Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 270 May 1985

# Toward a Richer Language for Describing Software Errors

Samuel M. Levitin

## Abstract

Several approaches to the meaning and uses of errors in software development are discussed. An experiment involving a strong type-checking language, CLU, is described, and the results discussed in terms of the state of the art language for bug description. This method of bug description is found to be lacking sufficient detail to model the progress of software through its entire lifetime. A new method of bug description is proposed, which can describe the bug types encountered not only in the current experiment but also in previous experiments. It is expected that this method is robust enough to be independent of the various factors of a software project that influence the realms in which bugs will occur.

# Table of Contents

# Table of Figures

# Table of Tables

# Introduction

Ever since ENIAC, the first electronic computer in the 1940's, computer scientists have had to worry about bugs. Once, this massive vacuum tube-filled computer failed, and the engineer who delved into the physical components of the machine found a moth inside one of the banks of tubes. The moth's presence caused the system to malfunction, and thus spawned the term "bug", or so the story goes. However, the meaning of bug as flaw may even go back to Edison, who spoke of "getting the bugs out" of an invention.[1] [12]

Indeed, for those less-than-perfect architects of computer programs, the debugging phase is a familiar one. Veterans of any amount of programming assignments will remember sayings such as

- *There's always one more bug,*
- *The debugging is 90% complete* said at any phase of testing, and
- *The first 90% of the problem takes the first 90% of the time; it's the last 10% of the problem that takes the other 90% of the time.*

Have you ever thought about what the types of bugs you encounter indicate about your programming style, or programming in general? Bugs have been tracked, removed, and exterminated throughout the computer era, but only recently have they been intently analyzed in order to provide insight into the way engineers think.

Chapter 1 summarizes the viewpoints toward software bugs from the fields of computer systems, software engineering, AI, and cognitive science. Chapter 2 relates why a study of only a subset of the problem space is possible. Chapter 3 discusses the

---

[1] page A94

details of the experiment. Chapter 4 contains the discussion of the data collected in terms of the current methods of bug description. Finally, Chapter 5 suggests a language for discussing software bugs, and contains improvements toward the experimental design. The various appendices contain the forms presented to the participants in the experiment. Appendix IV contains a sample of the code that implements the task of the experiment.

# Chapter One

# Various Philosophies Toward Bugs

As computer science has evolved, so has thought on the significance and purpose of bugs in software. Different approaches toward bugs originate in the fields of software engineering, cognitive science, artificial intelligence, and computer systems analysis. The approaches of the researchers from these disciplines can be closely intertwined. For example, the boundary between the AI field and the cognitive science field is not clearly defined.

## 1.1 A Computer Systems Perspective

Consider a large software project, such as the five-year development of OS/360 by a team of 1000, including engineers, managers, and support personnel. There were many bugs in the software, and much attention was paid to the bug appearance rate and the bug correction rate. Brooks, in [1], shows that the rate of bug detection is large at first, while the "kinks" are being worked out of the software, declines gradually as the software reaches its peak usage, and then increases as the lifetime of the software is reached and surpassed. One reason that the software becomes more "bug-filled" after a period of relatively trouble-free usage is that it is not forward-compatible with the users five years in the future.

The approach of the analysts in the systems environment is to quantify and track bug detection and bug resolution. They do not wish to attempt to punish the wrong-doers, or judge why a certain bug or set of bugs appeared; they wish merely to use the bug rates as metrics for software quality.

## 1.2 Software Engineering Perspectives

### 1.2.1 Dijkstra's view

In a collection of his essays [2], Dijkstra discusses the separation of astronomy from astrology, and likens it to the development of computer science. He writes

> ... the prevalence of anthropomorphic terminology in computing can also be viewed as a characteristic of its pre-scientific stage, and a consequence would be that computing *scientists* don't deserve that name before they have the courage to call a "bug" an "error".[2] (his italics)

### 1.2.2 The Guttag/Liskov View

In a draft of notes for an MIT course in software engineering [3], the following passage appears:

> The word "bug" is in many ways misleading. Bugs do not crawl unbidden into our programs. We put them there. *Don't think of your program as "having bugs;" think of yourself as having made a mistake.* Bugs do not breed in programs. If there are many bugs in a program, it is because the programmer has made many mistakes. You should never be proud when you track down a bug in your own program. It's like finding a cockroach in your kitchen. You should be embarrassed and upset that it was there in the first place. (their italics)

Although it is ludicrous to be embarrassed about bugs in software, there is some substance in the latter view; we will see later that the notion of programmer-introduced bugs is valid. There are certain circumstances under which programmers intentionally introduce errors in order to make progress toward the ultimate goal. Successive approximation is an idea crucial to problem solving. Thus, the Dijkstra notion that all bugs are errors, and especially, as he implies, that errors are faults that we should try to minimize, has little support in the current understanding of the reasons for bug appearance.

---

[2] page 290

## 1.3 AI Perspectives

Artificial Intelligence strives to discover how to program a machine to simulate human intelligence. A natural part of demonstrating human-like intelligence is the ability to learn from prior experience. Significant effort in AI has been made to analyze the method by which machines can be made to learn, particularly in specialized domains.

### 1.3.1 Sussman's HACKER

Sussman's HACKER [11] was a system to perform problem solving in one of these specialized domains, the blocks world. HACKER had a store of "plans", or canned answers, along with a set of rules for their applicability. When faced with a task, HACKER searched for and applied a canned answer whose applicability matched that of the situation at hand. If at some point an error occurred, e.g. because of incomplete knowledge, HACKER analyzed the differences between the situation at hand and the rule for applicability, fine-tuning the criteria for applicability. Faced with a similar situation later, HACKER would apply the correct rule. Consequently, HACKER became increasingly accurate in applying rules.

The classic example Sussman gives is to create a tower of three blocks, A, B, and C, by placing A on B and B on C. The general, unmodified rule is to assume that, in the absence of evidence to the contrary, goals are independent and can be performed in any order. To make the tower, first place A on B. Then to place B on C, establish an intermediate goal of clearing the top of B (due to the strength of the computer-controlled manipulator). These two steps do and then undo an action, namely placing A atop B. This wasted action implies that there is some room for streamlining the process. When faced with situations that necessitate amending the rules of applicability, HACKER creates a critic that summarizes the drawback of the original rule, generating an additional rule or clause. The critic will then inspect future situations for the possibility of the subgoals' interacting. Given a similar situation, with

different block names, HACKER will then succeed. Different errors, however, will need their own chance "in the spotlight" before a critic is created to screen out subsequent errors of that type.

The underlying theme of HACKER is the idea of PSBDARP: Problem Solving By Debugging Almost-Right Plans. That is, start with a plan and assume it to be correct until you find evidence to the contrary. Then, try to find out what element of the problem at hand makes the proposed plan inapplicable. Propose an improved plan to handle this problem that differs slightly from those for which the proposed plan failed. The more HACKER tries to solve problems, the more complete its knowledge of applicability of rules becomes, and the greater the population of critics. Sussman's approach supports the view that there is something desirable to be gained by analyzing the causes of bugs: a potential increase in applicability of rules.

### 1.3.2 Rich and Waters' View

Rich and Waters have developed the PSBDARP concept into the AID paradigm [8]. This paradigm comprises three phases that are key to engineering problem solving.

| | |
|---|---|
| Abstraction | Choosing the details to ignore so as to reduce the problem to a familiar form. |
| Inspection | Recognizing a known approach that is applicable to the problem at hand. |
| Debugging | Modifying the approach to represent the actual problem more closely, as by focusing on previously ignored factors, or by considering exceptions to the more general rules created in prior analysis. This phase also comprises alterations made to the plans due to faulty analysis. |

A crucial component of this problem-solving paradigm is the collection of standard problem forms and solution approaches used in programming, called

"cliches". Most of the cliches Rich and Waters have studied are low-level implementation forms. For example, when the programmer wishes to perform some action such as file access for searching for records with a certain property, he may invoke a cliche for the file control portion of the task, another cliche for the iterative record-reading portion, and another cliche for the testing and subsequent alteration of records.

The AID paradigm emphasizes the role of simplification or abstraction as a tool commonly used by programmers to battle problem complexity. To support this Rich and Waters' appeal to the analogy in electrical engineering, wherein it is a poor method of circuit analysis to invoke the most complex model of the circuit first. A good first approximation is to simplify the circuit by making assumptions about the ideal nature of the components. Such an analysis, which can be performed "on the back of the envelope," may be adequate for the amount of accuracy demanded. Only when greater accuracy is demanded should the ideal models be replaced with more complex ones, requiring larger envelopes or more difficult equations to describe their behavior.

In software engineering, introducing simplifications, even though their inaccuracies may be known, helps in initial problem investigation because it helps the programmer climb the "Software Development Curve." In the average piece of software, early versions taken at constant time intervals represent large strides toward the goal. As the desired functionality is approached, each successive version begins to comprise more fine-tuning and less gross code-installing. For this reason, I consider the change in functionality versus change in version curve to be roughly a rising exponential. Dually, I believe that the early inter-version differences are greater than the later inter-version differences. Practically, however, we don't have to wait forever to achieve desired functionality because software development is discrete. Eventually, we'll be exactly at 1 or close enough to neglect the shortcomings. One method of problem solution would be to plunge forward against a difficult problem without simplifying. This method would take several (perhaps many) versions to reach 25%

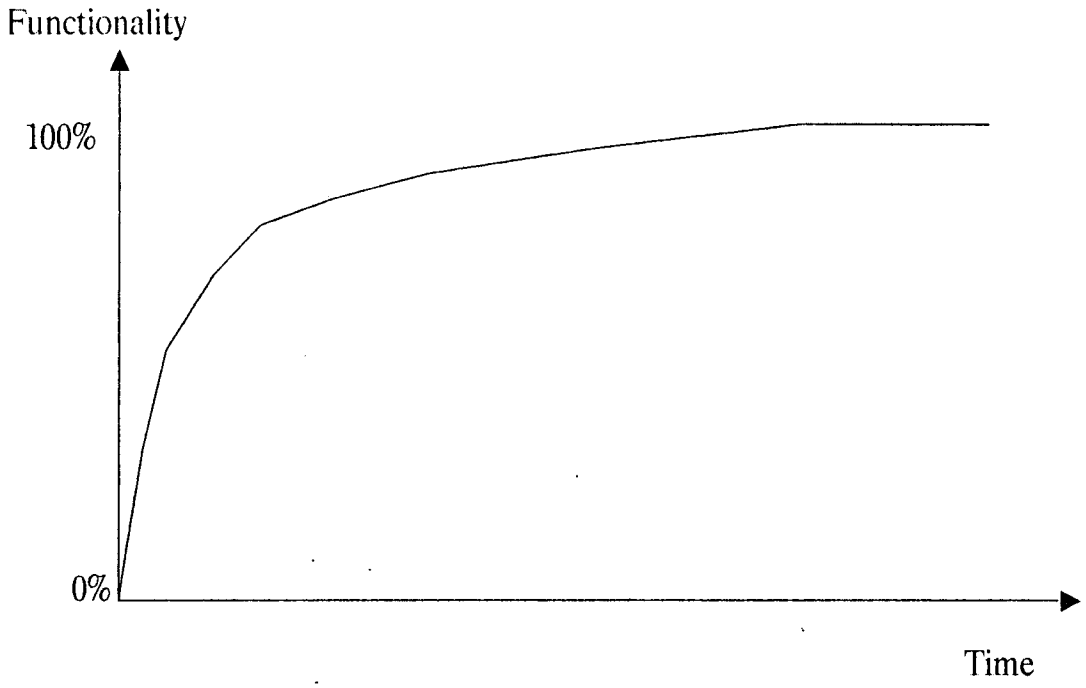



**Figure 1-1:** A Typcial Software Development Curve

functionality starting from zero. By using an alternative method, the programmer can use simplifications to start out higher, so that after the first version he has 25% functionality.

