be acceptable.

(3) Where information is not available before development must begin, the areas of incompleteness should be explicitly indicated.

(4) The product should be organised as a reference document rather than an introductory narrative about the system. Although it takes considerable effort to produce such a document, and a reference work is more difficult to browse than an introduction, it saves labour in the long run. The information that is obtained in this stage is recorded in a form that allows easy reference throughout the project.

3. Who writes the requirements document?

Ideally, the requirements document would be written by the users or their representatives. In fact, users are rarely equipped to write such a document. Instead, the software developers must produce a draft document and get it reviewed and, eventually, approved by the user representatives.

4. What is the mathematical model behind the requirements specification?

To assure a consistent and complete document, there must be a simple mathematical model behind the organisation. The model described here is motivated by work on real-time systems but, because of that, it is completely general. All systems can be described as real-time systems - even if the real-time requirements are weak.

The model assumes that the ideal product is not a pure digital computer, but a hybrid computer consisting of a digital computer that controls an analog computer. The analog computer transforms continuous values measured by the inputs into continuous outputs. The digital computer brings about discrete changes in the function computed by the analog computer. A purely digital or purely analog computer is a special case of this general model. The system that will be built is a digital approximation to this hybrid system. As in other areas of engineering, we can write our specification by first describing this “ideal” system and then specifying the allowable tolerances. The requirements document treats outputs as more important than inputs. If the value of the outputs is correct, nobody will mind if the inputs are not even read. Thus, the key step is identifying all of the outputs. The heart of the requirements document is a set of mathematical functions described in tabular form. Each table specifies the value of a single output as a function of external state variables.

5. How is the requirements document organized?

Completeness in the requirements document is obtained by using separation of concerns to obtain the following sections:

(1) Computer Specification: a specification of the machines on which the software must run. The machine need not be hardware -- for some software this section might simply be a pointer to a language reference manual;

(2) Input/Output Interfaces: a specification of the interfaces that the software must use in order to communicate with the outside world;

(3) Specification of Output Values: for each output, a specification of its value in terms of the state and history of the system's environment;

(4) Timing Constraints: for each output, how often, or how quickly, the software is required to recompute it;

(5) Accuracy Constraints: for each output, how accurate it is required to be.

(6) Likely Changes: if the system is required to be easy to change, the requirements should contain a definition of the areas that are considered likely to change. You cannot design a system so that everything is equally easy to change. Programmers should not have to decide which changes are most likely.

(7) Undesired Event Handling: the requirements should also contain a discussion of what the system should do when, because of undesired events, it cannot fulfil its full requirements. Most requirements documents ignore those situations; they leave the decision about what to do in the event of partial failures to the programmer.

It is clear that good software cannot be written unless the above information is available. An example of a complete document produced in this way is given in [9] and discussed in [8].

B. Design and document the module structure

Unless the product is small enough to be produced by a single programmer, one must give thought to how the work will be divided into work assignments, which we call modules. The document that should be produced at this stage is called a module guide. It defines the responsibilities of each of the modules by stating the design decisions that will be encapsulated by that module. A module may consist of submodules, or it may be considered to be a single work assignment. If a module contains submodules, a guide to its substructure is provided.

A module guide is needed to avoid duplication, to avoid gaps, to achieve separation of concerns, and most of all, to help an ignorant maintainer to find out which modules are affected by a problem report or change request. If it is kept up-to-date, this document, which records our initial design decisions, will be useful as long as the software is used.

If one diligently applies “information hiding” or “separation of concerns” to a large system, one is certain to end up with a great many modules. A guide that was simply a list of those modules, with no other structure, would help only those who are already familiar with the system. The module guide should have a tree structure, dividing the system into a small number of modules and treating each such module in the same way until all of the modules are quite small. For a complete example of such a document, see [3]. For a discussion of this approach and its benefits, see [15,6].

C. Design and document the module interfaces

Efficient and rapid production of software requires that the programmers be able to work independently. The module guide defines responsibilities but it does not provide enough information to permit independent implementation. A module interface specification must be written for each module. It must be formal and provide a black box picture of each module. Written by a senior designer, it is reviewed by both the future implementors and the programmers who will use the module. An interface specification for a module contains just enough information for the programmer of another module to use its facilities, and no more. The same information is needed by the implementor.

