The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Working Paper 220 December 1981

The Disciplined Use of Simplifying Assumptions

Charles Rich & Richard C. Waters

Abstract

Simplifying assumptions — everyone uses them but no one's programming tool explicitly supports them. In programming, as in other kinds of engineering design, simplifying assumptions are an important method for dealing with complexity. Given a complex programming problem, expert programmers typically choose simplifying assumptions which, though false, allow them to arrive rapidly at a program which addresses the important features of the problem without being distracted by all of its details. The simplifying assumptions are then incrementally retracted with corresponding modifications to the initial program. This methodology is particularly applicable to rapid prototyping because the main questions of interest can often be answered using only the initial program.

Simplifying assumptions can easily be misused. In order to use them effectively two key issues must be addressed. First, simplifying assumptions should be chosen which simplify the design problem significantly without changing the essential character of the program which needs to be implemented. Second, the designer must keep track of all the assumptions he is making so that he can later retract them in an orderly manner. By explicitly dealing with these issues, a programming assistant system could directly support the use of simplifying assumptions as a disciplined part of the software development process.

Submitted to the ACM SIGSOFT Second Software Engineering Symposium: Workshop on Rapid Prototyping. Columbia, Maryland, April 19-21, 1982.

Introduction

Simplifying assumptions — everyone uses them but no one's programming tool explicitly supports them. In this paper we explore the origin of simplifying assumptions in software engineering and suggest how to explicitly support their use as part of the software development process.

Given a complex programming problem, expert programmers typically choose simplifying assumptions which, though false, allow them to arrive rapidly at a program which addresses the important features of the problem without being distracted by all of its details. The simplifying assumptions are then incrementally retracted with corresponding modifications to the initial program. This methodology is particularly applicable to rapid prototyping because the main questions of interest can often be answered using only the initial program.

For example, when implementing a program to update a file of indexed records, a programmer might first assume that there will be no error conditions opening files and reading records. Only later, after making certain that the basic algorithm he had chosen was correct, would he go back and decide how to handle these various error conditions.

There are, of course, some limitations on the use of simplifying assumptions as a design method. There are certain solutions which one would never arrive at by this method. However, for routine software system creation (as opposed, for example, to the creation of fundamentally new algorithms) the benefits in time and expense far outweigh the cost of missing some possible solutions.

In the next part of this paper, we enlarge our perspective from the use of simplifying assumptions to a theory of how engineers in general, and software engineers in particular, cope with the complexity of engineering design problems. From this perspective we see that the use of simplifying assumptions is not a superficial phenomenon, but rather the manifestation of a powerful problem solving paradigm. We also briefly describe some recent work in artificial intelligence which has begun to give us the formal tools for describing the engineering design process. Finally, we present some specific ideas of how future programming systems should support the disciplined use of simplifying assumptions, including a scenario of a system we are currently developing.

Programming Viewed as an Engineering Activity

We believe that software engineering has much to learn from more mature engineering disciplines, such as electrical engineering, and that the problem solving behaviors of engineers in different disciplines have many similarities [3]. One of the first people to study engineering design from an artificial intelligence point of view was Sussman [9,8]. He observed that programs, like all other designed artifacts, tend to be constructed hierarchically, and that the parts at each level tend to be organized in stereotyped ways. (We call these stereotyped organizations *clichés*.) Sussman also noted that the use of simplifying assumptions played a crucial role in the design of complex devices. (Similar ideas are present in the earlier work of Newell, Shaw and Simon [2] and Sacerdoti [6].)

In summary, three key ideas in current artificial intelligence theories of engineering problem solving are:

(i) *Abstraction* — choosing a simplified view of the problem to guide the problem solving process. This includes both the use of simplifying assumptions as described above, as well as other mechanisms for postponing details, such as data abstraction.

(ii) *Inspection* — problem solving by recognizing the form of a solution. An initial implementation is constructed by selecting and combining appropriate clichés. Abstraction is an essential prerequisite for this step because it makes it possible for the designer to recognize which clichés are appropriate.