Rich and Waters discuss the concept of cliches, nuggets of algorithms or data structures, in [8]. Cliches, taken from the project library, are supposedly already correct. By properly assembling them, the programmer can build up a skeletal solution quickly, allowing the placement of specifics and modifications of the cliches to fit the particular problem to occupy his attention. It could be said that the goal of the Programmer's Apprentice project is, colloquially, to force the first 50% of the project into the first 10% of the time (performed automatically by the intelligent assistant), allowing the last 50% of the project to require the remaining 90% of the time. Perhaps this is overly optimistic, but without lofty goals little would be accomplished.

The effort of Rich and Waters has been directed not so much at analysis of bugs as toward analysis of the techniques of software development and of the parts of the Software Development Task that can be automated. The realm of bug analysis as a means of measuring what novice programmers (or experts, for that matter) know about their tasks, has been left for the cousin field of AI, cognitive science.

## 1.4 A Cognitive Science Perspective

The state of the art in bug analysis is the work of Elliot Soloway and his colleagues at Yale. Soloway's subjects are primarily students in introductory programming classes, working on learning the language's syntax and how to write a program.

The goal of Soloway's efforts is to make an intelligent programming critic that can analyze programs and highlight errors in the plan much as a compiler checks for errors in the syntax. One crucial step in this development is for a human critic to perform systematically such analysis and error categorization. Soloway's group modified the operating system of the computer used for the assignments to save a version of every program that was syntactically correct. In his analysis, however, he used only the first version [4]. His method is to compare the first try with elements of a finite and small set of implementation strategies for the given problem, and focus on the structural discrepancies between the recorded program and the idealized one. By measuring in some subjective sense how close the recorded version is from the various idealized implementations, his program analyzer is able to decide which idealized program the given version was most similar to. This measurement and comparison, Soloway concedes, is not easy. The group's outlook is that bugs encountered in introductory computer programming classes are malformed attempts at correct implementations.

Both Soloway and Rich and Waters rely on the concept of the *plan* as a central force in program development. In the Programmer's Apprentice project [7, 8, 13], the

intelligent assistant has a formal representation of the plan of the software under development. In the introductory programming classes, the plans are never formally stated, perhaps even never wholly understood by the programmer, but can be divined or guessed from the structure of the program.

# Chapter Two

# Focusing the Study

It is not feasible to attempt to study the entire space of bugs in programming. There are simply too many factors involved in the general topic of "programming" to make one study, or even one research group's multi-year work suffice to make a thorough analysis of the entire meaning of bugs.

On the issue of what to measure regarding the meaning of bugs, there are many outwardly visible features of a particular programming project, including

- the rate of bug introduction,
- the rate of bug detection,
- the rate of bug identification and removal, and
- the speed at which the successive program versions converge to the "desired" program.

I use the term *desired* rather than *correct* because any syntactically correct program is a correct one in the sense that it performs some task or computation. It just may not be the one indicated in the problem. The program that performs the task indicated in the problem is the desired program.

The factors that might influence these rates include

- the language being used, e.g. BASIC or Lisp
- the size of the project, e.g. sort algorithm or operating system
- the programming environment, especially the programmers' familiarity with it

- the cost of compiling a file
- the number of participants, i.e. solo or team effort
- the time constraints on the project, e.g. needed yesterday or at leisure
- individual differences, including sex, age, and education.

Because of all these factors, the researcher's lot is a difficult one. He must choose a specific segment of the problem space: a language, a task, an environment. He must attempt to control the individual differences (easily done by having a large sample space, and a captive subject pool such as the entire enrollment of a programming class). He must choose a salient metric. I use the term *factor space* or *problem space* to designate all possible combinations of factors that may influence bug appearance.

An alternative means to the end of a general theory on bug types is a piecemeal analysis of the problem space, segment by segment. By studying a certain segment or set of segments that have a common factor, we can generalize or summarize the bug types in the segments *within* the segments studied, and then hope to extrapolate the bug types in the segments *without.* As the various segments of the factor space are investigated, the knowledge of bug types will become more knowledge and less extrapolation.

The study detailed here was an investigation of the bug types of a certain segment of the factor space. The particular characteristics of this segment are identified in Chapter 3. There was a twofold purpose to this study: to categorize and enumerate the types of bugs encountered, and to gain sufficient insight to propose a working vocabulary or language for discussing bugs. A language for discussing bugs forms part of a cognitive theory of the software development process, which could lead to better programming tools.

# Chapter Three

# The Experiment

The factors of the experiment that describe exactly the segment of the factor space are the following. CLU, a strongly typed high-level language, was used for an assignment slightly larger than an average workday's coding. The subjects were given 8 hours to develop code, with the option to spend extra time if they wished. They were given the instructions (Appendix I) the day before to familiarize themselves. The programming environment was the same one in which they originally learned the CLU language, as part of an MIT course in Computer Science. Each participant worked alone, and there was no stated penalty for not finishing or reward for finishing the project. Every version of every file was preserved, but subjects were not told the reason for this, or the intent behind the ancillary forms (Appendices II and III). Subjects received an honorarium for their participation.

## 3.1 The CLU Language

The specific task entailed using CLU [6], a strongly-typed language that supports abstraction and separate compilation of modules. Strong type checking, performed by the compiler, ensures that the various modules in the project mesh together precisely. Type-checking also prevents the invocation of an operation on operands of inappropriate data types. One of the facilities provided in the CLU editor is an indenter that performs fairly complete syntax checking. The indenter is a useful reminder for those who work in several languages and have forgotten whether the syntax is, as in the C programming language, Figure 3-1, or as in the CLU language, Figure 3-2. Similarly, the indenter flags missing loop terminators, making it very difficult for a user of the indenter to have crossed loop endings.

```
if <expression1> <statement1>
  else <statement2>
```

**Figure 3-1:** C syntax for if-statement: only one statement may appear

```
if <expression2> then <body1>
  elseif <expression3> then <body2>
  elseif <expression4> then <body3>
  else <body4>
  end

and <bodyN> represents a series of statements.
```

**Figure 3-2:** CLU syntax for if-statement: many statements may appear

CLU does not support global variables; consequently, all references to variables must be resolved in the current block of code. Programmers may be used to referencing a global variable, say a grand total, in other languages. When they begin to use CLU, they have to alter their coding style to provide for modification of variables in the calling procedure. See Figure 3-3. This can be accomplished by having the caller pass the variable into the called routine, and then having the called routine return the modified value of the variable, which gets stored under the same name as before.

```
caller = proc ()
    grandtotal: int := 0
    more: bool := true
    while (more) do
        grandtotal := modifytotal (grandtotal)
        %decision whether to alter the value of more
        end
    end caller

modifytotal = proc (current: int) returns (int)
    newtotal, addend: int
    %prompts for addend
    newtotal := current + addend
    return (newtotal)
    end modifytotal
```

**Figure 3-3:** To access *grandtotal*, the called procedure must be passed it.

This example is contrived, for such a simple task as addition, but when the problem becomes more complex, and the variables under scrutiny "belong" more to one routine than to another, this contrivance or something like it becomes necessary. Eventually, the programmer appreciates the security gained by there being no unexpected name resolution.

In traditional languages, name resolution uses some search sequence to attempt to resolve a referenced variable name; sometimes different members of the same team use the same variable name, and unexpected references or modifications can result. See Figure 3-4, which contains a PL/1 variant. Module toplevel calls file_reader, which calls modify_record. Toplevel has a variable called **status**, which stores state information. Jane Doe codes procedures toplevel and modify_record; John Roe codes module file_reader. Unbeknownst to Jane, John has declared his own **status** and set it to some arbitrary value. When modify_record references **status**, it may be resolved to the value of **status** in the file_reader module. This behavior will be unexpected at best, disastrous at worst. In CLU, this issue is avoided.

CLU is not a functional language: it allows multiple assignments to the same variable. Data types that can have multiple assignments are called modifiable or mutable data types, while those that cannot be reassigned are called immutable. CLU supports both mutable and immutable data types. The different types of bugs that appear in a language with side-effects compared to the types that appear in a language without side-effects is a complicated issue. As we will see in detail in Section 4.3.1, the problems resulting from side-effects can be substantial. Functional languages, designed for highly pipelined architectures, would not necessarily share the same bug types. Languages that are especially designed for a specific domain, such as for expert systems, formal theorem proving, or a message-passing system, would also experience their own particular bug types.

```
toplevel:       PROC OPTIONS (MAIN);
% Author: Jane Doe
    DECLARE status FIXED BIN (31);


        status = interactive_user;

        CALL file_reader (input_file);


        END toplevel;


file_reader:    PROC (database_file);
% Author: John Roe
    DECLARE databse_file FILE;

    DECLARE (old_status, status) FIXED BIN (31);

        old_status = status; % preserve the status

        % mark that we're in the middle of changes
        status = database_inconsistent;

        CALL modify_record (this_record);

        status = database_consistent;
        % we're finished with changes

        status = old_status;    % reset to old value
        END file_reader;

modify_record:  PROC (db_record);
% Author: Jane Doe
   DECLARE db_record CHARACTER (32);

        IF status = interactive_user THEN
        % prompt user for new value
            ELSE % delete the record
                db_record = '';

        END modify_record;
```

**Figure 3-4:** Unexpected name resolution can result in module *modify_record*

## 3.2 Overview of the Task

The task involved writing a program that would simulate a world in which two species live in competition. Various parameters affect the ecological balance between the species. The struggle between the ranks of the predator and the prey is displayed on the terminal screen, which represents the watery surface of the planet. As in many video games, the edges of the planet wrap around, so that the planet's surface describes a torus. (Appendix I is the document given to the subjects contains the instructions and problem specifications.)

