A RATIONAL DESIGN PROCESS: HOW AND WHY TO FAKE IT

David L. Parnas
Department of Computer Science
University of Victoria, Victoria BC V8W2Y2 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

ABSTRACT

Software Engineers have been searching for the ideal software development process: a process in which programs are derived from specifications in the same way that lemmas and theorems are derived from axioms in published proofs. After explaining why we can never achieve it, this paper describes such a process. The process is described in terms of a sequence of documents that should be produced on the way to producing the software. We show that such documents can serve several purposes. They provide a basis for preliminary design review, serve as reference material during the coding, and guide the maintenance programmer in his work. We discuss how these documents can be constructed using the same principles that should guide the software design. The resulting documentation is worth much more than the "afterthought" documentation that is usually produced. If we take the care to keep all of the documents up-to-date, we can create the appearance of a fully rational design process.

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 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 often appear to make decisions without having reasons. They start without a clear statement of what they are going to build. They make a long sequence of design decisions with no clear statement of why they do things the way they do. Their goals are never defined; 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 methodology, 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 classified as "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. The further good news is that it pays to do so.

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

We will never see a software project that proceeds as suggested above. 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 what they do know.

2. Even if we were to know 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.

3. Even if we were to know all of the relevant facts before we start, experience shows that human beings are unable to fully comprehend 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 get to that point, 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.

5. Human errors can only be avoided if one can avoid the use of humans. No matter how rational our decision process, no matter how well we have collected and organised the relevant facts, we will make errors.

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; they may arise spontaneously from other sources.

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. We believe that 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 we have 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 methodology, 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 precisely, we can still write the documentation that we would have produced if we had followed the ideal process. Someone reading the documentation would have the benefit of following a rational explanation of the design. This is what we mean by "faking a rational design process".

Below we list 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 the ideal process, and to a rational design, if we try to follow the process than if we 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

would call 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 an ideal process, it will be easier to review.

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

We believe that the most useful form of a process description will be in terms of work products. For each stage of the process, we describe:

- what we should work on next;
- what criteria that work product must satisfy;
- what kind of persons should do the work;
- 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?

In this section, we describe the rational, idealised software design process that we 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 we ignore either of those. In this paper we are describing an ideal process; testing and review belong in the real process, not the ideal. When we apply the process described in this paper, we include extensive and systematic

reviews of each work product and testing of the executable code that is produced.

A. Establish and document requirements.

If we are to be rational designers we must begin knowing what we must do to succeed. We record that in a work product known as a requirements document. Completion of this document before we start allows us to design with all the requirements in front of us.

1. Why do we need a requirements document?
   - We will be less likely to make requirements decisions accidentally while designing the program.
   - We will avoid duplication and inconsistency. Without this 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.
   - Programmers working on a system are very often not familiar with the application area. Having a complete reference on externally-visible behaviour relieves them of any need to decide what is best for the user.
   - It is necessary (but not sufficient) for making good estimates of the amount of work and money that it will take to build the system.
   - It 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.
   - It provides a good basis for test plan development. Without it, we do not know what to test for.
   - It can be used long after the system is in place to define the constraints for future changes.

- It can be used to settle arguments; we no longer need to be, or consult, application experts.

Determining the detailed requirements may well be the most difficult part of this process because there are usually no well-organised sources of information. Ideally, it would be produced by representatives of the future users. In fact, it is probably going to be produced by software designers who must get it approved by the users' representatives.

2. What goes into the requirements document?

The definition of the contents of the requirements document, in the idealised design process, is simple: it should contain everything you need to know to write correct software, and no more. Of course, we may use references to existing information, if that information is accurate and well organised. The general rules for an ideal requirements document include:

- every statement should be valid for all acceptable products; none should depend on implementation decisions.
- the document should be complete in the sense that if a product satisfies every statement, it should be acceptable.
- where information is not available before development must begin, the areas of incompleteness are indicated, not simply omitted.
- the product is organised as a reference document rather than an introductory narrative about the system, because this is the most useful form. Although it takes considerable effort to produce such a document and is more difficult to read than an introduction, it saves labour in the long run because the information that is obtained in this stage is recorded in a form that allows for easy reference throughout the project.

