UC Irvine
ICS Technical Reports

Title
Reliable software through rational design

Permalink
https://escholarship.org/uc/item/6hh151df

Author
Freeman, Peter

Publication Date
1974

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no.55
c.2

RELIABLE SOFTWARE THROUGH RATIONAL DESIGN

Peter Freeman

Technical Report #55

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

RELIABLE SOFTWARE THROUGH RATIONAL DESIGN*

by

Peter Freeman

ABSTRACT

This paper describes two (related) ways that software unreliability may occur: in response to unanticipated demands or due to unreliable design processes. Five illustrative examples of design-induced unreliability are presented. Design rationalization, a technique for forcing careful and rational consideration of design decisions, is described and its use to improve the reliability of a design process is illustrated. Some experimental and abstract evidence supporting the use of design rationalization to increase software reliability is given.

ICS Technical Report #55
*Supported by National Science Foundation grant GJ-36414

Note: This is a preprint of a paper submitted for publication

Peter Freeman

RELIABLE SOFTWARE THROUGH RATIONAL DESIGN

We are concerned with the design methods used to create software and their effect on various properties of the resulting artifacts. In this paper we describe two (related) ways that software unreliability may occur: in response to unanticipated demands or due to unreliable design processes. We then present five illustrative examples of design-induced unreliability.

Design rationalization is a technique for forcing careful and rational consideration of design decisions. We describe it briefly and illustrate how it can improve the reliability of a design process. We conclude by describing some experimental and abstract evidence supporting the use of design rationalization to increase software reliability.

TWO VIEWS OF SOFTWARE RELIABILITY

Two broad aspects of software reliability are illustrated by the following figures:

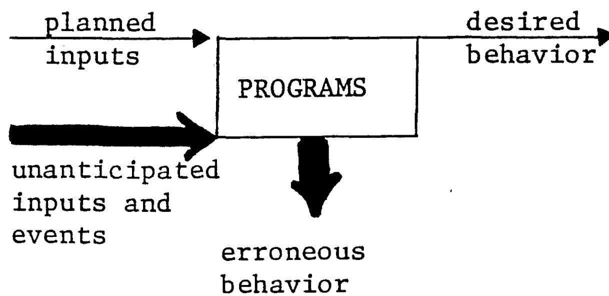

Figure 1: Reliability Under Unanticipated Demands

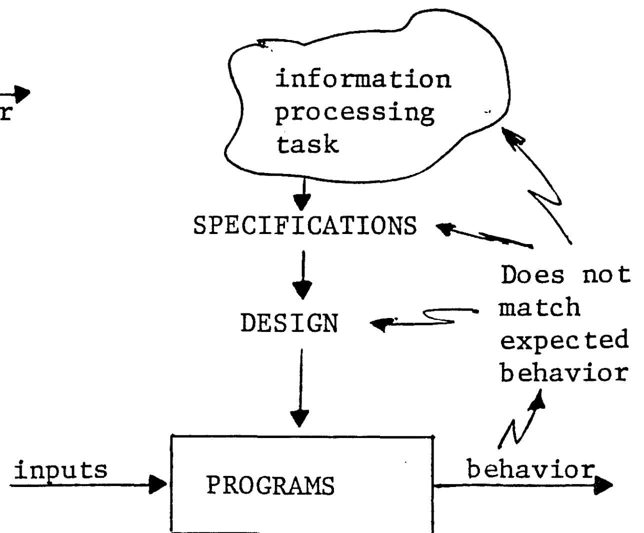

Figure 2: Reliability With Respect to Original Goals

The basic question in Figure 1 is, "How do we make a piece of software resilient to demands that were not anticipated in the original design?". These demands may be erroneous inputs, changed hardware characteristics (e.g. timings), the presence of other systems and/or data in the operating environment, hardware failure, and so on. Work on software structures that provide reliability in the presence of unexpected demands will provide us increased reliability (for example, see [15]).

The basic question in Figure 2 is "How do we insure that a piece of software accurately embodies the operational goals of the original task area?". In other words, from this viewpoint, reliable software must be the product of reliable design.

In practice, both these concepts of reliability are important. Whereas they must remain intertwined and interdependent, it is useful to emphasize their differences when considering how to improve their use. In the first case, we are concerned with software structures - data organizations, control structures, protection techniques, and so on. In the second case, we are concerned with the processes and information used in system design.