## 3.3 Programming Environment

The task was carried out first by me, in order to prove that such a simulator could indeed be written in accordance with the specifications. Then it was written by two groups of subjects. The computer system used was a DEC-20 running TOPS-20. The CLU language was used [6], since this was a language on a system that both the author and subjects were familiar with. The project instructions stated that the simulator was for use on VT100's only, and any other terminal types did not have to be provided for. EMACS and TED editors were used, along with other CLU-related and systems software packages.

## 3.4 Methodology

Whereas Soloway collected the syntactically correct versions and only examined the first of these, I did not have the luxury of a modified operating system. I judged this problem to be too complex for a one-version sweep to be indicative. I chose to keep each version of each file written to disk by the subjects. With all the versions, I had the freedom to decide what to examine: first versions, number of versions, syntactically correct versions, final versions or whatever I wanted. I had the subjects annotate their progress with forms of my own design (see Appendices II and III). Using these forms,

the files, and a source-file comparison tool, I tracked the progress of the development of the various files. In the pattern of Soloway et al [4], I attempted to categorize the various problems with the code. The results are discussed more fully in Chapter 4. The drawbacks of this methodology are elaborated in Section 5.3.

# Chapter Four

# Analysis of Data

My initial intent was to use methodology identical to that of Soloway to categorize the bugs. However there were problems applying this system: not every error was an error of commission. There were errors of omission also.

## 4.1 Omission vs. Commission

An error of commission is one in which the programmer actively attempts to construct working code to solve a problem, but fails to write the proper statements. An error of omission is one in which the programmer either fails to notice that a certain task needs to be performed, which constitutes an unintentional omission, or intentionally leaves the code out for some reason. The failure to notice that a certain task needs addressing is an oversight that may become obvious in the testing phase, or later in the coding. Intentional omissions may be the chosen course of action

- if the programmer wishes to compile what he already has, knowing full-well that he has left out a large chunk,
- if he is merely writing the specification of the various modules, to ensure they all have consistent interfaces, or
- if he is working top-down and leaving the bulk of a task solution to an as-yet unwritten subroutine.

These are not the only reasons why software changes in between versions. Sometimes, the programmer introduces diagnostic routines, to perform what some engineers call a sanity check. Other times, the programmer reverses or rethinks a design decision that is not an error *per se*, because many programs can evince the same behavior, but does represent a path of development that is retraced.

## 4.2 Bug Count

Seven subjects, including the author, participated in the experiment in some form. There were differing sets of instructions for the different groups, but common to each group was the preservation of some record of program development and inter-version differences. Two subjects produced data that was unusable or almost unusable. These were the files that did not accumulate enough versions to show any inter-version differences. One subject achieved a partial solution, with some identifiable bugs; the remaining four subjects each achieved a workable simulator, unearthing a handful of bugs.

The bugs are enumerated in Table 4-1, patterned after the tables in [5].

**Table 4-1:** Subject numbers and corresponding occurrences of bugs

| Bug type | S1 | S2 | S3 | S4 | S5 | S6 | S7 | Total |
|---|---|---|---|---|---|---|---|---|
| Missing Guard | 3 | 3 | 3 | 3 | 0 | x | x | 12 |
| Malformed Guard | 1 | 0 | 0 | 2 | 3 | x | x | 6 |
| Malformed/Missing Initialization/Finalization | 1 | 2 | 1 | 1 | 2 | x | 1 | 8 |
| Missing Output | 1 | 0 | 1 | 0 | 0 | x | x | 2 |
| Malformed Input | 1 | 0 | 0 | 0 | 0 | x | x | 1 |
| Missing Declaration | 0 | 1 | 0 | 0 | 0 | x | x | 1 |
| Malformed Plan | 1 | 0 | 0 | 1 | 0 | x | x | 2 |
| Malformed Update | 1 | 0 | 0 | 0 | 0 | x | x | 1 |
| Missing Update | 1 | 0 | 0 | 0 | 0 | x | x | 1 |
| Missing Return Statement | 0 | 0 | 1 | 1 | 0 | x | x | 2 |
| Missing Parameter | 0 | 4 | 2 | 0 | 0 | x | 1 | 7 |
| Switcheroo | 1 | 0 | 1 | 1 | 1 | x | x | 4 |
| Omissions | 1 | 1 | 0 | 1 | 0 | x | x | 3 |
| Other | 1 | 1 | 0 | 2 | 2 | x | x | 6 |

Soloway's bug categories involved the following concepts. A Guard is a piece of code that controls execution or looping, or filters inputs in any way. An Input or Output statement is one that performs I/O with either a file, a terminal, or the user. A Declaration statement is one that identifies the data type of a variable, or for other

languages, announces the external features of foreign procedures. An Initialization statement is one that sets a variable to a known value before it is used, to ensure that there will be no surprises if anyone attempts to evaluate it. An Update statement changes the value of some variable. Statements in Soloway's work were judged to be Missing if they were not present but needed, Malformed if they were present but not quite correct, Spurious if they were present but not needed, or Misplaced if they were correct but in the wrong location.

The table contains a total of 56 errors, using Soloway's convention of counting an error that occurs several times in a small area as one error. The first few entries from the table are similar to those from Soloway's catalog, resulting from some of the matches of statement type and adjectives. The last five are categories that have to be introduced to explain the causes for version changes. The category called Malformed Plan is demonstrated in [5] with one bizarre error; the sub-category Soloway names is much more indicative of what constitutes an entry into this category: "Update Embedded in Test," that is, an object that was updated while it was being tested.

The Other category holds the errors that still do not fit into the breakdown. A redesign was placed in this category: a change in which the subject decided to relocate a piece of functionality. A data type mismatch fell into this category, when a subject failed to advise the compiler about the return values of operations in other modules. Instances in which subjects chose the wrong variable in a procedure with several pairs of array indices went into the Other category also. In the *world* cluster, there were often several sets of x and y coordinates, used to serve as array indices. Choosing the wrong set of indices is considered a different error than choosing the wrong coordinate from the pair. Finally, malformed array descriptors ended up here. The malformed array descriptor error differs from the wrong-choice of array descriptor, in that the malformed descriptor involved the correct variable names, but the wrong offsets; the wrong-choice error involved the wrong variable names.

In the Omission category, we find such slips as merely forgetting to type in the meat of the procedure (an obviously well thought-out piece of code, but just a slip of the fingers) and an unforeseen but necessary function call. Of these two errors, one is an unintentional omission due to forgetting to type code in, the other is an unintentional omission due to not knowing that a function had to be written. Since they are both unintentional errors, their positions on the awareness spectrum mentioned earlier on Page 23 are close together. One is a slip that requires all of a few seconds to recode, since the intended code was already envisioned. The other is an unexpected piece of functionality that must be created from scratch, which requires thought about how to solve a new problem. Even though this issue, too, may be quickly solved, there is a longer time before the software in this case is returned to an equivalent level of functionality. Intentional Omissions appeared in this experiment, but they were best typified by versions that were not counted. A typical intentional omission is the leaving out of code until the module interfaces have been written. Versions that contained modifications only to module specifications but contained no code were not counted in this analysis.

## 4.3 Interesting Bugs

### 4.3.1 Modification of Object under Scrutiny

The most interesting bug was found in two subjects' code. One facet of the ecosystem simulator was the removal of dead animals. Subjects stored the data objects for these animals in an array, and marked a record element to signify that the animal was dead. When the cleanup phase came, they tried an algorithm similar to that in Figure 4-1. Unfortunately, the CLU primitives with which they chose to implement the control of the array indexing examined the size of the array once, at the beginning of the **while** loop, and then attempted to perform an action on each member in the array. The obvious flaw to the armchair coders is that as soon as the array is modified by

```
while there are more animals left do
  if current animal is dead then
    swap current animal with animal
        at high end of array
    delete highest animal in array
    end
  end
```

**Figure 4-1:** The first algorithm for animal deletion

deletion of the highest element, its size changes. This change is not reflected in the stored value of the array size. The error manifested itself in CLU as a *bounds* signal when the "current" animal was located at an invalid array index. The problem was easily corrected by changing the loop control to a dynamic evaluation of the array size.

### 4.3.2 Switcheroos

While it is not an error that seems to import much, one error also made by two people concerned the code that decided if a given ocean area was occupied. The actual before and after snapshot follows:

```
occupied = proc (sea: cvt, x: int, y: int) returns (bool)
    if sea[y][x] = open_sea then return (true)
       else return (false) end
    end occupied
```

**Figure 4-2:** First attempt to solve a simple problem

```
occupied = proc (sea: cvt, x: int, y: int) returns (bool)
    if sea[y][x] = open_sea then return (false)
       else return (true) end
    end occupied
```

**Figure 4-3:** Problem corrected in second attempt

The only difference between the two versions is the switch of true and false.

Were this an isolated incident, I might think that the coder just got confused. However, there were similar errors, in circumstances requiring the creation of an array descriptor or index out of either some x coordinate or some y coordinate. Two subjects wrote code in which the wrong coordinate was the first array index. They corrected this error in the next version. In comparison, a certain juncture contained two partially applicable worlds, the old one with the animals and state before the current turns' changes, and the new world representing the old world modified by the changes. In this instance, only one subject chose incorrectly between current and previous sets of coordinates. Perhaps something about the difficulty of the choice made the subjects think more carefully about which sets of coordinates were appropriate.

But of the five subjects that coded far enough to have encountered the issue of choice in a two-option universe, there were four errors (comprising the Switcheroo category.) Two made the error of reversing the boolean values, and two made the error of switching x and y. This leads me to believe that, in an instance where the alternatives are finite and few, and could be exhaustively enumerated, and especially when the cost of compilation is low, a coder is under less pressure to write the version that will work as planned on the first try. That is, when a programmer knows a line of code is either true and then false or false and then true, he might be tempted to be sloppy. His first try would be an arbitrary arrangement of true and false. If that didn't work, he would switch them, as the "correct" working version has to be either the new or the previous version.

This does not suggest that we deliberately avoid thinking about whether the order should be true then false or false then true, rather that we realize that there will be no catastrophe if we get them mixed up and have to switch them in the next version.

