A Case Study in Software Testing

by

Katherine Diane Franklin
B.A., Queen's University, 1986
B.Sc., Queen's University, 1988

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]
Dr. D.M. Hoffman

[redacted]
Dr. M. Van Emden

[redacted]
Dr. A. Astbury

[redacted]
Dr. J. Moehr

ISBN 0-315-62633-X

Canada

Supervisor: Dr. Daniel R. Hoffman

Abstract

Testing an entire software system or some part of one helps ensure that the software behaves as the designer intended; testing helps increase software quality. Testing falls into the classical cost/benefit tradeoff problem; testing software is expensive. The testing process is not well understood by many software developers. Due to the high costs and lack of understanding, testing is being performed inefficiently, ineffectively, or not at all.

We examine the characteristics of good testing and the issues that make testing expensive. We present the steps in the testing process and describe a case study in which we test two systems and two of their modules. We tested a stack based calculator, *scalc*, and a symbol table module, *symtbl*, from the *BB* software system, a demostrational system used for teaching software engineering. We tested *caseTool*, a CASE tool developed by ABC Inc. of Vancouver, and *caseTool*s memory management module, *Space*. We maintained careful records of the costs we incurred and we discuss the effectiveness of the testing developed.

Through our records, we demonstrate the need to understand the testing process, the roles people play in the process, and the relationship to the software development process. We have discovered the importance of effective planning in reducing the costs of testing, and in turn, reducing costs in developing software.

Examiners:

[redacted]

Dr. D.R. Hoffman

[redacted]

Dr. M. Van Emden

[redacted]

Dr. A. Astbury

[redacted]

Dr. J. Moehr

Contents

List of Figures

Acknowledgements

This thesis could not have been completed without the guidance of my supervisor, Dr. Daniel Hoffman. In particular, I would like to thank him for all that he has taught me and for his patience and perseverance.

My thanks also go to the people at ABC Inc, for their time, assistance and product.

Eric Davies, as my 'peer member', provided very helpful proofreading.

My family has given me much support over the years and deserve particular thanks for their encouragement on this thesis. For his support, thank you Bill.

Chapter I

Introduction

Testing software can help produce more reliable software and increase software quality. Software reliability and quality are increasingly important in our society as computers are becoming more widely used in many different industries. Testing software helps ensure that software behaves correctly. Finding and correcting faults through testing reduces the costs of software development and maintenance. However, testing is expensive. Tests must be individually developed, executed and maintained for each piece of software, as all software is different. This cost leads us to a classical cost/benefit tradeoff problem; cost versus software quality.

The testing process is misunderstood by many software developers. Some developers think that testing is only done during software development and throw it away after they believe they are done with it. This leads to even higher costs as the developers need to redevelop testing during software maintenance; often redeveloping testing is too expensive and developers will not do it. Other developers confuse the steps in the testing process, resulting in poorly planned and inefficient testing. The developers try to make their testing do too much or not enough. They may use tools that do part of the job but they use the tools incorrectly.

Due to the expense of testing and the developers' lack of understanding, testing is being performed inefficiently, ineffectively, or not at all. Inefficient testing is a waste of resources. Ineffective testing is not only a waste of resources but can lead to a false sense of security about the software's reliability. All of this can lead a software developer to the false conclusion that testing is not beneficial and the developers abandon testing as a development tool; few developers bother to replace it.

Last spring, ABC Inc. of Vancouver approached us about developing software engineering tools that would be useful as part the CASE (Computer Aided Software Engineering) tool they were developing called *CaseTool*. Through discussions with them we discovered that they knew very little about the testing process and how it related to software development. They did very little testing, applied no standard testing process and did not see large benefits to be gained by changing things. They were sceptical that costs could be reasonably controlled and still see benefits. We undertook testing of *CaseTool* to show that costs can be controlled without losing the benefits of testing.

In this thesis, we define the testing process and place it in perspective to the entire software development process. We examine the characteristics of good testing and the limitations that give us the benefits of testing but make testing expensive.

We describe a case study in which we test two software *systems* and two of their *modules*. A system is a single piece of software that is available for use, and a module is a piece of that software; we will discuss these in further detail in Chapter II. We

tested a stack based calculator, *Scalc*, and a symbol table module, *symtbl*, from the *BB* software system, a demonstration system used for teaching software engineering. Then, we tested the *CaseTool* system and *CaseTool's* memory management tool. *CaseTool* is a very large system and, for reasons we will discuss later, our first attempt at testing *CaseTool*, Scheme One, did not produce efficient testing. We recognised the problems that caused the high costs, and developed a second scheme, Scheme Two, that addresses some of these problems. We developed a scheme for testing call-based modules and applied it to *symtbl* and *Space*.