Our interest here is to illustrate design-induced unreliability and to propose a way of reducing that source of reliability problems.

EXAMPLES OF DESIGN-INDUCED UNRELIABILITY

The following examples illustrate how unreliable software may be the result of unreliable design processes.

Example 1

A program is specified that takes text files and produces an output file with a justified right margin. The input file may include any ASCII character and the output file may be sent to a variety of output devices (line printer, teletype, video display).

The program works fine until an input file containing ASCII control-characters is processed and the justified file is sent to a line printer. It is then noted that lines containing certain control-characters are not right justified. Although readily explainable by a systems programmer, this is considered unreliable behavior by the user.

The problem is caused by the fact that the designer used the same output routine for all three output devices. The operating system transliterates control-characters being sent to the line printer into 2-character sequences, thus destroying the justification produced by the program.

The constraints and information necessary to take account of this were implicitly present in the specifications, but the design procedure used did not force the designer to take account of them. Thus, they were effectively ignored with the ensuing unreliable behavior the result.

Example 2

A PRINT command is specified for an operating system. The intended effect is to create a line printer listing from a file.

The command works fine most of the time. However, sometimes it deletes the file after printing and other times it doesn't. This seems very unreliable to a group of users who only use the machine occasionally to perform Fortran calculations.

The designer built the program so that unless a special parameter is set, the command will delete the file after printing if its name has a particular form (e.g., a qualifier LST or TMP). A user does not normally think of PRINTing as being an operation that will also delete the file. Indeed, for most files it will not have this effect and since the need for the parameter is not prominently displayed in the documentation, many users do not know of it. Some language processors, such as Fortran, automatically add the LST or TMP qualifier to their output files but not to those of the user program.

The designer was not required to take into account any design goals relating to the user interface,* which resulted in this somewhat arbitrary design. Alternatives, such as warning the user that the file will be deleted and asking for confirmation, did not occur to the designer since he was able to pick the first thing that occurred to him. Since he was also the programmer responsible for maintaining the disks, his (hidden) design goal for the command was to free up space on the disk.

---

* For example, the complex of attributes, called user-centeredness [13], relating to the "friendliness" of a computing environment.

Example 3

A timesharing system is designed to handle 20 interactive terminals. Each is assumed to be performing small calculations in BASIC. A small average response time is desired.

The system works well in practice and gains new users. When more than 22 or 23 users are on-line, response time degrades very rapidly. In effect, the system breaks down (is unreliable).

This unreliable performance is found to be caused by a rigid scheduling algorithm, not a lack of resources. If the designer had tried several alternatives, a scheduler that was more flexible could have been found.

Example 4

A timesharing system which can be accessed remotely via telephone is designed to provide data security through a system of passwords. Because there are several passwords (associated with different files), once a user is logged on (which also requires a password) the system may be queried for the file passwords.

Some time after the system is declared operational, a user has his files "robbed" of important information. The timesharing company is sued over the unreliability of its system.

What happened in fact is this. Substandard telephone connections sometimes cause the connection to be broken between user and system. When this happens, the system does not log off the disconnected user. If someone else dials in on the same phone number that has been disconnected, the system simply connects the new user to the existing job

associated with that port. The new (and bogus) user may then obtain the file passwords of the original user associated with the job.

Although there are several fixes or safeguards to prevent this situation, if the designer had carefully understood the operation of the phone answering module when designing it, the pathological case that permits the system to connect a user to someone else's job might have been discovered.

Example 5

An order-entry system is constructed for on-line usage. It requires the user to enter something for every position on the standard company order form, even if no information is required for this particular order. This proves to be inconvenient since many orders do not use the full form. So, the system is modified to permit the user to skip certain positions by just hitting carriage-return when queried for the information.

It is discovered, however, that sometimes skipping entries later causes erroneous behavior on the order-processing system. The behavior appears to be quite random and the system becomes so unreliable that a manual back-up system is instituted.

After many tests the trouble is finally discovered. When an entry item is skipped, nothing is entered in that position in the data field. Correct operation of the order-processing system, however, requires a standard null entry if there is no information present. When nothing is stored in a position, there may or may not be a null symbol already present, depending on the past history of the file

and the system. Likewise, the behavior of the order-processing system is indeterminant when a null entry is expected but not found.