## 4.4 Difficulty with Soloway's System

The bugs encountered in this project cannot be easily reconciled with the categories of Soloway et al. Some of the bugs resulted purely from inability to foresee all conditions at a given juncture. Some of the bugs resulted from the inability to enumerate all the boundary conditions in the input space. The relative balance between types of bugs found in this project is due in great part to the complexity of the task. The presence or absence of whole categories, such as the syntax categories, is greatly affected by the choice of the CLU language. While there were control flow errors, there were few language-related errors.

### 4.4.1 How Complexity Relates to Bug Types

Complexity also influenced the bug types. The programs analyzed by Soloway's group entailed tasks such as converting elapsed time from two input times and "The Noah Problem", in Figure 4-4.

> Noah needs to keep track of the rainfall ... to determine when to launch his ark.... Your program should read the rainfall for each day, stopping when Noah types "99999".... Your program should print out the number of valid days typed in, the number of rainy days, the average rainfall per day over the period, and the maximum amount of rainfall that fell on any one day.[3] [5]

**Figure 4-4:** Problem statement for one of the tasks examined by Soloway

In a program consisting of a single routine, such as the Noah program, there will be no returned values; typically input and output will be performed with the terminal or files. In a program consisting of many modules, passing and returning parameters is crucial, and especially in a language such as CLU, which does not support global variables. Consequently, categories such as Missing Parameter or Missing Return Statement arise.

Consider someone who builds the same kind of program time after time. If he is doing something quite similar to a previous project, he is coding at a difficulty level

---

[3] page 1

lower than necessary. If he knows he will be writing many similar programs, his time might be better spent in automating the program-generation process, or separating the common block from the varying block. Then, when the project changes, he can reuse the common block and only have to write the new task-related part.

The underlying idea here is that if we don't operate at the peak of our intellectual capacity, then we are not learning anything new. Only by trying to write a program that we haven't written before, or that has some risk to it, are we challenged by the assignment. If we write unchallenging programs, we are probably aware of most of the boundary conditions and "quirks" related to the problem. If we are in uncharted territory, we are perhaps ignorant of the quirks. We can anticipate some of the non-intuitive difficulties associated with the problem, but cannot decisively identify all of them. Thus there is a tradeoff between how familiar we want to be (are) with the problem domain and its quirks, and how challenged we want to be (are).

Similarly, when one is challenged by a problem not fully understood, there is always the tradeoff between what to concentrate one's ability on and what to ignore for the time being. In a typical problem, this tradeoff may manifest itself by our concentrating on the main inputs first, and leaving the boundary inputs until later, when we know that the code we wrote for the main inputs is correct.

### 4.4.2 Differing Levels of Complexity

In Soloway's programming tasks, it was usually within the abilities of the novice programmers to envision the whole task before the coding began. Thus, there were no unforeseen circumstances. There were not even multiple routines, as the Noah program in [4, 5, 10] is intended to be a one-page, one-routine program.

When considering only this segment of the problem space, Soloway did not even begin to discuss the problems of interface, division of labor, and so on. Soloway also

restricted his analysis to the first version of the task; he compared the plan of the first version to a workable plan: he did not trace evolution of code or of plan. The language he used to describe the bugs encountered in this small segment suffices for the segment itself, but is not general or extensible enough to describe or explain the types of errors encountered here. There were some errors in the video-simulator data that fit in well with Soloway's bug types. One can certainly make low-level errors such as missing or malformed initialization of a variable in a higher level language like CLU. However, the errors that fall into these categories are a smaller percentage of the total number of errors, since there are vast new types of errors that would never have arisen in Soloway's chosen domain, and types of errors that did occur in Soloway's chosen domain that are impossible to make due to the features of CLU and its facilities.

Therefore, a gap exists between the state of the art in bug description and the language that suffices to describe the bugs encountered in longer and more complex programming tasks. Any language that describes bugs found in the domain of larger-sized programming tasks such as the one I investigated must include as a subset the ability to describe the same bugs Soloway encountered. Perhaps this containment would be achieved by using Soloway's categories exactly, or perhaps a new equivalent method of description would be composed.

# Chapter Five

# Onward

## 5.1 A More General Method of Bug Description

We have seen that more complex programming tasks create bug categories that do not fit in well with the existing taxonomic structure of Soloway. What we would like to have is a method of describing general bug types, irrespective of the nature of the programming task.

Sussman writes "The key to understanding one's errors is in understanding how one's intentions and purposes relate to his plans and actions."[4] This is fine for the later phases of the engineering process, but will not explain how the wrong intentions originally came to fruition inside our heads in the first place. We have to be able to describe bugs that result from misconceptions rooted earlier in the development process. In [9], Ruth mentions that errors can crop up at many stages in the engineering process. Indeed, an error in the simulator experiment appeared in the problem specifications. I asked for algorithms that were **O**(number of animals), and not **O**(size of the ocean), while my own code to solve this problem was **O**(number of animals)$^2$. Thus, we see that a method of bug description has to be able to describe many different sorts of bugs, not all programmer introduced.

### 5.1.1 Framework

The components of a language of bug description are, first, a set of categories, $\mathfrak{C}$, into which the bugs will be grouped and, second, a set of metrics, $\mathcal{M}$. There will be at least one metric per category; there may be alternate metrics for a given category,

---

[4] [11], page 13

pertinent to the different perspectives of analysis. Perhaps a better term for category is dimension, which is more relevant to the idea of possible graphing or plotting errors in |C| dimensions.

These components can be analogized to some demographic data used in polls. People who respond to the poll can be categorized in terms of their income, their political affiliation, and their age. Each of these categories has an obvious measure, but a category such as height could have a measure from the metric system or from the English system, or from some other system. Just as one metric is not the definitive metric for all purposes, I allow potentially several metrics for the categories.

The method of bug analysis proposed is the following:

1. Select a set of categories, C. I propose one that may suffice.
2. For each category, select a metric with which to situate the given error within the category (or place the error on that category's axis).
3. For each error, find the |C|-tuple of values that describes the error.
4. Tabulate the errors, or (optionally) plot the tuples in |C|-space.

Since the level at which we code today is not the level at which we coded 20 years ago and is probably not the level at which we will code 20 years from now, the framework must be pliant. If software engineering radically changes, new categories can be added to the set I propose, or alternate metrics may be developed. Significant advances made in expert assistants, multiprocessing, distributed applications, and so on may radically alter the way we construct software. Such changes could be handled by the system of categories and metrics with proper extension to the existing system.

### 5.1.2 A Proposed Set of Categories

There are three dimensions or categories on which bugs can be classified. The most obvious category I call Severity of Error. This is how costly or serious the error is to the development process. Another dimension is the location of the mental mistake that produced the bug. This dimension, which I call Locus of Error, represents at what level of the thought process the engineer had a misunderstanding or oversimplification that was not applicable to the real world behavior. A third dimension is necessary to explain some of the behavior observed in the experiment. This dimension, which I call Intent of Error, is the consciousness with which a programmer ignores some element of the task, as discussed previously in Section 4.4.1. That is, if an error had a high Intent of Error, it was a "planned" error. This dimension has the added advantage of being able to handle the errors of omission that Dijkstra would demand that I acknowledge as errors.

### 5.1.3 Proposed Metrics

Since the engineering process has been so widely studied, the realm of Locus of Error is the most easily measured. The metric will be the name of a phase of the engineering process. The following analysis is based on the assumption that the engineering process goes through these possibly overlapping phases:

- Specification writing, as by professor, supervisor, or committee
- Specification interpretation, as by students or engineers
- Problem fracturing, as in a large project such as an operating system, compiler, or editor (who codes what)
- Design and Interface creation
- Algorithm or Plan formulation (choice of algorithms)
- Algorithm translation or Code Generation

- Testing omissions ("non-bugs")

In order to measure the Locus of Error, the analyst must bring to bear a substantial amount of knowledge about the problem, perhaps similar to Soloway's Plan-Goal tree, that includes the various methods of solving a particular problem. It is not always easy to determine Locus of Error, as many of these phases overlap, and misconceptions are not always uniquely the product of one phase. Sometimes the design of a simple subroutine will imply or force the resulting code, other times the design will lead up to a brick wall, requiring careful consideration of code to achieve the desired end. Progress is sometimes forward (from specifications to code), sometimes backward (from testing to reformed design). Because of the unpredictable progress, Locus of Error is not uniquely determinable. However, approximate judgments should suffice if the space of errors analyzed is large enough.

A weighty problem is how best to measure Severity of Error. I propose and then criticize several metrics.

Metric 1: Amount of code changed. This is bad because it is language-dependent. Consequently, what can be corrected in one line of a high level language might be just as serious a bug as something requiring many pages of machine code. Additionally, the primitives provided by one high-level language to achieve a goal, such as a random-number generator, are not necessarily equivalent to the primitives of another language. Therefore, we'd like to introduce some relative measure.

Metric 2: Relative amount of code changed. The major drawback here is that in order for there to be a relative measurement, especially amidst various programming languages, there must be an invariant denominator that answers the question, "relative to what?" I have serious difficulties coming up with a solid, standard denominator by which I can size pieces of code. The suggestion of using size relative to the size of the system or project under development has the following drawbacks: the size of the

project is language-dependent, and is hard to measure until it's finished. Also, some functional block that is small relative to the size of the project might be absolutely crucial to the functionality. A pitfall in this critical element might derail the whole project.

Metric 3: Amount of "plan" changed. Using the plan notions of Rich and Waters, we can translate from code to underlying intentions. However, each line in a plan is not as "action-packed" or "information-filled" as every other line, so "1 line of plan" is a non-constant unit.

With any measure that attempts to show changes in the history of a program on an electronic medium, there is the problem of various stages of implementation. What if the problem has not yet been implemented, but the algorithm has been designed and scratched out on paper? There is still some amount of backtracking, but sheer recording of versions of a piece of code will not record it. Therefore, we need some measure that does not focus on the medium of progress (i.e., pencil and paper, electrons in someone's head, or electrons on a magnetic disk).

How about some measurement in time? Any measurement in time will by its very nature encounter problems of individual differences. It takes one person three hours to do what it takes another 1 hour to do. Any time measure can be compared only to other measurements concerning the same engineer. However, the drawbacks due to individual differences can be overcome, or controlled for, given a large enough sample space. Therefore, I do not reject time as a metric for Severity of Error.

Metric 4: Amount of time to localize the bug. There are several phases to debugging, and I am by no means an expert debugger. I would classify the steps as

1. realizing there is a discrepancy between expected and actual behavior

2. localizing the discrepancy to a certain segment of code, or a missing portion

3. proposing a correction

4. implementing it

5. rebuilding the executable image, and

6. testing to make sure the level of performance of the new image performs at least as well as (identically to) the old one and behaves correctly on the segment of the input space that triggered the error.