(iii) *Debugging* — incremental modification of an almost satisfactory solution to a more satisfactory one. Debugging is required when a simplifying assumption is retracted. It is also needed due to interactions between clichés and in response to changes in the problem specifications.

These three techniques used together form a powerful mental tool used by engineers of all kinds for dealing with complex design problems. For example, consider the following advice found in a textbook on designing integrated circuits [1].

> Probably the worst way to proceed in analyzing the behavior of a small-signal circuit is to use the most complex model available for the device. Not only does such a procedure lead to lengthy computation, but by its complexity it also obscures the really important aspects of the circuit behavior. Instead, one should begin with a highly idealized simple model for the device. One can then focus on important circuit behavior and obtain some idea as to

reasonable approximations; one can usually determine also what features of detailed device behavior will next be important.

The rest of this paper focuses on the single issue of simplifying assumptions. The reader is referred to [5] for a discussion of the other implications of this theory for software engineering.

Appropriate Simplifying Assumptions

The simplifying assumptions which are chosen for a software design problem strongly affect both the ease of implementation and the form of the program which is eventually produced. In general, the appropriateness of a simplifying assumption is relative to the goals of the particular design. A good simplifying assumption simplifies the design problem significantly without changing the essential character of the program which needs to be implemented. This general notion of appropriateness can be further refined by considering two more specific criteria which we call *consistency* and *continuity.*

A consistent simplifying assumption is one that allows the resulting software to work correctly at least some of the time. For example, it would usually be consistent to ignore error conditions such as missing files and records in the initial design phase of a file update program because these conditions are typically not involved in the normal operation of the system. On the other hand, ignoring round-off error in a large numerical processing application might violate this criterion because the results thus obtained could have zero significant digits remaining. Consistency is particularly important in rapid prototyping because it makes it possible to experiment with the broad outlines of the design without having to implement all of its ultimate complexity.

A continuous simplifying assumption is one that allows the resulting software to be incrementally modified to its final desired form. For example, ignoring input-output errors in a file update program as discussed above is also a continuous assumption because it is easy to add code later to handle these conditions. On the other hand, assuming in a distributed data base application that there was no concurrent access to the information would probably violate this criterion because the retraction of this assumption would probably force a major reorganization of the program. (Note however that this assumption is consistent.)

There is a difficulty with the account of appropriateness given above. It implies that in order to choose appropriate simplifying assumptions, one must already know something about the solution. The way out of this apparent paradox is to realize that an important part of a designer's expertise is his knowledge of standard simplifying assumptions and the types of problems to which they are applicable. This knowledge is just as important as knowing the standard implementation techniques in a field.

The Disciplined Use of Simplifying Assumptions

Supporting the disciplined use of simplifying assumptions in software development will require significant extensions to current programming tools. There are two key issues to be addressed: choosing appropriate assumptions and keeping track of assumptions so they can be incrementally retracted. This section describes how a system we are developing, called the *programmer's apprentice* (PA), deals with these issues.

The PA makes crucial use of two important ideas from recent work in artificial intelligence. The first idea is that, in addition to the code for a program, it is important to have an explicit representation of its *plan* [4]. The plan for a program expresses not only its "physical" structure at some level of detail, but also its *teleology*. In general, the teleology of a device assigns a purpose to each of its parts. Note that not all parts of the same type have the same purpose. For example, in the plan for a program the purpose of one particular list APPEND operation might be to compute the union of two sets, while the purpose of another might be to concatenate two sequences.

Using the plan representation, we are compiling a library of programming clichés which includes common data structures (such as linked lists and hash tables), algorithm fragments (such as searching), and system organizations (such as master file systems). Each cliché in the library is annotated with appropriate simplifying assumptions from which the programmer can choose. This supports the use of standard simplifying assumptions.

The second important idea from artificial intelligence is to explicitly represent the *dependencies* between features of plans at different levels of abstraction. Dependencies are similar to the justifications in a logic proof. The justifications in a logic proof record the premises and intermediate results that each step in the proof depends on. Similarly, the dependencies in the plan for a program record the problem features and simplifying assumptions that each feature of the program depends on. Dependencies are the key to the *disciplined* use of simplifying assumptions because they guarantee that these assumptions are not forgotten. Dependencies also make it possible to determine which parts of a program are affected when a simplifying assumption is retracted.