If the designer making the change had been aware of the requirements of the order-processing system for null symbols and had explored the alternatives for action when a carriage-return was given, then this unreliable behavior might have been avoided.

Reliable Design and Designing for Reliability

Most readers have a set of similar examples which could illustrate our point that the design process may introduce into the system what appears to the user as unreliability. Before describing a partial cure for unreliable design, it is important to stress the difference between reliable design and designing for reliability.

It should be obvious (but judging by much of the software produced, it is not) that everything possible should be done in designing a system to insure the reliability of the system once completed (in whatever terms are appropriate for the case at hand). Robust structures (i.e., ones that will not blow up when presented with erroneous inputs) should be used; safeguards against failure in one area spreading to another should be used; correct operation should be carefully verified. This is usually called designing for reliability.

Our main point in this paper, however, is that even if one designs for reliability the very processes used to arrive at the design may eventually introduce unreliability. Using design techniques that increase the chance that the resulting system will be reliable, independently of the software structures used, is what we term reliable design.

RATIONALIZED DESIGN AND ITS EFFECT ON RELIABILITY

We propose using a technique we call design rationalization to improve the reliability of the design process itself. This, in turn, will improve the reliability of the resultant software.

Rationalized design [1,2] intuitively is straightforward and obvious. Basically, it consists of nothing more than making a design rational -- that is, explainable and based on logical reasoning supported by facts. It proceeds from the assumption that rational design decisions will lead to better designs.

In spite of the fact that no one sets out to design in any other than a rational way, we frequently fall by the wayside at some point. We believe the discipline and structure of a coherent methodology can help significantly in such cases. The methodology that we are developing is aimed at providing this structure and at improving the rationality of designs over and above what they might be without its usage.

The Methodology

A rationalized design is one in which as many of the design decisions as possible are *explicitly* recorded as a choice among feasible alternatives and in which the reasoning that led to the choice of one and rejection of the others is explicitly recorded. Design goals and constraints are also laid out explicitly.

We have been experimenting with two different approaches to the creation of rationalized designs. The first we call *analysis* (or *ex post facto*) *rationalization* and the second we call *synthesis* (or in-process) *rationalization*.

Analysis Rationalization. In this approach we work from a piece of software already designed (and probably implemented). The objective is to reconstruct the design decisions and applicable information logically necessary to obtain the given piece of software. We do not try to recreate the precise decisions (or their sequence) taken by the original designer. In effect, we redesign the object but constrain ourselves to arrive at the same end result. Among other possibilities this technique should be particularly useful in maintenance activities.

We have tried two variations on this basic theme. The first is a top-down approach. We pose a sequence of design problems with alternatives and a rational (justified) choice given for each. If the sequence is followed, it should lead to the piece of software under study. Reference [3] contains several examples of this approach.

The second variation concentrates on particular features found in the software and attempts to provide rationalizations for them. It is more of a bottom-up approach. It appears to be more useful as a tool for critically analyzing a piece of software in a regular manner. Reference [4] contains an example of this technique applied to a small usage accounting program.

Synthesis Rationalization. In-process rationalization applies the same idea of making explicit all the problem-solving aspects of a design process (problem statements, space of alternatives, justifications, decisions) as the design is being done initially. The designer can use whatever design methodology seems appropriate, the only constraint being to justify all decisions and record the information as a choice among alternatives.

Synthesis rationalization in many instances can provide the discipline necessary for reliable design. It assists in finding feasible structures, assessing results of a proposed design decision, and discovering inconsistencies. Thus, it appears to be the more useful approach for the purposes of software reliability.

Producing a rationalized design is difficult (especially in the case of analysis rationalization). It is not easy to identify the design decisions to be rationalized. There are obviously thousands of them ranging from the overall organization of a system to the choice of program variable names. Selecting the most important is hard, but relation of design goals to decisions is one useful way of focussing the rationalization. Likewise, the identification and evaluation of alternatives is difficult, especially since many evaluations should take into account more global considerations. One aspect of our current experimentation is the development of better ways of producing a rationalization.

Some of the tradeoffs of rationalization methodology and a more complete description of it can be found in Reference [2]. A short example is given in the Appendix to provide the flavor of this technique.

