Instilling a Defect Prevention Philosophy

Laurie Williams
Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract *- A hundred years ago, people faced problem in using the new typewriters . . . Typing was error prone . . .Correcting a character, even a word, might not be so bad. Correcting a missed sentence, early on the typed page, was better fixed by starting the page over.*

With this background for almost a human generation of using typewriters, the new idea of touch-typing, typing without looking at the keys, was a very strange one. "That's silly. Who could possibly do that?" . . .

Of course, we know how touch-typing turned out as an internationally useful method . . . ***But we also know that very few key-looking typists learned how to touch type. It was the young generation coming into the field.***

There is a lesson in touch typing for software development. In software, teaching a programming language and how to compile and execute programs allows people to write programs immediately. Very likely, such programs will require considerable debugging. . . errors and unit debugging are just an expected and integral part of programming.

However, people with the right education and training need not unit debug their software any more than people need to look at the keys when they type . . . Yet, just as in touch-typing, serious programming begins only with formal methods, more explicit design, and verification from specifications.[1].

Introduction

Undoubtedly, the most efficient way to produce quality software is to prevent the injection of defects in the first place. Published data indicates that the cost of repairing a software design error can be a hundred times more costly when the design error is found after software delivery than when found during software design. [2] A proven software development process, Cleanroom Software Engineering, adds engineering rigor to the process and focuses on defect prevention and statistical quality control. Cleanroom has historically produced software with significantly superior quality and improved productivity. As a result, the University of Utah offers a new undergraduate course in Cleanroom Software Engineering to teach the students to "touch-type before they are too used to looking at the keys." The course instills a defect-prevention, systematic/engineering philosophy in these students. The only prerequisites to the course are CS1/CS2. The course was offered for the first time in Autumn 1997 and consisted of 29 students, spread fairly evenly between sophomores, juniors and seniors. Students were allowed to choose between C++ and Java for program development.

The course requires students to develop using some radically different techniques. For example, the students work in groups to specify, design and develop. They inspect and verify each other's code. But, they **do not** compile their own code prior to testing. Another group in the class acts as a "test group" for their code and is the first to compile and test it. "This has an immediate psychological effect on the developers, which translates into a quality improvement for their software. Knowing that all execution errors will be given public scrutiny leads to more conservative designs which exactly match the requirements, use solutions with high confidence of working, and are prepared with more care." [1] Each group functionally or black box tests another group's code. The high degree of confidence in the verification process done before compilation eliminates the need for any white box/unit testing.

This paper will contain some introductory information on Cleanroom Software Engineering and will explain how its principles are taught and enforced in the classroom.

Two Guiding Principles

Cleanroom Software Engineering was developed by the late Harlan D. Mills at IBM. The name is taken from the clean rooms, which are used in semiconductor manufacturing. There, a scrupulously dust-free environment is maintained because it is much more expensive to remove defects from the chips after fabrication than to prevent their introduction during the process. [3] Mills felt these same techniques should apply to software. He, therefore, developed a process that applied engineering rigor to the software development process. His methods are based on two principles: ***programs are rules for mathematical functions*** and ***software testing is sampling.*** The five core Cleanroom techniques that support these two principles are discussed below. You will notice that most of these steps, taken individually, are not unique to Cleanroom Software Engineering. Incorporated as a group, they comprise a unique process.

0-7803-4762-5/98/$10.00 ©1998 IEEE

98CH36214

Formal specification/Design of Intended Behavior

Perhaps the most important step is the development of a formal, implementation-free specification that completely, consistently and correctly describes the desired system. Cleanroom uses the box structure methodology for producing a formal specification, in which a hierarchy of three types of box structures can rigorously describe the behavior of a system. The *black box,* developed in the specification phase, gives an external view – user inputs and outputs, and hides all of the objects implementation details, including any data implementation or processing. The *state box*view (a finite state machine) partially exposes the data implementation while continuing to hide procedures. The *clear box* view exposes procedures. The last two views are developed in the design phase and may include reference to new black boxes in an iterative, functional decomposition process.

The black box specification phase uses the perspective that ***programs are rules for mathematical functions*** by assigning a response for each stimulus history (See figure 1). (A *stimulus* is any user-originated event, which may affect the future behavior of the system. A *response* is system behavior which is observable by a user.) A black box is *complete* if it gives a response for every stimulus history. A black box is *consistent* if the response is unique. [4] It is *correct* if the response is the one-and-only correct response. A black box must satisfy all three criteria in the course of verification.