We obtain completeness in our requirements document by using separation of concerns to obtain the following sections:

- a specification of the machine on which the software must run. The machine need not be hardware -- for some systems this section might simply be a pointer to a language reference manual;
- a specification of the interfaces that the software must use in order to communicate with the outside world;
- for each output, a specification of its value at all times in terms of the software-detectable state of the system;
- for each output, how often or how fast the software is required to recompute it;
- for each output, how accurate it is required to be.
- if the system is required to be easy to change, the requirements must 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, and programmers should not have to decide which things are most likely to be altered.
- the requirements must 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 discuss what will happen when everything works perfectly but leave to the programmer the decision about what to do in the event of partial failures.

We hope it is clear that correct software cannot be written unless each of those requirements is defined, and that once you have succeeded in specifying each of those things, you have completely specified the requirements for your system.

To assure a consistent and complete document, there must be a simple mathematical model behind the organisation. Our model is motivated by our work on real-time systems but because of that it is completely general. All systems are real-time systems.

We assume that for real-time control systems the ideal product is not a pure digital computer, but a hybrid computer consisting of a digital computer that controls an analogue computer. The analogue computer transforms continuous values measured by the inputs into continuous outputs. The digital computer brings about discrete changes in the function computed by the analogue computer when discrete events occur. The actual system is a digital approximation to this hybrid system. As in other areas of engineering, we write our specification by first describing this "ideal" system and then specifying the allowable tolerances. In our requirements document we treat outputs as more important than inputs. If we get the value of the outputs correct, nobody will mind if we do not even read the inputs. Thus, the key to the first stage in the process is identifying all of the outputs. The heart of our requirements document is a set of mathematical functions in tabular form. Each function specifies the value of a single output as a function of external state variables that are relevant to the application. 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 now 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 con-

sist of submodules, or it may be considered to be a single work assignment.

We need a module guide 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 he must work on when he has a problem report. Again, we see that the document that records our design decisions is the same one that will be used during the maintenance phase.

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. Our module guide has 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 of the modules. Precise interfaces must be specified for each module. A Module Interface Specification is written for each module; it must be formal and provide a black box picture of each module. They are written by senior designers and reviewed by potential implementors together with the programmers who will use those interfaces. An interface specification for a module contains just enough information for the programmer of another module to use its facilities, and no more. This is also the information needed by

the implementor. The document we produce is used by both.

While there will be one person responsible for each such document, they are actually produced by a process of negotiation between those who are expected to implement the module, those who will be required to use it, and others interested in the design, e.g., reviewers. The main content of these specifications consists of:

- a list of programs to be made invokable by the programs of other modules, (called "access programs");
- the parameters for those access programs;
- the effects of these access programs on each other;
- timing constraints and accuracy constraints, where necessary;
- definition of Undesired Events (forbidden happenings).

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 with which we are concerned at this stage in the process.

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

D. 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 we begin coding we want to record the major design decisions in a document that we call the module design document. 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 simply 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. In other cases, we

begin by describing the internal data structures; in some cases these data structures are implemented (and hidden) by submodules. For each of the access programs, we include a function [10] or LD-relation [14] that describes its effect on the data structure. For each value returned by the module to its caller, we provide another mathematical function, known as the abstraction function, which 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, we provide a "verification", an argument that programs with these properties would satisfy the module specification.

We continue the decomposition into, and design of, submodules 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.

If we are unable to code in a readable high level language, e.g. if no compiler is available, we include pseudo-code as part of the documentation. We have found it 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].

E. 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].

F. Write Programs

After all of this design and documentation has been carried out, we are ready to write actual executable code. We find that this goes quickly and smoothly. We believe that 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 while increasing the likelihood that the code will not be consistent with the documentation.

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

A. What is wrong with the current documentation? 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 that is done as an afterthought, and done only because some bureaucrat requires it. We believe that documentation that has not been used before it is published will always be poor documentation.

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

- 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 people other than the authors 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.

- boring prose. We find lots of words to say what could be said by a single programming language statement, a formula or a diagram. We find certain facts repeated in many different sections. This increases the cost of the documentation and its maintenance and leads to inattentive reading and undiscovered errors.
- 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, the terms are not used consistently. Careful readings reveal that there are many terms used for the same concept and many similar but distinct concepts described by the same term.
- incompleteness. 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 to newcomers. There are always newcomers on large software projects.

B. How to avoid these problems?

Documentation in the ideal design process meets the needs of the developers and the needs of the maintenance programmers who come later. Each of the documents mentioned above records 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. They will always be up to date. The documentation in our design process is not an afterthought; it is viewed as one of the major products of the project. There are checks that 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 rely on the old staff for their information. They will have an up-to-date and rational set of documents available.