The Effect of Rationalization on Reliability

Rationalization techniques applied to the preceding examples would have increased the chance of catching design flaws prior to implementation. In the first four examples, let us assume that either the designer is producing a rationalization as he goes along or that the design he produces is rationalized before

implementation (perhaps at the design review stage). In general, this means that multiple alternatives for each design decision will be explicitly stated and evaluated, design goals and constraints will be stated in operational terms and each decision will be explicitly evaluated in light of them.

In Example 1 the designer would have been required to inspect each design specification to determine its relation to the design of the output module. In evaluating the decision to treat all output the same and let the operating system handle the device dependence, the designer will be forced to consider the alternative of taking care of device dependence in his module. This should be sufficient to trigger recognition of the different handling of control-characters.

Example 2 portrays a case suffering from the misplaced goals of the designer. Had he been required to do a rationalization of the design (or had one been done _ex post facto_) the decision concerning disposition of the file after printing would have been more evident. Had the designer been forced to think through this particular decision, he might have chosen another alternative on his own. At any rate, the decision and its alternatives would have been accessible so that it could have been caught in a design review and/or documented properly.

In Example 3 let us see what would have happened had the designer been forced to seek alternatives. Instead of just choosing a particular scheduling strategy, the designer would have been required to justify this choice and compare it to alternatives. The

search for alternatives might have turned up a more general possibility and the evaluations (if done properly) would have included consideration of the policy's characteristics. In this event, its inflexibility would likely have been evident.

Example 4 involved not seeing the consequences of a decision. If a rationalization had been forced, the logic of the module would have been scrutinized more closely. In particular, the decision to accept input without checking for log-in after the call was answered might have been discovered.

Example 5 could have benefited from rationalization as part of its documentation. If such a rationalization had existed, then it would have been more obvious to the person making the change that one of the functions of the order-entry system was to initialize elements of a file. It then would have been easier to see that the change was bypassing that function.

This is the heart of our argument: Explicit rationalization of a design can reduce the tendency of a design process to introduce unreliability into the systems being designed. Pulling out and making explicit the design decisions that are made, forcing a search for multiple alternatives, and exploring their strengths and weaknesses with respect to the goals and constraints of the design should reduce the chance of design-induced unreliability.

We must emphasize that design rationalization is more than just the obvious use of sound reasoning. One of the premises of design rationalization is that even when the soundest design reasoning is

used, it can be improved by recording it in an accessible format. There are several reasons why this seems to improve the design process. Requiring the designer to seek out alternatives for purposes of comparison may force the discovery of ones that might otherwise be overlooked (this is consistent with psychological work on functional fixity [5]). It provides a record of decisions and rejected alternatives readily available for independent review. Finally, it provides a working record (when done in-process) that the designer can use to keep from losing track of alternatives (an important function, considering the small working memory of the human mind [6]).

Design rationalization is also more than the typical design review that is performed in many multi-person design situations (although a rationalized design should be a great aid to those responsible for formally reviewing the designs of others). While design reviews have the same goal as design rationalization -- providing some assurance that a design is complete and correct -- they typically are too ill-structured and have too little information about design alternatives.

DOES IT WORK?

Evaluation of any methodology, especially one for a complex and expensive task such as software design, is difficult. We do, however, have two types of evidence to present.

Empirical Investigations

In addition to several small designs and design fragments involving synthesis rationalization, we have carried out a major design using a form of in-process rationalization [7]. While it is difficult to measure in any experimentally convincing way, the forced rationalization appears to have led the designer to discover improved solutions to several of the design problems he faced in a new content area.

The continued use of rationalization by several of us whenever we design, as well as its use in some experimental design situations constructed to study other aspects of the design process, are accumulating additional evidence that the regimen of rationalization, while costly in time, pays for itself in the increased quality and reliability of the design.

Finally, a more controlled investigation involving approximately 20 computer science seniors designing various text-handling systems is currently underway. While not an experiment that will prove or disprove the worth of rationalization, it should give us a good deal of valuable data in the same way that other software investigations have shed light on the design process [8, 9, 10].

Theoretical Arguments

While there is no well-developed theory of design to use for the analysis of proposed methodologies, we can abstract enough from what is known informally about design to provide some additional illumination on the role of rationalization. Various models of design have been proposed and are useful for different purposes: functional reasoning [11], stepwise refinement [12], the standard analysis-specification-programming-coding paradigm, formulating assertions and filling in code to satisfy them [15] and others.