Certain bugs may be trivial to find, but difficult to correct. A programmer or a team may be set back in work quite a bit (possibly on the order of man-months, or even possibly so far that the project has to be scrapped) by a bug that is pinpointed after five minutes of code analysis. Therefore, time to localize is not a good metric by itself.

Metric 5: Amount of time required to bring code to equivalent level of functionality. This is somewhat confusing because part of the time required to provide functionality is pure coding time. It's hard to distinguish what is retro-fitting an algorithm from what is new development.

Metric 6: Amount of work lost (measured in time, not in pages of code that are thrown away). What if there has been no work lost, and there is little backtracking because the problem appears as soon as its domain is examined? It is not quite a bug; it is more of an aspect of the problem that the programmer has chosen to ignore or has been lucky enough to spot before damage gets too great.

Metrics 5 and 6 are closely linked. Each one has some good features. It seems intuitive that errors that set you back only a bit are minor; errors that cause a whole division of engineers to lose their jobs are more serious. These two metrics are not the same, as can be seen from Figure 5-1. Whether it is better to measure from perceived level of functionality (before the bug appeared) down to actual level of functionality (what the software is truly worth) or from actual level up to the not-yet-achieved level of functionality (when the correct code is in place) is debatable.

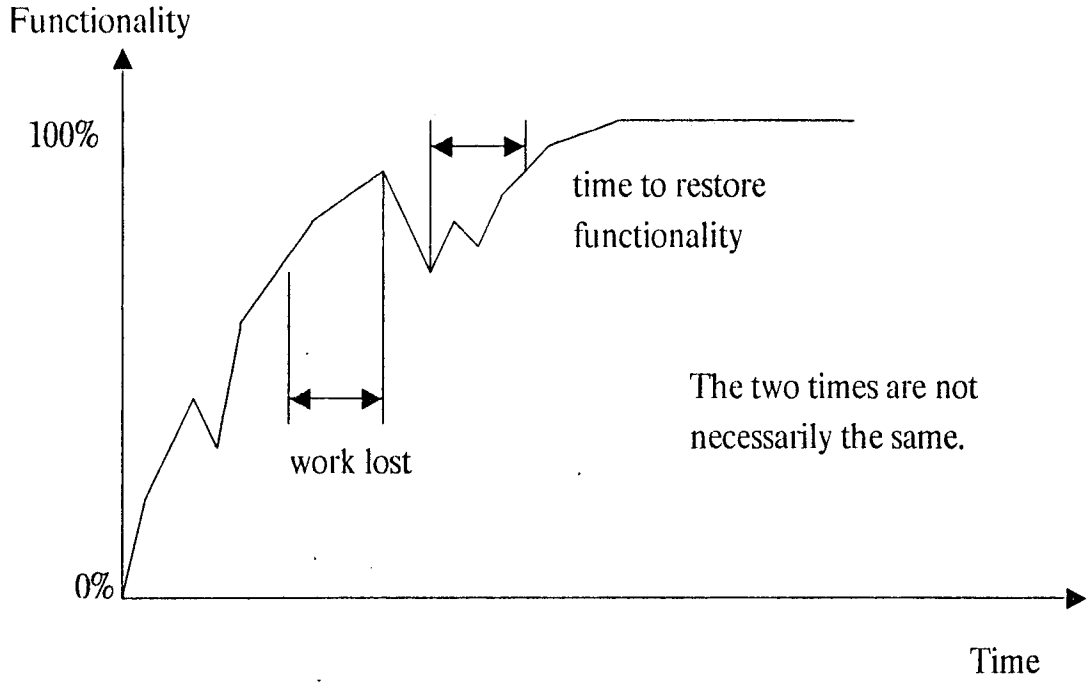



**Figure 5-1:** A portion of the Software Development Curve, in detail

Some composite metric might be more applicable to a specific project, such as the product of the time to localize with the time to restore functionality. Such a metric might be appropriate in the case of a highly-complex, large project near completion. Perhaps the maximum of time to localize and time to restore functionality is appropriate under different circumstances. Armed with a diverse set of metrics to measure Severity of Error, the analyst has the freedom to select the most appropriate one, or can try to find the most appropriate one.

In Section 4.1, two potential reasons for ignoring aspects of a problem were mentioned. The metric for Intent of Error has two required parts. The first component is the answer to the question, "At what phase in the Engineering Process did you realize that you were ignoring some (classes of) inputs?" The second component is the answer to the question, "At what phase in the Process did you realize the exact nature (i.e.,

exactly how many there were, what was their common feature, what was the boundary to the class) of the ignored inputs?" More than one facet of this metric is necessary, because an engineer could have a fuzzy picture of something missing in the design phase, but only get a concrete grasp that the class was precisely even numbers from 1 to 9 that caused problems. A complete measurement would be a statement such as "Network transmission errors in general were identified in the design phase, but due to the mercurial nature of performance across networks, I decided to handle each specific error as it occurred."

This metric should carry enough information to gauge Intent of Error, but to know as accurately as possible, one would need a graph of the programmer's understanding of the nature and scope of the ignored problem versus time or versus phase in the engineering process. An exact measurement of this graph would be impossible, because the programmer can only give general ideas based on introspection, but this graph, despite its fuzziness, contains more information than the previous metric.

One can ignore certain input classes at different phases in the process (sometimes for different reasons). Various reasons might be the higher priority of the software to handle a certain class of inputs, with the remaining inputs at a lower priority; the difficulty in producing code that will behave correctly on a certain class of inputs; or that a certain class of inputs is known to be easily solvable while another is proven to be $\mathcal{NP}$-hard.

One might imprecisely identify a set of unhandled exceptions possible in a given routine, for instance, all the possible error conditions when a file is accessed: protection violation, unreadable physical medium, I/O buffer overflow, and so on. Or one might be surprised in the debugging phase by a completely unexpected error. Because awareness of ignoring an issue is a fuzzy distinction, we need to have at least the two components mentioned.

## 5.2 Applying the New Scheme

For most of the errors identified in the analysis of data, determining the Locus of Error is relatively simple. Most errors were situated in the translation from algorithm or plan into code. There were a few cases in which the programmers' plans had bugs, such as the error demonstrated in Section 4.3.1. There was at least one instance in which the error was situated in the design phase, which I deposited into the Other category in Table 4-1. And, embarrassingly, there was the error in the specification, mentioned on page 32.

As to severity, the errors ranged from so severe as to be unsolvable (with the specification error) down to hardly severe at all, such as the malformed initialization of an ocean border as 19 dashes instead of 20.

Adding the category of Intent of Error allows consideration of many preliminary versions that had only specifications of how the different modules meshed together. The subjects who worked toward specifications before adding code can now have their progress accurately tracked, whereas before their early versions were filtered out of the analysis on the basis of having no code in them. In one instance an omission was a total surprise, and in another it was what we usually think of as an omission: an oversight.

The number of bugs was not large enough for there to be any advantage to plotting them in 3-space, but it would be more interesting with larger sets of bugs, such as those Soloway encountered.

## 5.3 Suggestions for Future Research

There were a few aspects of the experiment that should be changed were it to be performed again. It required two iterations to compose an adequate instruction document. I was initially unclear how much of the design to propose and how much to leave to the subjects. Were I performing this experiment again, or one of a similar size,

I would rigidly enforce a certain modular decomposition. Enforcing this may be difficult, especially if there is a large number of subjects, but the ensuing similarity of code will more than pay back this early effort.

Another drawback was that, in the absence of knowing at what time the files were written relative to each other, a modular dependence bug cannot be identified precisely, if each module has its own file. I would suggest that the whole program be developed using one file. This would have the disadvantage of requiring more storage space, and more time to compile -- this completely defeats CLU's separate compilation mechanism -- but again, the advantage of being able to see the whole project as it develops outweighs the resource inefficiencies.

The subjects in the experiment used different editors, which interact differently with the CLU compiler. One editor writes out the results of each compilation attempt, the other does not. Consequently, for some subjects, I had track records of what happened every time they invoked the CLU compiler. For other subjects, all I had was the source files and the paper forms on which they had written a symbol to indicate the results. If Module A calls Module B and the programmer has told the compiler nothing about Module B, the compiler will complain that the operations in Module B are undefined: it cannot check that the operands for the routines or the return values are correct, since it knows nothing about Module B. This difficulty would be avoided with a one-file project. Files containing no syntax violations were marked as having compiled correctly, even if they had undefined operations. This led to the Data Type Mismatch error, from the Other category from Table 4-1.

Additionally, despite the supplying of two clusters, *animal* and *prong*, the task was a bit too big to fit into one day's work. Either a smaller task should be fit into a single day, or more time should be allotted for completion. Greater preparation for standardized testing might help also. It was hard to compare the functionality of several subject's simulators due to the randomness inherent in the pseudo-random number

generator. It would have been easier to make some simple test cases to see if the various sticky points were correctly handled, such as "dead animals being reborn", "food-finding around ocean edges", or "two animals in one place".

If I were trying to enlarge the scope of the investigation, I would attempt to vary the size of the task, the language, or other features. An interesting test to perform is a similar task, with the subjects advised that the number of different versions is to be minimized. There might be an effect on the incidence of true-false errors.

## 5.4 Conclusions

CLU is a rich language, well-suited for analysis of programming errors. Many common programming pitfalls are easily avoidable with the CLU type-checking and other features. The errors that result can be given greater attention on the part of the investigator, since there might be fewer errors in CLU than in a different high-level language. The issue of the types of errors evident from CLU programs cannot be laid to rest on the basis of this elementary inquest.

The errors that did appear, however, sufficed to show that the current method of bug description lacks the complexity to describe more general classes of errors. The proposed system of bug description suffices for the types of bugs we discovered; furthermore, it is expected to be easily expandable for the advancements in programming techniques that will invariably be made in the years to follow.

# References

[1] Brooks, Frederick P. Jr.
*The Mythical Man-Month: Essays on Software Engineering.*
Addison-Wesley Publishing Co., Reading, MA, 1975.

[2] Dijkstra, Edsger W.
EWD618: On Webster, Users, Bugs, and Aristotle.
In *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, New York, NY, 1982.

[3] Guttag, John, and Barbara H. Liskov.
*Abstraction and Specification in Program Design.*
M.I.T. Press.
To appear.

[4] Johnson, W. Lewis, Steven Draper, and Elliot Soloway.
Classifying Bugs is a Tricky Business.
1982.