We avoid "stream of consciousness" and "stream of execution" documentation by spending a great deal of effort designing the structure of each document. We define the document by stating the questions that it must answer; we carry that discipline down to individual sections. We try to have a place for every fact that must be contained, and make sure that there is only one such place. Only after we have determined the structure of a document do we begin to write it. If we write many documents of a certain kind, we write and publish a standard organisation for those documents [5]. All of our documents are designed in accordance with the same principle that guides our software design, separation of concerns. Each aspect of the system is described in one section and nothing else is described in that section. When our documents are reviewed, we review them for

adherence to the documentation rules as well as accuracy.

The resulting documentation is not easy or relaxing reading, but it is not boring. We make use of tables, formulae and formal notation to increase the density of information. Our 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 we use a system of special brackets and typed dictionaries. Each of the many terms that we must define, is enclosed in a pair of bracketing symbols that reveals its type. For each such type we have a dictionary that contains only definitions of that type. Although beginning readers find the presence of !+terms+!, %terms%, #terms#, etc., disturbing, regular users of our 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. NOW, HOW DO WE FAKE 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. We fake the process by producing the documents that we would have produced if we had done things the ideal way. We attempt to produce the documents in the order that we have described. If we cannot get a piece of information, we note that in the part of the document where the information should go and proceed to design as if that

information were expected to change. If we find errors we change them and make the consequential changes in subsequent documents. We make the documentation our medium of design and no design decisions are considered to be made until their incorporation into the documents has been approved at all levels. No matter how often we stumble on our way, the final documentation will be easier to understand and accurate. We do not show the way things actually happened, we show the way we wish they had happened and the way things are.

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.

We believe that 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 idealised documentation in one important way. We make a policy of recording all of the alternatives that we considered and rejected, including decisions that were recorded in the earlier versions of a document. 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 go back and find out why. Twenty 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 that we wrote some years ago as part of a demonstration of the ideal process [9]. Normally, one assumes that a requirements document is produced before coding starts and is never used again. However, that has not proven to be the case. The original version of the software, which satisfies our requirements document, is still undergoing revision. The organisation that has to test the software after each change 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. The first document produced in the process is being used many years after the software went into service. The clear message is that if the 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.

It is very hard to be a rational designer and we will probably never achieve it. In our attempts to follow this process, we have often found places where we inherit a design decision that was made for unknown reasons. An example is the value of a constant in an equation that we would like to use. When we ask for a derivation of the constant, we find that there is none or that the derivation is not valid. When we press further, we are told that the decision was made "because it works". In such situations the designer can either open a research project to find out why it works or simply "Get On With It". Those who are paying for our work have made "GOWI" a standard response to many such problems, and we do not expect that real work will ever be different. However, wherever we have made decisions "because they work", we will record the honest reason for our decision rather than mislead the future maintainers into thinking that we had a deep and philosophic reason for what we did.

VIII. ACKNOWLEDGEMENTS

Stuart Faulk and John Shore of NRL provided thoughtful reviews of this paper.

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

REFERENCES

1. Britton, K.H., Clements, P., Parnas, D.L., Weiss, D. Interface Specifications for the A-7E (SCR) Extended Computer Module; NRL Memorandum Report 4843, Jan. 1983.

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, 1981.

3. Britton, K.H. and Parnas, D.L. A-7E Software Module Guide, NRL Memorandum Report 4702, December 1981.

4. Brooks, F.P. Jr. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley Publishing Company, 1975.

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 1984.

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. 1983

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. 1979.

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. 1980.

9. Heninger, K., Kallander, J., Parnas, D.L. and Shore, J. Software Requirements for the A-7E Aircraft, NRL Memorandum Report 3876, 27 November, 1978.

10. Linger, R.C., Mills, H.D., Witt, B.I. Structure Programming: Theory and Practice, Addison-Wesley Publishing Company, 1979.

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, 1980.

12. Parnas, D.L. "On the Design and Development of Program Families", IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March, 1976.

13. Parnas, D.L. "Designing Software for Extension and Contraction", Proceedings of the Third International Conference on Software Engineering, pp. 264-277, 10-12 May, 1978.

14. Parnas, D.L. An Alternative Control Structure and its Formal Definition, Technical Report FSD-81-0012, Federal Systems Division, IBM Corporation, Bethesda, MD, 1981.

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 1984.