While there will be one person or small team responsible for each specification, the specifications are actually produced by a process of negotiation between implementors, those who will be required to use it, and others interested in the design, e.g., reviewers. The specifications include:

(1) A list of programs to be made invokable by the programs of other modules, (called "access programs").

(2) The parameters for those access programs.

(3) The externally-visible effects of these access programs.

(4) Timing constraints and accuracy constraints, where necessary.

(5) Definition of Undesired Events.

In many ways this module specification is analogous to the requirements document. However, the notation and organisation used is more appropriate for the software-to-software interfaces that is the format that we use for the requirements.

Published examples and explanations include [11], [2], [1], [5].

D. Design and document the uses hierarchy

The "uses" hierarchy [13] can be designed once we know all of the modules and their access programs. It is conveniently documented as a binary matrix where the entry in position (A,B) is true if and only if the correctness of program A depends on the presence in the system of a correct program B. The "uses" hierarchy defines the set of subsets that can be obtained by deleting whole programs and without rewriting any programs. It is important for staged deliveries, fail soft systems, and the development of program families [12]. The "uses" hierarchy is determined by the software designers but must allow the subsets specified in the requirements document.

E. Design and document the module internal structures

Once a module interface has been specified, its implementation can be carried out as an independent task except for reviews. However, before coding the major design decisions are recorded in a document called the module design document [16]. This document is designed to allow an efficient review of the design before the coding begins and to explain the intent behind the code to a future maintenance programmer.

In some cases, the module is divided into submodules and the design document is another module guide, in which case the design process for that module resumes at step B above. Otherwise, the internal data structures are described; in some cases, these data structures are implemented (and hidden) by submodules. For each of the access programs, a function [10] or LD-relation [14] describes its effect on the data structure. For each value returned by the module to its caller, another mathematical function, the abstraction function, is provided. This function maps the values of the data structure into the values that are returned. For each of the undesired events, we describe how we check for it. Finally, there is a "verification", an argument that programs with these properties would satisfy the module specification.

The decomposition into and design of submodules is continued until each work assignment is small enough that we could afford to discard it and begin again if the programmer assigned to do it left the project.

Each module may consist of one or more processes. The process structure of the system is distributed among the individual modules.

When one is unable to code in a readable high-level language, e.g., if no compiler is available, pseudo-code must be part of the documentation. It is useful to have the pseudo code written by someone other than the final coder, and to make both programmers responsible for keeping the two versions of the program consistent [7].

F. Write Programs

After all of the design and documentation has been carried out, one is finally ready to write actual executable code. Because of the preparatory work, this goes quickly and smoothly. The code should not include comments that are redundant with the documentation that has already been written. It is unnecessary and makes maintenance of the system more expensive. Redundant comments increase the likelihood that the code will not be consistent with the documentation.

G. Maintain

Maintenance is just redesign and redevelopment. The policies recommended here for design must be continued after delivery or the "fake" rationality will disappear. If a change is made, all documentation that is invalidated must be changed. If a change invalidates a design document, it and all subsequent design documents must be faked to look as if the change had been the original design. If two or more versions are being maintained, the system should be redesigned so that the differences are confined to small modules. The short term costs of this may appear high, but the long term savings can be much higher.

VI. WHAT IS THE ROLE OF DOCUMENTATION IN THIS PROCESS?

A. What is wrong with most documentation today? Why is it hard to use? Why isn't it read?

It should be clear that documentation plays a major role in the design process that we are describing. Most programmers regard documentation as a necessary evil, written as an afterthought only because some bureaucrat requires it. They don't expect it to be useful.

This is a self-fulfilling prophecy; documentation that has not been used before it is published, documentation that is not important to its author, will always be poor documentation.

Most of that documentation is incomplete and inaccurate but those are not the main problems. If those were the main problems, the documents could be easily corrected by adding or correcting information. In fact, there are underlying organisational problems that lead to incompleteness and incorrectness and those problems, which are listed below, are not easily repaired:

(1) Poor organisation. Most documentation today can be characterised as "stream of consciousness," and "stream of execution." "Stream of consciousness" writing puts information at the point in the text that the author was writing when the thought occurred to him. "Stream of execution" writing describes the system in the order that things will happen when it runs. The problem with both of these documentation styles is that subsequent readers cannot find the information that they seek. It will therefore not be easy to determine that facts are missing, or to correct them when they are wrong. It will not be easy to find all the parts of the document that should be changed when the software is changed. The documentation will be expensive to maintain and, in most cases, will not be maintained.