Supporting the choice and retraction of arbitrary simplifying assumptions is much more difficult than supporting standard assumptions because it requires general reasoning facilities that are at or beyond the current state of the art. For example, in a program with complex data structures one might initially want to assume there was no sharing of data structures. When this simplifying assumption is retracted, the system should be able to deduce the possible conflicts which may arise due to side effects [7].

A Scenario

We end this paper with a scenario of how the PA will support the disciplined use of standard simplifying assumptions. In this scenario a programmer is adding a simple user accounts system to an existing operating system. We assume that the existing operating system was originally implemented using the PA, so that the PA already understands much of its structure and function. The user accounts system is keeps track of a balance for each user and prevents him from logging in when his balance is overdrawn. (This scenario is drawn from a longer scenario [5] which also illustrates other features of the PA.)

The scenario is presented as a sequence of screen images. The screen is divided into two parts. In the top part, the programmer types commands (in lower case) and the PA responds (in upper case). The PA uses the lower portion of the screen to display relevant sections of code. The interaction between the programmer and the PA shown in the scenario is carried on in natural English. The demonstration systems we are implementing, however, will use a simple formal command language. Also, although the code shown in the scenario is Lisp, the basic PA concept as well as the actual implementation is substantially programming language independent.

The scenario picks up at the point where the programmer has just completed his first pass at implementing the user accounts subsystem. In the first pass the programmer has been using simplifying assumptions in order to ignore a number of details of lesser importance. For example, in the simple routine USER-OVERDRAWN (shown below as the system would display it during initial design) the programmer has ignored the fact that the user master file might not exist, and that there might not be a record corresponding to the given user ID. The programmer was guided to make these simplifying assumptions by the fact that the suggestion to make them is part of the PA's clichéd understanding of operations on files. (LET* used below is a variation of the Lisp special form LET; LET* binds its variables sequentially instead of in parallel.)

```
(DEFUN USER-OVERDRAWN? (ID)
  (LET* ((FILE (OPEN "<SYS>USER" '(IN KEYED)))
         (RECORD (KEYED-READ FILE ID)))
    (CLOSE FILE)
    (MINUSP (USER-BALANCE RECORD))))
```

It is important to realize that the code displayed by the PA is only the tip of an iceberg. The PA knows many things about this program — in particular that it is only correct under the assumption that the aforementioned exceptional conditions do not occur. As the scenario begins, the programmer decides to go back and deal with some of the details which were suppressed in the initial implementation.

```
> Consider retracting simplifying assumptions.
SIMPLIFYING ASSUMPTIONS:
 (1) NO MISSING RECORDS (KEYED-READ) -- USER-OVERDRAWN?, UPDATE-USER-FILE, LOGOUT
 (2) NO MISSING FILES (OPEN) -- USER-OVERDRAWN?, UPDATE-USER-FILE, LOGOUT
 (3) NO INVALID COMMANDS (COMMAND LOOP) -- UPDATE-USER-FILE, EDIT-USER-RECORD
 (4) NO READ/WRITE CONFLICTS -- UPDATE-USER-FILE, LOGOUT
 ...
                                                  -- SHOW USER-OVERDRAWN? ? yes

(DEFUN USER-OVERDRAWN? (ID)
  (LET ((FILE (OPEN "<SYS>USER" '(IN KEYED))))
    (COND ((NULL FILE) ..missing-file-handler..)
          (T (LET ((RECORD (KEYED-READ FILE ID)))
               (CLOSE FILE)
               (COND ((NULL RECORD) ..missing-record-handler..)
                     (T (MINUSP (USER-BALANCE RECORD)))))))))
```