When any of these approaches to design are actually used, we generally find behavior involving refinement (iteration), generation of alternatives, and exploration of the effect of alternatives. If we view design as a process carrying us through a space of alternatives till we reach a system satisfying our design goals, then we can portray these three activities as shown in Figure 3.

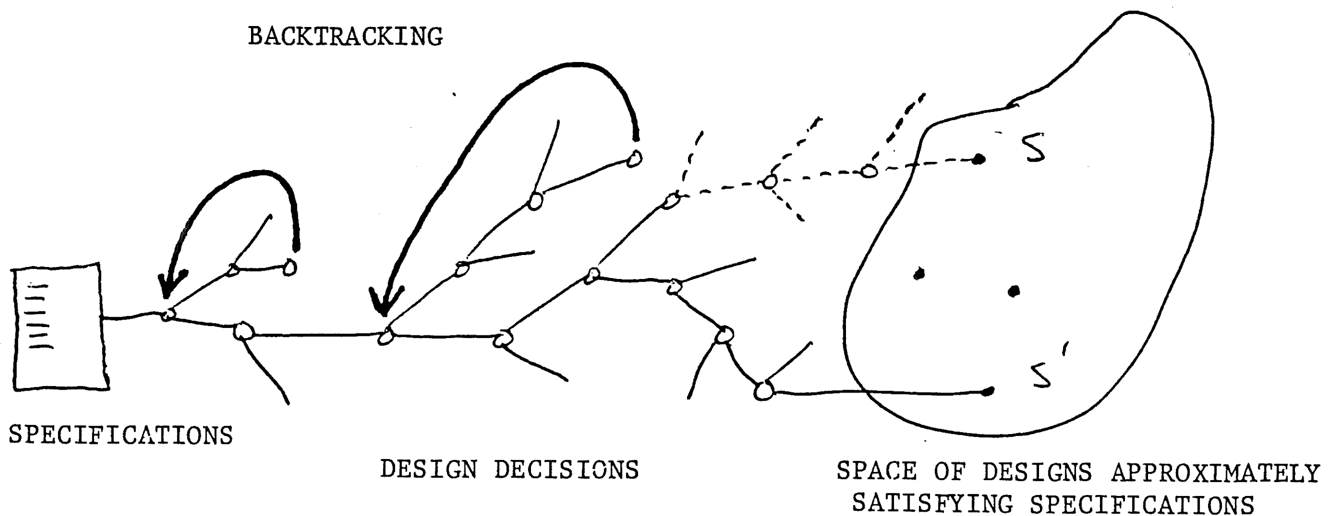

Figure 3: Search for a Reliable Design

When considering reliability, we want our design process to take us to a system S that completely matches our specifications, not a system S' which is similar but which will give unreliable performance because it does not meet the specifications in some (perhaps) subtle ways. When we look at refinement (or iteration) we see we must backtrack; synthesis rationalization, by recording decisions and alternatives, will make it easier to see how far we must go back in order to take an appropriately different path. In the case of alternative generation, the explicit record may provide us with alternatives generated in other parts of the design (but which we might otherwise forget). Likewise, exploration of alternatives is aided by our improved ability to draw on previous evaluations recorded in other parts of the design or in other designs. Further, the explicitness makes it easier to relate specific design goals and constraints to specific alternatives, a necessary operation for evaluation.

CONCLUSION

We have illustrated by examples our thesis that one form of unreliability is due to the unreliability of design processes independent of the content of the design. We have proposed a technique, design rationalization, for improving the reliability of design and hence of the systems produced. Some initial experience with rationalization and some theoretical arguments were presented to support the value of design rationalization as an aid in achieving reliability.

Design rationalization is still largely an experimental technique. Even with work now underway, it will be difficult to "prove" convincingly its value. As with almost any construct or methodology in program creation (e.g., use of goto's, structured programming) counter examples exist and other factors can be found that may partially explain whatever differences in performance are observed.

Nonetheless, we feel that design rationalization shows sufficient promise to warrant further investigation by us and others as a tool to improve design reliability.

ACKNOWLEDGEMENT

The comments of T.A. Standish and Larry Yelowitz are gratefully acknowledged.

REFERENCES

1. Freeman, Peter. "Software Design Rationalization" (abstract). 2nd Computer Science Conference, Detroit, 1974.

2. Freeman, Peter. "Rational Design," in preparation, October, 1974.

3. Ells, T.D. and Peter Freeman. "Design Rationalization of Three BASIC Systems," ICS Tech Report #38, University of California, Irvine, November, 1973.