(2) Boring prose. Lots of words are used to say what could be said by a single programming language statement, a formula or a diagram. Certain facts are repeated in many different sections. This increases the cost of the documentation and its maintenance. More importantly, it leads to inattentive reading and undiscovered errors.

(3) Confusing and inconsistent terminology. Any complex system requires the invention and definition of new terminology. Without it the documentation would be far too long. However, the writers of software documentation often fail to provide precise definitions for the terms that they use. As a result, there are many terms used for the same concept and many similar but distinct concepts described by the same term.

(4) Myopia. Documentation that is written when the project is nearing completion is written by people who have lived with the system for so long that they take the major decisions for granted. They document the small details that they think they will forget. Unfortunately, the result is a document useful to people who know the system well but impenetrable for newcomers.

B. How can one avoid these problems?

Documentation in the ideal design process meets the needs of the initial developers as well as the needs of the programmers who come later. Each of the documents mentioned above records requirements or design decisions and is used as a reference document for the rest of the design. However, they also provide the information that the maintainers will need. Because the documents are used as reference manuals throughout the building of the software, they will be mature and ready for use in the later work. The documentation in this design process is not an afterthought; it is viewed as one of the primary products of the project. Some systematic checks can be applied to increase completeness and consistency.

One of the major advantages of this approach to documentation is the amelioration of the Mythical Man Month effect [4]. When new programmers join the project they do not have to depend completely on the old staff for their information. They will have an up-to-date and rational set of documents available.

"Stream of consciousness" and "stream of execution" documentation is avoided by designing the structure of each document. Each document is designed by stating the questions that it must answer and refining the questions until each defines the content of an individual section. There must be one, and only one, place for every fact that will be in the document. The questions are answered, i.e. the document is written, only after the structure of a document has been defined When there are several documents of a certain kind, a standard organisation is written for those documents [5]. Every document is designed in accordance with the same principle that guides our software design: separation of concerns. Each aspect of the system is described in exactly one section and nothing else is described in that section. When documents are reviewed, they are reviewed for adherence to the documentation rules as well as for accuracy.

The resulting documentation is not easy or relaxing reading, but it is not boring. It makes use of tables, formulae, and other formal notation to increase the density of information. The organisational rules prevent the duplication of information. The result is documentation that must be read very attentively but rewards its reader with detailed and precise information.

To avoid the confusing and inconsistent terminology that pervades conventional documentation, a system of special brackets and typed dictionaries is used. Each of the many terms that we must define is enclosed in a pair of bracketing symbols that reveals its type. There is a separate dictionary for each such type. Although beginning readers find the presence of !+terms+!, %terms%, #terms#, etc., disturbing, regular users of the documentation find that the type information implicit in the brackets makes the documents easier to read. The use of dictionaries that are structured by types makes it less likely that we will define two terms for the same concept or give two meanings to the same term. The special bracketing symbols make it easy to institute mechanical checks for terms that have been introduced but not defined or defined but never used.

VII. FAKING THE IDEAL PROCESS

The preceding describes the ideal process that we would like to follow and the documentation that would be produced during that process. The process is "faked" by producing the documents that we would have produced if we had done things the ideal way. One attempts to produce the documents in the order that we have described. If a piece of information is unavailable, that fact is noted in the part of the document where the information should go and the design proceeds as if that information were expected to change. If errors are found, they must be corrected and the consequent changes in subsequent documents must be made. The documentation is our medium of design and no design decisions are considered to be made until their incorporation into the documents. No matter how often we stumble on our way, the final documentation will be rational and accurate.

Even mathematics, the discipline that many of us regard as the most rational of all, follows this procedure. Mathematicians diligently polish their proofs, usually presenting a proof very different from the first one that they discovered. A first proof is often the result of a tortured discovery process. As mathematicians work on

proofs, understanding grows and simplifications are found. Eventually, some mathematician finds a simpler proof that makes the truth of the theorem more apparent. The simpler proofs are published because the readers are interested in the truth of the theorem, not the process of discovering it.

Analogous reasoning applies to software. Those who read the software documentation want to understand the programs, not to relive their discovery. By presenting rationalised documentation we provide what they need.