[5] Johnson, W. Lewis, Elliot Soloway, Benjamin Cutler, and Steven W. Draper.
Bug Catalogue: I.
October 1983.

[6] Liskov, Barbara, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Schiefler, and Alan Snyder.
*CLU Reference Manual.*
1981.

[7] Rich, Charles.
*Inspection Methods in Programming.*
PhD thesis, Massachusetts Institute of Technology, June 1981.

[8] Rich, Charles, and Richard C. Waters.
*Abstraction, Inspection and Debugging in Programming.*
A.I. Memo No. 634, MIT Artificial Intelligence Laboratory, June 1981.

[9] Ruth, Gregory Robert.
*Analysis of Algorithm Implementations.*
PhD thesis, Massachusetts Institute of Technology, May 1974.

[10] Spohrer, James C., Edgar Pope, Michael Lipman, Warren Sack, Scott Frieman, David Littman, Lewis Johnson, and Elliot Soloway.
*Bugs in Novice Programs and Misconceptions in Novice Programmers.*
Technical Report, Department of Computer Science, Cognition and Programming Project, Yale University, August 1984.

[11] Sussman, Gerald J.
The Virtuous Nature of Bugs. Conference on Aritificial Intelligence and the Simulation of Behavior, University of Sussex, July 1974.

[12] Warsh, David.
Writing the history of the computer.
*Boston Sunday Globe* 227 #69:A89, A94, March 10, 1985.

[13] Waters, Richard C.
The Programmer's Apprentice: Knowledge Based Program Editing.
*IEEE Transactions on Software Engineering* SE-8(No. 1), January 1982.

# Appendix I

# Wa-Tor Instructions

## I.1 Intro

Wa-Tor is a toroidal world where fish and sharks live in competition. The original idea for this project comes from the column on Computer Recreations in *Scientific American* December 1984. Your goal is to write the simulator described in the article, with a few efficiency improvements.

## I.2 Summary of Provided Clusters

### Prong

PRONG stands for Pseudo-Random Number Generator. It is really a function that returns a floating point **(real)** number in the range from 0 to almost 1. It takes one input but does not use it. It keeps track only of its internal state, which you don't need to worry about. Basically, at a given instant, *prong (0)* will return the same value that *prong (27)* would return.

### Animal

An animal is a generic data object that can be coerced into behaving like a shark or like a fish. See the specs for the animal cluster and the requirements for the differences in the sharks' and fish's behavior.

## I.3 Requirements

## User Interface

To me, user interface is secondary to pure functionality. I don't really care if the front end dies if you enter alphabetic where you needed numeric input, or reprompts. Whatever you have time to write, go ahead, but save the *bells and whistles* for the end. The necessary input parameters are the following:

1. Number of fish in initial population
2. Number of sharks in initial population
3. Number of days before a fish breeds (or, Age at which a fish breeds)
4. Number of days before a shark breeds (or, Age at which a shark breeds)
5. Number of days a shark has to eat (or, Maximum time-to-starve)

If you think it would be easier to write the simulator, or you want to get fancy toward the end, you might want to provide these additional parameters:

- Width of the Ocean (number of columns across)
- Height of the Ocean (number of rows down)
- Maximum Number of Turns to let the Simulation run

## Behavior of Fish

On a given turn, each fish eats some of the invisible and ubiquitous food (*read: you don't have to worry about feeding the fish or the fish starving; they never have to eat*). A fish decides to swim to an adjacent position randomly chosen from the empty squares that are directly to the north, south, east, and west of him. He does not swim onto an occupied square, so if his four neighboring squares are all taken, he doesn't move. If he does move, and it is time to breed, another fish of randomly determined age is created at an adjacent empty square. (I positioned all offspring at the vacated square.) When a fish gives birth, his age is set so that it will be */Number of days before a fish breeds/* before he gives birth again. A fish does not have to be exactly that age, just that age or

older in order to give birth.

## Behavior of Sharks

On a given turn, each shark attempts to find an adjacent fish to eat. Sharks are always hungry and can eat as much as one fish per turn. A shark eats a fish by occupying the same square as the fish. If a shark can eat any of several fish (possibly all four of the shark's neighbors are fish), his meal is determined at random. Sharks breed analogously to fish. If it is time to breed, and there is space for a new shark to be positioned, then the shark gives birth; if there is no space, the shark waits until there is an adjacent empty square.

## Toroidal Oceans

A torus is a doughnut, or the commonly used screen for video games such as Asteroids. If a fish swims off the east edge, he appears on the west edge on the next turn. If a shark on the south edge eats a fish on the adjacent north edge, he appears at the north edge on the next turn. Note that it is impossible to wrap in two directions on the same turn, since only north-east-south-west movement is possible. (A square has only four neighbors, not eight.)

## Dying Animals

A fish dies if and only if a shark eats him by occupying the same square he occupies. A shark dies if and only if he goes for more than */Number of days a shark has to eat/* without eating any fish. Any animal that dies is removed from the ocean and never seen again.

## Screen

Output should be displayed on the terminal screen once per turn. Also, the world should be shown before any movement begins. There is a screen cluster, called SCREEN.CLU, with SCREEN.TBIN and SCREEN.SPECS located in <SE.LIB>, which I have copied into the project library, <VS.LIB>. Although I did not use the SCREEN cluster in my implementation, feel free to use it in yours.

## I.4 Algorithms

The original *Scientific American* article mentioned maintaining arrays for keeping the fish and sharks in that were of the same dimension as the ocean. The drawback of this approach is that updating the world becomes proportional to the size of the ocean, and not proportional to the number of animals in it. **I want to see algorithms that are *O*(number of animals), not *O*(size of ocean).** If you don't understand what I mean, please ask me.

## I.5 Design

The coding of this project is substantial, possibly more than 8 hours worth. To simplify your work, I have already provided a module dependency graph (see last page). Please use this modular breakdown to solve the problem. Feel free to call your clusters anything you like, but the functionality should be distributed in roughly the same way it is in the MDD.

| | |
|---|---|
| Start–up | This module asks the questions of the user. |
| World | This cluster stores the state information related to the world, and provides a function *simulate* that does most of the work. |
| Ocean | This cluster stores the locations of the animals graphically. It has routines such as *display*, *occupied*, and various inspector routines that return information about the state of the area local to a given square. |
| Animal | This generic cluster can represent both fish and sharks. See above. |
| Prong | This is the random number generator. |
| Direction | This is not a cluster but a data object passed around in Wa-Tor. It is a **oneof** 5 **null**'s, although I did not use the *none* case. I used the **direction** data type to represent relations between two squares of interest. |

## I.6 Coding Style

The goal is to get a working simulator, but not at the expense of crufty code. Assume you are in 6.170, but you don't need to worry about providing the specs of every procedure you write, or doing exhaustive error handling or checking. However, **do** strive for readable style, logical variable and procedure names, no *own* variables, no exposing the rep, etc.

## I.7 Forms

Several forms will be provided tomorrow. They are the **Wa-Tor Administrivia** sheet, on which to put vital information so that you can be paid; the **Wa-Tor Version Track Record**, for reducing the amount of unneeded files sitting around; and the **Wa-Tor Bug Report Sheet. Please do not write your name on the version sheet or on the bug sheet; directory name only.**

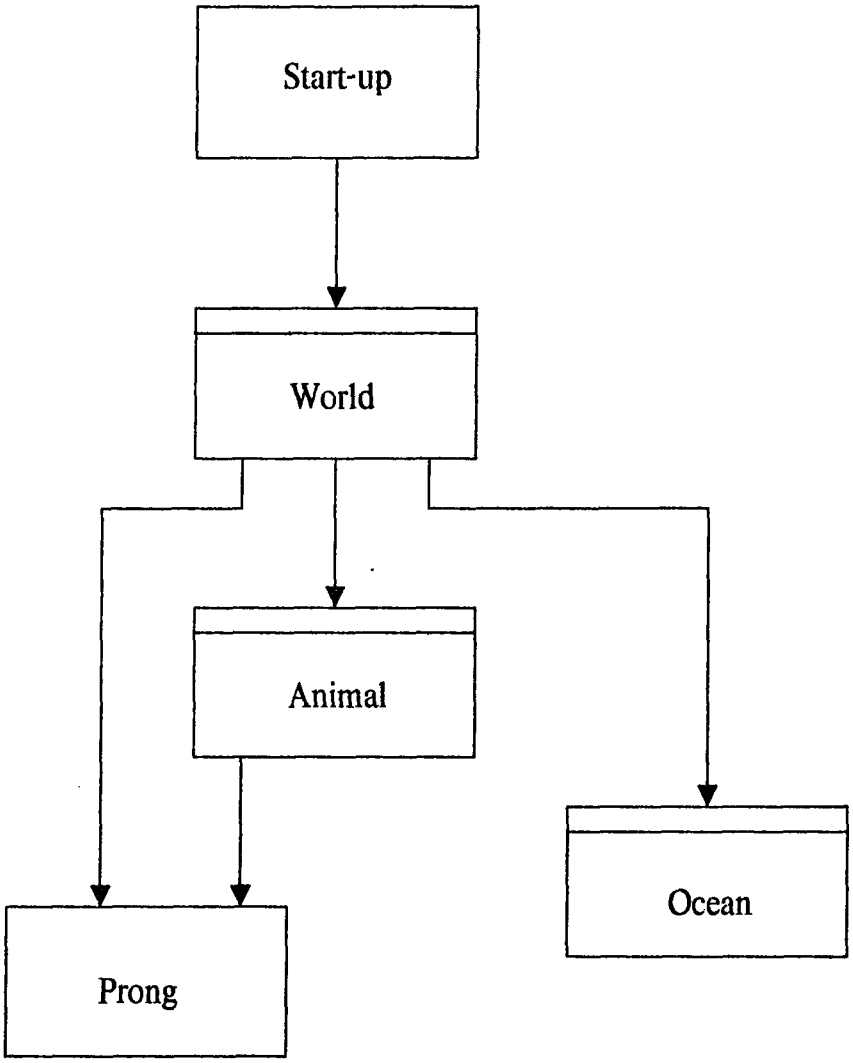
Keep a track record of the versions you write. No file should need more than 60 versions, but you can continue onto another block by saying something like *Filename.clu, c't'd. add 60 to generation #'s.* If a version does not compile, X out the appropriate version number. If you don't compile a given version, say because you leave for lunch without compiling, leave the number unmarked. I will delete versions that are marked as necessary. **Please do NOT delete any files.** Your directories have been set up so that no automatic housekeeping will take place. EE is rich in disk space, so don't sweat the possibility of 45 versions of a file existing. To get the highest versions of files listed, you can say *@dir *.*.0* or *@vdir *.clu.0.*

As you go bug hunting, and you spot a bug and correct it, please report it by entering the file name of the code you changed, the routine name, and if you can, in one or two sentences, describe the error. Ex: *Positioning an animal on an occupied square resulted in an unhandled exception.*

## I.8 Mechanics

Please bring your CLU Manual and any other handouts from 6.170 that were helpful, such as TED manual, TLINK commands, etc. (Also, bring these directions.) There is demo file, WATOR.EXE in the project library, <VS.LIB>, where other useful files reside. The protection of the directory is set so that you can read from the library directory, but the password is *video* just in case you need to connect to it. Participants will be paid an honorarium of $5 per hour up to 8 hours. If you want to work on the simulator more than 8 hours to finish it up, go ahead. The TLINK started on default from inside TED is the one that has \debug as a default (and won't let you make an .EXE file). If you want to make an .EXE file, a different version exists. Ask me where to find it.