4. Freeman, Peter, Richard Marino, and Wil Plouffe. "Design Rationalization of Usage Accounting Program," Tech Report, in preparation, October, 1974.

5. Duncker, K. "On Problem-Solving," Psychology Monographs, 58, 5, 1945.

6. Newell, A. and H.A. Simon, Human Problem Solving, Prentice Hall, 1971.

7. Levin, Steven. "The Distributed BASIC Interpreter System," ICS Tech Report #33, University of California, Irvine, June, 1973.

8. Parnas, D.L. "Some Conclusions from an Experiment in Software Engineering Techniques," Proc AFIPS 1972 FJCC, 1972.

9. Naur, Peter. "An Experiment on Program Development," BIT, Vol. 12, pp. 347-365, 1972.

10. Newell, Allen. private communication.

11. Freeman, Peter and Allen Newell. "A Model for Functional Reasoning in Design," Proc 2nd Int. Jt. Conf. on AI, London, 1971.

12. Wirth, Nicklaus. "Program Development by Stepwise Refinement," Comm ACM 14, 4, 1971.

13. Kling, Rob. "User-Centered Design," Proc ACM National Conf., 1973.

14. Randell, B. "Operating Systems: The Problems of Performance and Reliability," Proc IFIPS 71, pp. I100 ff, 1971.

15. Yelowitz, L. "A Symmetric, Top-down Structured Approach to Computer Program/Proof Development," (Bethesda, Maryland: IBM Report, July 1973) FSC 73-5001.

APPENDIX *

NOTE: This example is in rough form, but it will be cleaned up in the next draft. The design rationalizations are boxed.

Design Problem

Design a grading system to keep track of test scores for a class. The system should be capable of printing a class list that includes students names, id numbers, text scores and some summary test statistics (minimum and maximum scores, the mean, median and standard deviation).

Example Design Specification with Rationalization

This is an example of a program specified in an informal design language with embedded design rationalization information. It should provide some idea of what rationalization information looks like and how to incorporate it into a design.

****

Data file specification

The test data file will be stored on disk. Records will be fixed length with the following format:

```
chars  1 - 25    student name
chars 26 - 30    student id number
chars 31 - 33    score for test 1
chars 34 - 36    score for test 2
chars 37 - 39    score for test 3
chars 40 - 42    score for test 4
chars 43 - 45    score for test 5
```

Missing test scores will be recorded as 0.

```
Problem/issue: format of test data file
Alts: 1) fixed length fixed format records
      2) fixed length variable format (free) records
      3) variable length free format records
Choice: 1
Rat: Fixed format is chosen over free format because the amount of
     effort necessary to put the data into the right columns is not
     significant enough to justify the programming effort needed to
     interpret a free format data record. Fixed length is chosen over
     variable length because the storage savings of variable length
     would be insignificant in this application.
```

* Prepared by Steven Levin.

Program specification

```
proc GRADER

    /* top level procedure */

    filename ← GETFILENAME
    numberoftests ← GETNUMBEROFTESTS
```

Problem/issue: how should the test data be represented in-core Alts: 1) some form of linked list 2) an array Choice: 2 Rat: Processing will require tabular access both across by student and down by test. Arrays would provide the easiest accessing for this type of processing.
Problem/issue: how to determine how many students are in the class Alts: 1) have the user input this information 2) have the procedure GETTESTDATA return the information 3) use a special data value in the data array that flags the last entry Choice: 2 Rat: Eliminate alt 1 because the information is available directly from the data. Alt 3 is a poor choice because it would require putting a special test (the same one) into several other procedures.

```
    array,numberofstudents ← GETTESTDATA(filename)
    PRINTSCORES(array,numberoftests,numberofstudents)
    PRINTSTATS(array,numberoftests,numberofstudents)
endproc

proc GETFILENAME

    /* prompts user, gets file name, checks its legality */

    while TRUE do
        (prompt user by printing 'input the filename';
         get the filename;
         if the filename is legal then exit returning the filename
            else print 'not a legal filename')
endproc
```

This design is not complete but it should illustrate the format and content of rationalization data. Even in the section of design given above there are several problems/issues that were encountered but simply left unrecorded. This was done here to simplify the example