Our documentation differs from the ideal documentation in one important way. We make a policy of recording all of the design alternatives that we considered and rejected. For each, we explain why it was considered and why it was finally rejected. Months, weeks, or even hours later, when we wonder why we did what we did, we can find out. Years from now, the maintainer will have many of the same questions and will find his answers in our documents.

An illustration that this process pays off is provided by a software requirements document written some years ago as part of a demonstration of the ideal process [9]. Usually, a requirements document is produced before coding starts and is never used again. However, that has not been the case for [9]. The currently operational version of the software, which satisfies the requirements document, is still undergoing revision. The organisation that has to test the software uses our document extensively to choose the tests that they do. When new changes are needed, the requirements document is used in describing what must be changed and what cannot be changed. Here we see that a document produced at the start of the ideal process is still in use many years after the software went into service. The clear message is that, if documentation is produced with care, it will be useful for a long time. Conversely, if it is going to be extensively used, it is worth doing right.

VIII. CONCLUSION

It is very hard to be a rational designer; even faking that process is quite difficult. However, the result is a product that can be understood, maintained, and reused. If the project is worth doing, the methods described here are worth using.

IX. ACKNOWLEDGEMENTS

Stuart Faulk, John Shore, David Weiss, and Stan Wilson of the Naval Research Laboratory provided thoughtful reviews of this paper. Pamela Zave and anonymous referees provided some helpful comments.

Funding for this work has been supplied by the U.S. Navy and by the National Science and Engineering Research Council (NSERC) of Canada.

REFERENCES

1. Parnas, D.L., Weiss, D.M., Clements, P.C., Britton, K.H. Interface Specifications for the SCR (A-7E) Extended Computer Module; NRL Memorandum Report 5502, 31 December 1984. (Major revisions to NRL Report 4843.)

2. Britton, K.H., Parker, R.A. and Parnas, D.L. "A Procedure for Designing Abstract Interfaces for Device-Interface Modules", Proceedings of the Fifth International Conference on Software Engineering, l98l.

3. Britton, K.H. and Parnas, D.L. A-7E Software Module Guide, NRL Memorandum Report 4702, December l981.

4. Brooks, F.P. Jr. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Publishing Company, l975.

5. Clements, P., Parker, A., Parnas, D.L., Shore, J. and Britton, K. A Standard Organization for Specifying Abstract Interfaces, NRL Report 8815, 14 June l984.

6. Clements, P., Parnas, D. and Weiss, D. "Enhancing Reusability with Information Hiding", Proceedings of a Workshop on Reusability in Programming, pp. 240-247, Sept. l983

7. Elovitz, Honey S. "An Experiment in Software Engineering: The Architecture Research Facility as a Case Study", Proceedings of the Fourth International Conference on Software Engineering, Sept. l979.

8. Heninger, K.L. "Specifying Software Requirements for Complex Systems: New Techniques and their Application", IEEE Transactions on Software Engineering, vol. SE-6, pp. 2-13, Jan. l980.

9. Heninger, K., Kallander, J., Parnas, D.L. and Shore, J. Software Requirements for the A-7E Aircraft, NRL Memorandum Report 3876, 27 November, l978.

10. Linger, R.C., Mills, H.D., Witt, B.I. Structure Programming: Theory and Practice, Addison-Wesley Publishing Company, l979.

11. Parker, A., Heninger, K., Parnas, D. and Shore, J. Abstract Interface Specifications for the A-7E Device Interface Module, NRL Memorandum Report 4385, 20 November, l980.

12. Parnas, D.L. "On the Design and Development of Program Families", IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March, l976.

13. Parnas, D.L. "Designing Software for Ease of Extension and Contraction", Proceedings of the Third International Conference on Software Engineering, pp. 264-277, 10-12 May, l978.

14. Parnas, D.L. An Alternative Control Structure and its Formal Definition, Technical Report FSD-81-0012, Federal Systems Division, IBM Corporation, Bethesda, MD, l981.

15. Parnas, D.L., Clements, P. and Weiss, D. "The Modular Structure of Complex Systems", Proceedings of the Seventh International Conference on Software Engineering pp. 408-417, March l984.

16. Faulk, S., Labaw, B., Parnas, D. "SCR Module Implementation Document Guidelines", NRL Technical Memorandum 7590-072:SF:BL:DP, 1 April 1983.

17. Parnas, D.L. and Weiss, D.M. "Active Design Reviews: Principles and Practices", Proceedings of the 8th International Conference on Software Engineering, London, August 1985.