## I.9 Module Dependency Graph



# Appendix II

# Wa-Tor Version Track Record

Enter the filename in the space provided. Cross out the version numbers of those files that compile with any errors.

**Your directory name:**

**File Names:** Version Numbers

---

**Filename:**

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 |
| 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |

**Filename:**

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 |
| 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |

# Appendix III

# Wa-Tor Bug Report Sheet

Account name:

**Bug in filename (with version number:)**

Corrected in version number:

Modified procedure name:

**Short description of error:**

**Bug in filename (with version number:)**

Corrected in version number:

Modified procedure name:

**Short description of error:**

# Appendix IV

# Sample Code

The following is one complete set of listings from a successful implementation of the simulator. There are 5 files here:

- Start_up.clu: prompts user for world parameters
- World.clu: data abstraction that represents state of world
- Animal.clu: data abstraction for an animal, either shark or fish
- Ocean.clu: data abstraction for ocean and contents of each square
- Prong.clu: pseudo-random number generator

```
start_up = proc ()

    terms = "\t\n "
    s = stream
    ip  = int$parse

    numsharks, numfish, sbreed, fbreed, starve: int
    width, height, turns: int %the fancy parameters
    pi: s := s$primary_input()
    po: s := s$primary_output()

    begin
        s$putl (po, "Welcome to WA-TOR. We wil simulate a toroidal world with")
        s$putl (po, " sharks and fish, whose breeding patterns and initial")
        s$putl (po, " populations you control. Enter the paramters and then ")
        s$putl (po, " we'll play.")
        s$puts (po, " Initial number of sharks: ")
        numsharks := ip (s$getl (pi))
        s$puts (po, " Initial number of fish: ")
        numfish := ip(s$getl (pi))
        s$puts (po, " Days between shark breeding: ")
        sbreed := ip(s$getl (pi))
        s$puts (po, " Days between fish breeding: ")
        fbreed := ip(s$getl (pi))
        s$puts (po, " Maximum number of days between shark's feeding: ")
        starve := ip(s$getl (pi))
        s$puts (po, " Ocean width [max of 80]: ")
        width := ip(s$getl (pi))
        s$puts (po, " Ocean height [max of 23]: ")
        height := ip (s$getl (pi))
        if width > 80 then width := 80 end
        if height > 23 then height := 23 end
        s$puts (po, " Number of turns for simulation to run: ")
        turns := ip (s$getl (pi))
        end % except others: end
    wator: world := world$create (fbreed, sbreed, starve,
                                  width, height, turns)
    i, j: int
    while i < numfish do
        wator := world$add_fish (wator)
        i := i + 1 end
    while j < numsharks do
        wator := world$add_shark (wator)
        j := j + 1 end
    stream$putl (po, "Initialization complete. Simulation starting")
    world$simulate (wator)
    end start_up
```

Figure IV-1: The file start_up.clu

```
world = cluster is
        create,
        add_fish, add_shark,
        simulate


    rep = record [
              fbreed: int,
              height: int,
              predators:      as,
              prey:   af,
              sbreed: int,
              starve: int,
              terrain:        ocean,
              turns: int,
              width:  int
                  ]

    direction = oneof [north: null,
                       east: null,
                       south: null,
                       west: null,
                       none: null]

    ad = array [direction]
    as = array [shark]
    af = array [fish]
    fish = animal
    shark = animal

    create = proc (fishpreg: int, sharkpreg: int, meal: int,
                   w: int, h: int, n: int) returns (cvt)
        return (rep${
                    width: w,
                    height: h,
                    prey: af$new(),
                    predators: as$new(),
                    fbreed: fishpreg,
                    sbreed: sharkpreg,
                    starve: meal,
                    turns: n,.
                    terrain: ocean$create(w, h)})
        end create

    add_fish = proc (w: cvt) returns (cvt)
        f: fish := fish$create(0, w.fbreed) %1st param 0 b/c fish don't eat
        x, y: int
        x, y := situate (w, f)
        af$addh (w.prey, fish$setxy (x, y, f)) %put the fish into the array
```

**Figure IV-2:** The file world.clu

```
        ocean$store_fish (w.terrain, x, y) %let the ocean know about it
        return (w)
        end add_fish

    add_shark = proc (w: cvt) returns (cvt)
        s: shark := shark$create(w.starve, w.sbreed) %sharks do eat
        x, y: int
        x, y := situate (w, s)
        as$addh (w.predators, shark$setxy (x, y, s)) %put the shark into the array
        ocean$store_shark (w.terrain, x, y)
        return (w)
        end add_shark

    situate = proc (w: rep, a: animal) returns (int, int)
        x, y: int
        nearby: ad
        to_go: direction
        situated: bool := false
        while ~situated do
            x := real$trunc (prong (0) * real$i2r (w.width))
            y := real$trunc (prong (0) * real$i2r (w.height))
            animal$setxy (x, y, a)
            if ocean$occupied (w.terrain, x, y) then
               nearby := ocean$adjacent_empty (w.terrain, x, y)
               if ad$size (nearby) > 0 then
                  to_go := nearby [real$trunc (real$i2r (ad$size (nearby))
                                                  * prong (0))]
                  x, y := animal$getxy (animal$move (a, to_go))
                  x := x // w.width
                  y := y // w.height
                  situated := true
                  end %if adjacent square is empty
               else situated := true
               end %if occupied
            end %while do
        return (x, y)
        end situate

    simulate = proc (w:cvt)
        num_turns: int := 1
        max_turns: int := w.turns
        new_sea: ocean
        baby_fish: fish
        baby_shark: shark
        turns_left_to_eat: int
        where_not_to_go, where_to_eat, where_to_go: ad
        size_to_go, size_to_eat: int
        to_go, to_eat: direction
        must_eat: bool
        can_go: bool
        can_eat: bool
```

Figure IV-2: (Continued)

```
fishx, fishy, sharkx, sharky, newx, newy: int
world_width: int := w.width
world_height: int := w.height
while num_turns < max_turns do
    display (w) %show it before it's modified
    new_sea := ocean$create (world_width, world_height)
    for f:fish in af$elements (w.prey) do
        fish$age1 (f)
        fishx, fishy := fish$getxy (f)
        where_to_go := ocean$adjacent_empty (w.terrain, fishx, fishy)
        where_not_to_go := ocean$adjacent_food (new_sea, fishx, fishy)
        where_to_go := reconcile (where_not_to_go, where_to_go)
        size_to_go := ad$size (where_to_go)
        if size_to_go > 0 then %select a direction
           to_go := where_to_go[real$trunc (real$i2r (size_to_go)
                                              * prong (0))]
           newx, newy := fish$getxy (fish$move (f, to_go))
           newx := newx // world_width
           newy := newy // world_height
           fish$setxy (newx, newy, f)
           %now this fish is in the right range
           ocean$store_fish (new_sea, newx, newy)
           ocean$store_empty (w.terrain, fishx, fishy)
           if fish$getage (f) >= w.fbreed then %time to breed
              baby_fish := fish$create (0, w.fbreed)
              baby_fish := fish$setxy (fishx, fishy, baby_fish)
              %put the kid at old location, since we know it's empty
              af$addh (w.prey, baby_fish) %keep him around
              ocean$store_fish (new_sea, fishx, fishy) %store the kid
              fish$renew (f) %if he just gave birth, renew him
              end %for a new baby fish
           end %if some place to go
        end %for fish
    for s:shark in as$elements (w.predators) do
        %Bookkeeping -- age each one by 1 time unit
        % make him hunger by 1
        shark$age1 (s)
        turns_left_to_eat := shark$hunger1 (s)
        if turns_left_to_eat = 0 then must_eat := true
           else must_eat := false end
        sharkx, sharky := shark$getxy (s)

        where_to_eat := ocean$adjacent_food (new_sea, sharkx, sharky)
        size_to_eat := ad$size (where_to_eat)
        if size_to_eat > 0 then can_eat := true
           else can_eat := false end

        where_to_go := ocean$adjacent_empty (new_sea, sharkx, sharky)
        where_not_to_go := ocean$adjacent_predator
                              (w.terrain, sharkx, sharky)
        where_to_go := reconcile (where_not_to_go, where_to_go)
```

Figure IV-2: (Continued)

```
            size_to_go := ad$size (where_to_go)
            if size_to_go > 0 then can_go := true
               else can_go := false end
            if can_eat cor can_go then %shark can move or eat
               if can_eat then %do eating stuff
                  to_eat := where_to_eat [real$trunc (real$i2r (size_to_eat)
                                                      * prong (0))]

                  newx, newy := shark$getxy (shark$move (s, to_eat))
                  newx := newx // world_width
                  newy := newy // world_height
                  find_food (w, newx, newy)
                  shark$satiate (s, w.starve)
                  else begin %can_go must be true
                           to_go := where_to_go [real$trunc (real$i2r (size_to_go)
                                                      * prong (0))]
                           newx, newy := shark$getxy (shark$move (s, to_go))
                           newx := newx // world_width
                           newy := newy // world_height
                           end %begin
                  end %if can_eat
               %if it moved or ate + moved, then reposition
               shark$setxy (newx, newy, s)
               %check for breeding
               if shark$getage (s) > w.sbreed  then %time to breed
                  baby_shark := shark$create (w.starve, w.sbreed)
                  baby_shark := shark$setxy (sharkx, sharky, baby_shark)
                  as$addh (w.predators, baby_shark)
                  ocean$store_shark (new_sea, sharkx, sharky)
                  shark$renew (s)
                  end %for a new baby shark
               end %if can_eat cor can_go

            %don't need to check for breeding here

            if must_eat cand ~can_eat then shark$die (s)
               else begin ocean$store_shark (new_sea, newx, newy)
                        %transcribe him to new_sea with new (x,y)
                        %and clear out the old location
                        ocean$store_empty (w.terrain, sharkx, sharky)
                        end %begin
               end %to write or not to write
            end
         w := expunge (w) %to clear out dead animals
         w.terrain := new_sea
         num_turns := num_turns + 1
         end
      display (w)
      end simulate
```