Figure 1: Black Box

The students used the ToolSET_Specify tool developed by Software Engineering Technology to develop their black box and state boxes. The tool is based on "sequenced based specification" [6] where sequences of stimuli represent all stimuli received (up to and including the most recent stimulus) by the software system. The students use the tool to generate every possible sequence of stimuli. For each they must map the sequence to the response the system would give to the user for that particular stimuli sequence. The system guides the students in identifying the canonical stimuli sequences, which represent the valid stimuli sequences, each with a unique state/future behavior. (The other sequences are "dispositioned" as illegal sequences or equivalent in future behavior to a previously analyzed stimuli sequence.) These canonical sequences are analyzed to develop the black box and state boxes.

The development of black and state boxes is beneficial in the development of "objects" in the object-oriented sense. The black box analysis helps identify the class methods needed to appropriately change the object's state and to issue a response to the user, given the object's current state and most recent stimuli. The state box analysis helps determine the minimum set of class data variables that will be necessary to hold the state of the object.

Incremental Development Process Model

Cleanroom professes the use of an incremental development process in order to manage the risk of developing large systems. Each increment must be externally executable and be developed and tested as a separate product. First, increment one is developed and tested. Then, increment two is developed, integrated with increment one, and tested. Development proceeds with a pipeline of increments, each accumulating all the prior increments until the product is complete. Each increment can be delivered to a customer for testing and feedback.

The students are taught about incremental development and its advantages. They were given a group project which consisted of developing a telephone system, with an operator and subscribers. The subscribers must have basic service, but also may have added features such as call waiting, call forwarding, conference calling, last number redial, and call transferring. When the students began to be overwhelmed with all the permutations and combinations of stimuli sequences, they were encouraged to develop an increment plan – beginning first with a basic telephone system with no features. This allowed them to keep the project within their "intellectual control" and complaints of being overwhelmed with the project ceased.

Stepwise Refinement of Specifications to Code

With the help of the toolSET_Specify tool, the students iteratively developed their black boxes and state boxes. Developing the stimuli sequences might identify the need to abstract a certain stimuli into another black box, which would then have to be analyzed to determine its state box. Using stepwise refinement, each of these black boxes/state boxes is used to develop clear boxes, which contain the algorithms necessary to change the state and issue the appropriate external response.

The use of orthodox structured programming[7] is essential to Cleanroom Software Engineering. Structured programming involves developing code using disciplined control structures and disciplined data structures. Only seven disciplined control constructs, called *prime program structures*, are allowed – sequence, if-then, if-then-else, while, do-while, for loop, and case/switch. These structures are nested over and over again in a hierarchical structure. Additionally, these structures may only have single-entry and single-exit paths (eliminating the use of break, continue and the like) that cannot produce side effects in control flow. These restrictions result in code that is far easier to

verify and calm the "control flow jungle"[5]. The reasoning will be later discussed further in the verification section. Students were limited to these disciplined control constructs in their code development.

Students must also use disciplined data structures in design and code development. "Undisciplined data structures – such as arrays and pointers – lead to more complex designs that are harder to verify. These data structures allow random access to data, just as gotos provide random access to instructions."[4] Any aggregate data structure, such as stacks, queues, arrays, must be implemented as *anonymous data* which allow access only to selected data elements. Since the students used an object oriented language, this meant implementing the aggregate data as a class, with class methods used for data access and update. Then, these methods could be verified once to ensure any data manipulation, which must use the class methods, would be done properly.

Correctness Verification of Developed Code

Intended Functions

The proof of correctness is a derivation process whereby the design's functionality is compared against its *intended function*, which is defined during clear box design. Intended functions are statements of how the values of state variables change because of executing a code segment. Intended functions are written as comments and remain with the code as it is written in its structured programming control structure form. By convention, intended functions are enclosed in brackets. Some examples:

```
/* [ m :=1
  : n := 0 ]  */
m = 1;
n = 0;
```

This is the simplest intended function. It says that in the new state, the value of m is 1 and the value of n is 0 and, by implication, all other state variables are unchanged. In the new state, each variable on the left-hand side of the ":=" is assigned the value in the right hand side.

```
/*  [ m := n
   : n := m  ]  */
```