The PA responds to the programmer's request by listing the simplifying assumptions he has made, and the subroutines which are affected by these assumptions. The programmer then looks at individual subroutines and modifies them in order to eliminate their dependence on simplifying assumptions. Following the PA's suggestion, he looks at USER-OVERDRAWN? first. Now that the programmer is explicitly considering the exceptional conditions which could arise during the execution of the program, the PA's display of the program contains skeletal code fragments (underlined in the figure) which indicate where simplifying assumptions are being used, and how the code could be modified to explicitly deal with them. The first fragment tests whether the OPEN failed and includes an unspecified action to take when it does (when OPEN or KEYED-READ fail, they return NIL). The other underlined fragment tests for the failure of KEYED-READ.

```
> If the record is missing, return t.

(DEFUN USER-OVERDRAWN? (ID)
  (LET ((FILE (OPEN "<SYS>USER" '(IN KEYED))))
    (COND ((NULL FILE) ..missing-file-handler..)
          (T (LET ((RECORD (KEYED-READ FILE ID)))
               (CLOSE FILE)
               (OR (NULL RECORD)
                   (MINUSP (USER-BALANCE RECORD)))))))))
```

The programmer decides that if the record is missing then the user should be considered to be overdrawn. The PA modifies the code and plan appropriately and removes the dependency on the assumption that the record will exist.

```
> Check that the master file exists at system startup time and
  protect it from deletion.
MODIFICATIONS:
  (1) FILE EXISTS CHECK ADDED TO SYS-STARTUP.
  (2) <SYS>USER MARKED AS NON-DELETABLE.
  (3) DEPENDENCE ON THE SIMPLIFYING ASSUMPTION THAT <SYS>USER WILL EXIST
        REMOVED FROM USER-OVERDRAWN?, UPDATE-USER-FILE, AND LOGOUT.
                                               -- SHOW SYS-STARTUP ? no

(DEFUN USER-OVERDRAWN? (ID)
  (LET* ((FILE (OPEN "<SYS>USER" '(IN KEYED)))
        ;file existence checked at startup and protected from deletion.
        (RECORD (KEYED-READ FILE ID)))
    (CLOSE FILE)
    (OR (NULL RECORD)
        (MINUSP (USER-BALANCE RECORD)))))
```

One way to deal with a simplifying assumption is to modify the system as a whole so that it becomes true. Here, the programmer instructs the PA to add code to the operating system to check that the user master file exists at system startup time, and to protect it from subsequent deletion (e.g. by setting a file descriptor bit). The PA deduces that this means that the calls to OPEN on the user master file cannot fail. As a result it removes the dependencies which indicate that the calls to OPEN on the user file are assumed to be successful. Note, however, that the PA installs new dependencies which indicate that OPEN on this file cannot fail because of the global constraints imposed by the programmer. The underlined comment above is the PA's method of displaying this dependency to the programmer. If this dependency is ever violated in the future, the PA can inform the programmer.

```
> Show update-user-file.

(DEFUN UPDATE-USER-FILE ()
  (LET ((FILE (OPEN "<SYS>USER" '(UPDATE KEYED))))
         ;file existence checked at startup and protected from deletion.
    (PROG (COMMAND ID) ;command loop with quit command.
       LP (SETQ COMMAND (SYS-PROMPT "type Edit, Delete, Create, or Quit:"))
          (COND ((EQ COMMAND 'QUIT) (RETURN NIL)))
          (SETQ ID (SYS-PROMPT "User id:"))
          (CASEQ COMMAND
            (EDIT (KEYED-WRITE FILE ID (EDIT-USER-RECORD (KEYED-READ FILE ID))))
            (DELETE (KEYED-DELETE FILE ID))
            (CREATE (KEYED-WRITE FILE ID (EDIT-USER-RECORD
              (MAKE-USER-RECORD 'USER-ID ID 'USER-PASSWORD "" 'USER-BALANCE 0
                                'USER-NAME "" 'USER-ADDRESS ""))))
            (T ..invalid-command-handler..))
          (GO LP))
    (CLOSE FILE)))
```