We maintained careful records of the costs we incurred. We describe these results and use them to demonstrate the effectiveness of testing. We discovered the importance of maintaining a perspective of the role testing has in the software development process and of the role that people play in the process. We emphasise the importance of effective planning in reducing the costs of testing and in turn, the costs of developing software.

Building Blocks

Building Blocks (BB) is a demonstration software system running under UNIX and is used at the University of Victoria for teaching and developing software engineering techniques. *BB* contains two application systems: *Scalc*, an interactive stack based calculator; and *Reachset*, a batch system that calculates the reachability set of a graph. In order to build these systems, there are several service

modules: *token*, a scanner module; *symtbl*, a symbol table module; *stack*, a stack module; and *spath*, a shortest path module. The *BB* software system was re-implemented, in the same environment as *CaseTool*, for the Macintosh using MPW Object Pascal. The *Scalc* and *Reachset* systems are implemented as MPW tools as opposed to Macintosh applications, and must be executed in the Macintosh Programming Workshop(MPW) environment. Each service module is implemented in two Pascal units. Firstly, the interface is implemented as a Pascal unit of the same name; e.g. the *symtbl* module is implemented as the *symtbl* unit. The second unit implements the exception signalling procedures in the Pascal unit with the *Exc* extension; e.g the *symtblExc* unit implements *symtbl* exceptions handling.

As preparation for testing *CaseTool*, we developed testing for the *Scalc* system and the *symtbl* module. *Scalc* is 300 lines of Pascal code and *symtbl* is 200 lines of Pascal code. The advantage of first performing test experiments on *BB* modules is that *BB* was developed using a disciplined software development process [15]. The modules are well defined, and manageable. However, testing *BB* is limited in its usefulness as *BB* has been implemented on the Macintosh as a collection of MPW tools and does not have the added complexities of a Macintosh application such as the human interface and graphics.

CaseTool

CaseTool is a CASE tool for the Macintosh computer currently under development by ABC Inc. of Vancouver, British Columbia. *CaseTool* accepts Pascal source code as input, provides a set of tools for analysing the source code and produces answers to specific questions about the relationships between identifiers. *CaseTool* builds an abstract syntax tree from the source code and uses this tree to provide answers to queries. A hypertext feature supports on-line documentation and allows the user to build composite reports of multiple queries. *CaseTool* takes advantage of the interactive graphics interface of the Macintosh to present the information visually to the user [11]. *CaseTool* is being developed using the MPW Object Pascal and contains approximately 75 000 lines of code.

We have developed and performed system testing for *CaseTool*. We have also developed and performed module testing for *CaseTool's* memory management module, *Space* which is 200 lines of code.

ABC's emphasis on the interactive interface and graphics to reveal information to the user allows the test developer to take advantage of the testers' abilities to process visual information quickly. This advantage has a drawback: it makes manual testing expensive and automatic testing difficult to develop.

Furthermore, *CaseTool* was not developed with any particular software development process. The lack of requirements definition and interface specifications for *CaseTool* makes testing difficult; we have no clear idea of the designers' intentions. The developers

change their minds frequently on the requirements of the human interface. This development process caused instability in the human interface and the *CaseTool* source code.

Equipment

Unless otherwise specified, all work that we did was done on the Macintosh SE in the University of Victoria Software Testing Lab. This machine is configured with 2.5 Mb RAM and a 20 Mb hard disk with MPW, *BB* and *CaseTool* installed.

Thesis Overview

This thesis contains a discussion of terminology and fundamental issues (Chapter II), and previous and related work (Chapter III). We present several case studies in testing. In Chapter IV, we discuss our experiences system testing *Scalc* and *CaseTool*. Chapter V contains a discussion of module testing *symtbl* and *Space*. Our conclusions and future directions for research are discussed in Chapter VI.

Chapter II

Key Concepts and Terminology

Software Development Process

The steps in the software development process that we use as a framework are described in detail in [15]. These steps can occur in sequence or parallel, and are repeated until satisfactory results are achieved. The seven phases are:

1. Develop requirements analysis
2. Decompose system into modules
3. Design module interfaces
4. Design internal structure of modules
5. Code module implementations
6. Test

The first phase is to define the requirements of the system under development. The *requirements document* defines what the system must do without mention of the internal system construction.

Module decomposition divides the system into individual work assignments called *modules*. Each module encapsulates a design decision which is hidden from the rest of the system; this criteria for module decomposition is called *information hiding*. A *module guide* describes each module by briefly stating the service it offers and the design decision it encapsulates [15].

A *module interface* for each module provides a black box description of the module by completely describing the set of assumptions that programmers using the module are permitted to make about its behaviour. Module interfaces define the service that each module provides but not how the service is performed.