Figure IV-2: (Continued)

```
reconcile = proc (occupied: ad, empty: ad) returns (ad)
    if (ad$size (occupied) > 0) cand (ad$size (empty) > 0) then %weed out
       for s_vec: direction in ad$elements (occupied) do
           empty := remove_dir (empty, s_vec)
           end %for s_vec
       end %if
    return (empty)
    end reconcile

remove_dir = proc (compass: ad, vec: direction) returns (ad)
    return_array: ad := ad$create(0)
    for point: direction in ad$elements (compass) do
        if point ~= vec then ad$addh (return_array, point) end
        end %for
    return (return_array)
    end remove_dir

find_food = proc (w: rep, eatx: int, eaty: int) signals (error (string))
    %internal procedure that finds the unlucky fish eaten
    % by reason of its being at (posx, posy)
    alibix, alibiy: int
    for f: fish in af$elements (w.prey) do
        alibix, alibiy := fish$getxy (f)
        if (alibix = eatx) cand (alibiy = eaty) then f := fish$die (f)
                                                    return
           end %if
        end %for loop
    signal error ("Miscoded food search")
    end find_food

expunge = proc (w: rep) returns (rep)
    fsurvivors: af := af$new ()
    ssurvivors: as := af$new ()
    for i: int in int$from_to (af$low (w.prey), af$high (w.prey)) do
        if fish$alive (w.prey[i]) then af$addh (fsurvivors, w.prey[i])
           end
        end %for
    for j: int in int$from_to (as$low (w.predators), as$high (w.predators)) do
        if shark$alive (w.predators[j]) then
           as$addh (ssurvivors, w.predators[j]) end
        end %for
    w.prey := fsurvivors
    w.predators := ssurvivors
    return (w)
    end expunge
```

Figure IV-2: (Continued)

```
display = proc (w: rep) %internal proc
    ocean$display (w.terrain)
    %all that follows is for debugging
    return %leave out the rest of the stuff
    hungry, age, locx, locy: int
    prefix, suffix: string
    for s: shark in as$elements (w.predators) do
        locx, locy := shark$getxy (s)
        age := shark$getage (s)
        hungry := shark$get_hunger (s)
        prefix := "(" || int$unparse (locx) || ", " ||
                  int$unparse (locy) || ") has age "
        suffix := " and has " || int$unparse (hungry) || " turns to eat."
        stream$putl (stream$primary_output(), prefix ||
                                              int$unparse (age) || suffix)
        end
    end display


end world
```

Figure IV-2: (Continued)

```
animal = cluster is create, die, move, age1, renew, alive, setxy, getxy,
                    getage, hunger1, satiate, get_hunger

    direction = oneof [north: null,
                       east: null,
                       south: null,
                       west: null,
                       none: null]

    rep = record [
              age: int,
              alive_p: bool,
              eat: feed_mode,
              xpos: int,
              ypos: int
                  ]

    feed_mode = oneof [pred: int, prey: null]

    fm = feed_mode

    create = proc (time_to_eat: int, time_to_breed: int) returns (cvt)
        % time_to_eat = 0 signifies that this animal does not need to eat.
        % nonzero time_to_eat means that this animal does need to eat.
        need2eat: fm
        if time_to_eat = 0 then need2eat := fm$make_prey (nil)
           else need2eat := fm$make_pred (time_to_eat)
           end
        return (rep${xpos: 0,
                    ypos: 0,
                    eat: need2eat,
                    age: real$trunc (real$i2r (time_to_breed) * prong (0)),
                    alive_p: true})
        end create

    setxy = proc (x: int, y: int, f: cvt) returns (cvt)
        f.xpos := x
        f.ypos := y
        return (f)
        end setxy

    getxy = proc (being: cvt) returns (int, int)
        return (being.xpos, being.ypos)
        end getxy

    die = proc (f: cvt) returns (cvt)
        f.alive_p := false
        return (f)
        end die
```

Figure IV-3: The file animal.clu

```
move = proc (f: cvt, vector: direction) returns (cvt)
    tagcase vector
       tag north: f.ypos := f.ypos - 1 %don't sweat the range
       tag east: f.xpos := f.xpos + 1 % do it later
       tag south: f.ypos := f.ypos + 1
       tag west: f.xpos := f.xpos - 1
       others:
       end
    return (f)
    end move

age1 = proc (f: cvt) returns (cvt)
    f.age := f.age + 1
    return (f)
    end age1

hunger1 = proc (a: cvt) returns (int)
    turns_left: int
    if fm$is_pred (a.eat) then
       turns_left := fm$value_pred (a.eat) - 1
       a.eat := fm$make_pred (turns_left) end
    return (turns_left)
    end hunger1

get_hunger = proc (a: cvt) returns (int)
    if fm$is_pred (a.eat) then return (fm$value_pred (a.eat))
       else return (-1) %signals not a pred
       end
    end get_hunger

satiate = proc (a: cvt, how_long: int) returns (cvt)
    if fm$is_pred (a.eat) then a.eat := fm$make_pred (how_long) end
    return (a)
    end satiate

getage = proc (a: cvt) returns (int)
    return (a.age)
    end getage

renew = proc (f: cvt) returns (cvt)
    f.age := 0
    return (f)
    end renew

alive = proc (f: cvt) returns (bool)
    return (f.alive_p)
    end alive

end animal
```

Figure IV-3: (Continued)

```
ocean = cluster is
        create, %creates an empty ocean
        clear,  %empties an existing ocean
        display,        %displays an ocean

        adjacent_empty, adjacent_food, adjacent_predator,
        % parameters must be in range of ocean
        store_shark, store_fish, store_empty,
        occupied                        % ditto


    rep = array [string]

    direction = oneof [north: null,
                       east: null,
                       south: null,
                       west: null,
                       none: null]

    s = string
    ac = array [char]
    aac = array [ac]
    ad = array [direction]

    open_sea = ' '
    fish_char = 'F' % or some other eye-pleasing characters
    shark_char = 'S' % for fish and sharks
    blank100 = "

    create = proc (xdim: int, ydim: int) returns (cvt)
        % creates an empty array of blanks with given dimensions

        blankline: s := s$substr (blank100, 1, xdim)
        return (rep$fill (0, ydim, blankline))
        end create

    clear = proc (o: cvt) returns (cvt) %empties the ocean of animals
        blanks: s := s$ac2s (ac$fill (1, s$size (o[1]), open_sea))
        for index: int in int$from_to (0, rep$size (o) - 1) do
            o[index] := blanks
            end
        return (o)
        end clear
```

Figure IV-4: The file ocean.clu

```
display = proc (o: cvt)
    po: stream := stream$primary_output ()
    top_size: int := s$size (o[1]) + 2
    top_line: s := s$ac2s (array[char]$fill (1, top_size, '-'))
    clear_vt100: s := "\033H\033J"

    stream$puts_image (po, clear_vt100)
    stream$putl (po, top_line)
    for data_line: s in rep$elements (o) do
        stream$putl (po, "|" || data_line || "|")
        end
    stream$putl (po, top_line)
    end display

adjacent_empty = proc (sea: cvt, x: int, y: int) returns (ad)
    return (adjacent_char (sea, x, y, open_sea))
    end adjacent_empty

adjacent_food = proc (sea: cvt, x: int, y: int) returns (ad)
    return (adjacent_char (sea, x, y, fish_char))
    end adjacent_food

adjacent_predator = proc (sea: cvt, x: int, y: int) returns (ad)
    return (adjacent_char (sea, x, y, shark_char))
    end adjacent_predator

adjacent_char = proc (sea: rep, x:int, y:int, c: char) returns (ad)
    xcur, ycur, wide, high, left1, right1, up1, down1: int
    high := rep$size (sea)
    wide := string$size (sea[1])
    up1 := (y - 1) // high
    down1 := (y + 1) // high
    ycur := y // high

    left1 := x // wide      %same correction as in store_char below
    if left1 = 0 then left1 := wide end
    xcur := (x + 1) // wide
    if xcur = 0 then xcur := wide end
    right1 := (x + 2) // wide
    if right1 = 0 then right1 := wide end

    point: ad := ad$create(0) %start at 0
    if sea[ycur][left1] = c then ad$addh (point, direction$make_west (nil)) end
    if sea[up1][xcur] = c then ad$addh (point, direction$make_north (nil)) end
    if sea[down1][xcur] = c then ad$addh (point, direction$make_south (nil)) end
    if sea[ycur][right1] = c then ad$addh (point, direction$make_east (nil)) end
    return (point)
    end adjacent_char
```

Figure IV-4: (Continued)

```
store_shark = proc (sea: cvt, x: int, y: int) returns (cvt)
    return (store_char (sea, x, y, shark_char))
    end store_shark

store_fish = proc (sea: cvt, x: int, y: int) returns (cvt)
    return (store_char (sea, x, y, fish_char))
    end store_fish

store_empty = proc (sea: cvt, x: int, y: int) returns (cvt)
    return (store_char (sea, x, y, open_sea))
    end store_empty

store_char = proc (sea: rep, x: int, y: int, c: char) returns (rep)
    %While the store_foo operations give arguments in coordinates, the
    % rep has arguments in the same range for the array slots, but the
    % strings are indexed from 1 to /length/, not 0 to /length/ - 1.

    sea[y] := s$substr (sea[y], 1, x) || s$c2s (c)
                || s$rest (sea[y], x + 2)
    return (sea)
    end store_char

occupied = proc (sea: cvt, x: int, y: int) returns (bool)
    if sea[y][x + 1] = open_sea then return (false)
       else return (true) end
    end occupied

end ocean
```

Figure IV-4: (Continued)

```
prong = proc (a: int) returns (real)
    %PRONG is a pseudo-random number generator, for
    % use with the WA-TOR program. It uses a linear
    % congruential formula, and returns a pseudo-random
    % number in the range 0 to 999.

    own last: int := 1

    modulo: int := 1000
    lin: int := 21
    delta: int := 37


    new_val: int := (lin * last + delta) // modulo
    last := new_val
    return (real$i2r (new_val) / real$i2r (modulo)) % to put in range 0 to 1-
    end prong
```

Figure IV-5: The file prong.clu