Standard simplifying assumptions exist not only for primitive operations such as OPEN and KEYED-READ, but also for intermediate level clichés. For example, the *command loop* cliché (used in the program UPDATE-USER-FILE above) has associated with it the idea that the command read may be invalid, but that it is appropriate to ignore this problem during initial design. In response to the programmer's request, the PA displays the program and indicates where code should be added to deal with invalid commands. Below the programmer makes a simple incremental modification to the program in order to deal with the problem. His change closely follows the form suggested by the PA.

```
> If the command is invalid, print "invalid command" and continue.

(DEFUN UPDATE-USER-FILE ()
  (LET ((FILE (OPEN "<SYS>USER" '(UPDATE KEYED))))
         ;file checked at startup and protected from deletion.
    (PROG (COMMAND ID) ;command loop with quit command.
       LP (SETQ COMMAND (SYS-PROMPT "type Edit, Delete, Create, or Quit:"))
          (COND ((EQ COMMAND 'QUIT) (RETURN NIL)))
          (SETQ ID (SYS-PROMPT "User id:"))
          (CASEQ COMMAND
            (EDIT (KEYED-WRITE FILE ID (EDIT-USER-RECORD (KEYED-READ FILE ID))))
            (DELETE (KEYED-DELETE FILE ID))
            (CREATE (KEYED-WRITE FILE ID (EDIT-USER-RECORD
              (MAKE-USER-RECORD 'USER-ID ID 'USER-PASSWORD "" 'USER-BALANCE 0
                                'USER-NAME "" 'USER-ADDRESS ""))))
            (T (PRINT "invalid command")))
          (GO LP))
    (CLOSE FILE)))
```

At this point the programmer goes home for the night, putting off the problem of how to deal with potential read/write conflicts between UPDATE-USER-FILE and LOGOUT for a later time. The PA remembers that this issue still needs to be dealt with and will not let the programmer claim that the system is complete until this issue is resolved.

Conclusion

Our aim in this paper has been to draw attention to a direction of research in software tools which we believe is very important and which has been neglected in the past. While we have not yet solved all of the problems inherent in implementing the scenario above, we do hope to have established three basic points.

First, it should be clear that the use of simplifying assumptions is a necessary and commonly used mental tool in the construction of large and complex programs. We feel that the myth, current in some circles, the approximation and debugging have no place in the programming process has inhibited progress in developing more effective tools for programmers.

Second, it is possible to use simplifying assumptions in a disciplined way without sacrificing program reliability. This is especially true if the paradigm is supported by an intelligent programming environment, such as the programmer's apprentice.

Finally, as in other aspects of the engineering design, there is a large amount of leverage to be obtained by indentifying the clichés which are used. In particular, many program structure clichés, such as master file systems, have associated clichéed simplifying assumptions.

Bibliography

[1] D.J. Hamilton, W.G. Howard, *Basic Integrated Circuit Engineering*, McGraw-Hill, 1975.
[2] A Newell, J.C. Shaw, and H.A. Simon, "Report on a General Problem-Solving Program for a Computer," *Information Processing: Proc. Intl. Conf. Info. Processing*, UNESCO, Paris, 1960, pp. 256-264. (Reprinted in *Computers and Automation*, July 1959.)
[3] C. Rich, H.E. Shrobe, R.C. Waters, G.J. Sussman, and C.E. Hewitt, "Programming Viewed as an Engineering Activity", (NSF Proposal), MIT/AIM-459, January, 1978.
[4] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", *Proc. of 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, Canada, August, 1981.
[5] C. Rich and R.C. Waters, "Abstraction, Inspection and Debugging in Programming", MIT/AIM-634, June, 1981.
[6] E.D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces", *Artificial Intelligence*, Vol. 5, No. 2, 1974, pp. 115-135.
[7] H.E. Shrobe, "Dependency Directed Reasoning for Complex Program Understanding", (Ph.D. Thesis), MIT/AI/TR-503, April, 1979.
[8] G.J. Sussman, "The Virtuous Nature of Bugs", *Proc. of Conf. on Artificial Intelligence and the Simulation of Behavior*, U. of Sussex, July 1974..

[9] G.J. Sussman, "Electrical Design, A Problem for Artificial Intelligence Research", *Proc. of 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Massachusetts, August 1977.