This is an intended function for the swap function. Again, the values on the right side are in the new state, the values/variables on the left side are in the current state. Intended functions are also called *concurrent assignments*, suggesting that the variables are all receiving their new values concurrently.

```
/* [ n > 0 → avg := sum/n
  | true → I          ]  */
```

This is a conditional concurrent assignment. In the current state, the preconditions (on the left side of "→" are evaluated in the order in which they appear. The first to evaluate to true is selected to determine the next state, using the statement on the right hand side. The above code sets the new value of avg to be sum/n if n > 0. If n is not >0, the second condition automatically evaluates to true. true → I means that the identify function should be applied (no state variables should change).

```
// [*this → garbage ]
```

This is a typical intended function for a class destructor. It implies that after the function has executed *this could point to anything and should not be used.

These intended functions add a mathematical precision to commenting. They are very useful in the verification process in which the intended function and the code are compared to ensure they execute the same function. Intended functions are also mapped against the design to ensure the code is doing what is outlined in the design. If the intended function does not match the design, then the design or the intended function/code must be modified and reviewed again. The students were not enthusiastic about using intended functions, a natural reaction to adding formality to their process. Several did admit they used some sort of intended functions or at least thought in "intended function terms" in their other classes after learning them.

Correctness Conditions and the Axiom of Replacement

The use of structured programming and intended functions greatly aids in the verification process. As discussed above, the code is limited to a set of specified control structures. Each of these control structures has a correctness condition, which is used for verification purposes. See Appendix A for the correctness conditions for each of the control structures [5].

Once a particular control structure is verified, it can be incorporated into a hierarchical structure without any further verification. The "axiom of replacement" then allows the substitution of the intended function for the control structure in any further verification. The substitution localizes later verification to the control structure at hand and assumes the correctness of imbedded structures.

Each student group performs multiple verification reviews of each other's work products. These reviews would consist of incrementally verifying for consistency, correctness and completion: the specification vs. the customer requirements, the design vs. the specification, the intended functions vs. the design, and the code vs. the intended function.

Statistical Certification of Compiled Software Products

Certification supports the Cleanroom tenet *testing is sampling.* Like the Cleanroom name itself, statistical certification has roots in manufacturing. For example, at the end of an automobile assembly line, a quality control employee randomly tests various aspects of the finished products. If a particular defect recurs at a statistically significant rate, the manufacturing process is examined for the cause of the defect and appropriate step is corrected.

Likewise, the statistical certification involves randomly selecting and applying functional/black box tests to the software. The focus of the testing is on the external system behavior – the way users tend to use it. "Then, the goal for measuring software quality is to establish the probability that a software system will not fail while it is in use."[4] In other words, the testing is used to calculate a Mean Time to Failure, MTTF.

Traditionally, the purpose of software testing is to remove bugs. However, because of the defect prevention rigor in Cleanroom Software Engineering, the testers are not overwhelmed with defects and are able to focus on scientifically certifying the software's quality and on continuous process improvement. Finding failures also occurs, but only as a (desirable) side effect. "The important attitude for Cleanroom practitioners to develop is that defect prevention in design allows testing to be conducted more systematically and scientifically."[4]

The class involves two weeks of instruction on black box testing techniques. However as discussed above, each student group does not *compile or test* their own code. When the groups have completed code development and verified their code, another student group black box tests their code. The two groups work together to get the code to compile. Then, "test group" is given access to the "development group's" executables but does not have the ability to look at the other group's code. They send "problem report" email messages to the "developers" who may correct errors and recompile their code.

Unfortunately, in an undergraduate quarter-length class, statistical testing can not be thoroughly covered. The students learn the new skill of black box testing and how it is used in statistical testing. They must document the black box test cases they will perform and show that these black box test cases cover the most common uses of the telephone system. However, time does not allow in-depth coverage of statistical evaluation of the program defects.

Summary

Addison-Wesley will be publishing an undergraduate Cleanroom textbook in Autumn 1998, *Toward Zero-Defect Programming.* Several chapters of this book were successfully used, in manuscript form, for the Autumn 1997 class.

Instructing students in the Cleanroom Software Engineering process imparts a radical departure from the students' previous software development practices. As a whole, Cleanroom imparts "formality" to a degree that is "digestable" to students. The students reject or accept these radical changes in varying degrees. Many students have reported that they began using the techniques in their other classes immediately. Some will follow Cleanroom precisely in their future development; some will not. However, the course has undoubtedly changed their perspective on software development for correctness and their awareness of its importance.

Bibliography

[1] Dyer, M., *The Cleanroom Approach to Quality Software Development,* John Wiley, New York, 1992.

[2] Boehm, B.W., *Software Engineering Economics,* Prentice-Hall, 1980.

[3] Deck, M., *Cleanroom and Object-Oriented Software Engineering: A Unique Synergy,* Cleanroom Software Engineering, Inc. 1996.

[4] Becker, S. A. and Whittaker, J., *Cleanroom Software Engineering Practices,* Idea Group Publishing, 1997.

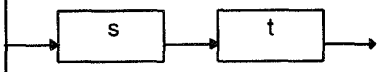
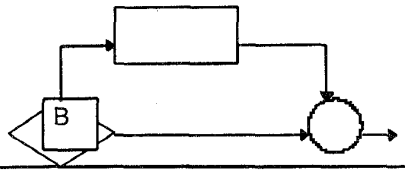
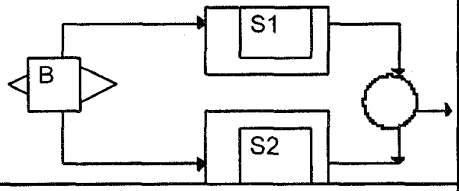
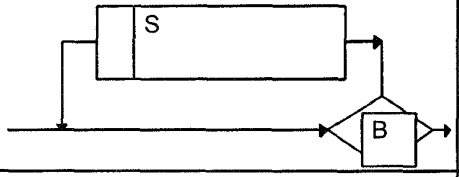
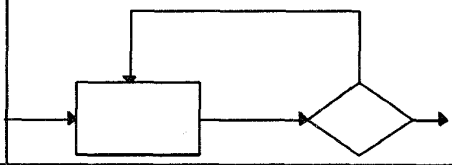
[5] Poore, J. H. and Trammel, C.J., *Cleanroom Software Engineering: A Reader,* Blackwell Publishers, 1996.

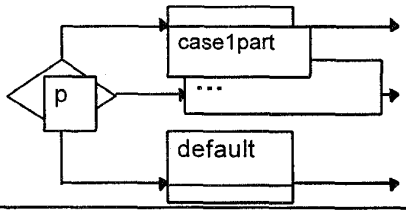
[6] Prowell, Stacy J. *Sequence-Based Software Specification,* Ph.D. Dissertation, Department of Computer Science, University of Tennessee, 1996.

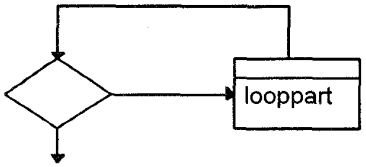
[7] Linger, R.C., Mills, H.D. and Witt, B.I., *Structured Programming: Theory and Practice,* Addison Wesley, 1979.

[8] Rosen, Steven J., *Design Languages for Cleanroom Software Engineering,* Proceedings of the Hawaii International Conference on System Sciences, 1992

Appendix A: Disciplined Control Structures

Control Structure	Flowgraph	Code	Correctness Conditions/ Verification Questions
Sequence		[f] s; t;	Does doing s followed by doing t do f? (Do the trace table)
If-then		[f] if B S1;	When B is true, does doing S1 do f? When B is false, does doing nothing do f? (Do the trace table)
If-then-else		[f] if B S1; Else S2;	When B is true, does doing S1 do f? When B is false, does doing S2 do F? (Do the trace table)
While		[f] while B do S	Is loop termination guaranteed? When B is false, does doing nothing do f? When B is true, does doing S followed by doing f do f?
Do-while		[f] do S While (B)	Is loop termination guaranteed? After doing S, if B is false, have you done f? After doing S, if B is true and you now compute f, has the whole computation done f?

Case/switch		[f]
		switch(p) {
		case 1: case1part;
		break;
		. . .
		default: defaultpart;
		}
			When p = case 1, does case1part do f?
			. . .
			When p != (case1 . . . casen) does defaultpart do f?

For		[f] for (indexlist) looppart;	Does loop terminate? Does firstpart, followed by secondpart, . . . followed by lastpart, do f?
Recursion			Is loop termination guaranteed? Perform verification on disciplined control structures in routine.