The internal structure of each module is designed independently. The module specifications allow the implementation of any module to use the services of any other module in the system.

Writing the program consists of implementing the modules and building the system up in the layers laid out in the uses hierarchy.

Maintaining the software requires repeating these steps throughout the life of the system. Requirements change as the product is used and depending on the extent and complexity of the change requested, any or all of the steps may have to be repeated or have their documentation updated.

Testing and the Software Development Process

Although Parnas lays out the software development process quite clearly [15], he does not detail how each stage should be accomplished. In particular, he does not show where testing fits into this software development process.

There are three types of testing:

1. module testing
2. system testing
3. regression testing

Module testing ensures that the module performs the services described in its interface. *System testing* ensures that the system performs all and only the services described in the requirements document.

Regression testing ensures that the changes made to the code of a module do not adversely affect the module or the system. During system maintenance, it is important to ensure that changes are performed accurately, and that they do not affect the existing system beyond the documented and desired effects. Changes affect documentation, source code and testing. The testing for each module affected by the change should be updated, as well as the testing for the system, to reflect the changes in the requirements and module interfaces. Any module modified during a change must pass the updated testing. The system must pass the updated system testing.

While module testing and system testing is performed during the writing of the software, regression testing is performed during maintenance. Due to changing requirements and specifications, regression testing needs to be performed often and must be easy to update. Regression testing must be efficient to execute and maintain.

It is inefficient to develop two sets of tests: one for writing the initial code and one for maintenance. Instead, good module and system testing can be used to perform the regression testing. Test developers should keep regression testing in mind while developing the module and system testing.

Aim of Testing

The princpal aim of testing is to increase software quality: how accurately the software system meets the designers' intentions. The amount of testing that is performed is limited by the high cost of the testing process. At each stage of testing, cost and difficulty depend on the item being tested. These costs might not be economical if the tests yield insufficient information about the quality of the software system.

Many testing methodologies address only some aspect of testing. They rarely look at the costs of applying their testing to different types of software and the costs of developing and maintaining these testing mechanisms.

Characteristics of a Good Testing Methodology

The tests should be cost effective to execute, maintain, as well as develop. This allows the testing to be re-used during the maintenance phase.

Automation of testing reduces the costs of human labour in the testing process by making it possible to cheaply re-execute the same tests. Automation also reduces the likelihood of human error in the testing process.

Good testing effectively isolates the item being tested. Isolating the item reduces the amount of source code being tested; this helps the testers determine the cause of failures in the code.

Limitations of Testing

A *failure* occurs when the results of a test are not as expected: testing has located a failure of the code being tested to behave as expected. A *fault* is the error in the code that causes the unexpected behaviour. Testing can only find failures. Testing tells you *when* the software being tested does not perform as expected, but it can not tell you *what* is causing the fault. However, by isolating the software, the tester can reduce the amount of source code to be searched for the fault. Control over the software allows the test developer to tailor test cases to further isolate the amount of source code in the software that is executed.

Controllability refers to the ease with which inputs can be submitted to the software under test. *Observability* refers to the ease with which the behaviour of the software under test can be observed. Controllability and observability affect the amount of automation that can be performed in the testing. The more difficult it is to control and observe the software the more expensive it is to use automated tools to control and observe the software; existing tools may not apply and the test developer may have to develop new tools to perform the tasks. These new tools may make it inefficient to do as much testing as desired because of the expense of developing and maintaining these custom tools. If the custom tools are too expensive, then the cost may exceed the benefits of automation and the use of automation must be reduced or eliminated.

Testing Process

Sometimes the test developers are not the people that perform the testing and even if they are the same people, the test developer must resolve several issues:

1. Who performs the testing?
2. What knowledge do they possess?
3. How often is testing performed?
4. How long should the testing take?

The test developers need to know who will perform the testing and how much knowledge the testers have prior to the testing in order to develop testing that is easy for the testers to understand and perform efficiently. Testing that is too difficult to execute is not performed. Similarly, if testing is too simple for the testers then they lose interest and they miss subtle errors.

Time estimates allow the developer to create reasonable testing schedules, and prioritise and focus the testing. Depending on priorities and time, the testing may be broken up into parts with each part executed on different schedules. Making these decisions gives the testers a more accurate view of their responsibilities; how they can schedule and perform the tests, and how they can provide feedback to the test developers.

Like the software development process, testing is performed by iterating through a set of steps [7]:

1. Develop a test plan
2. Build the test harnesses
3. Choose the inputs
4. Determine the expected outputs
5. Execute the tests
6. Compare the expected and actual outputs
7. Evaluate the test results

A *test plan* lays the ground work for the entire testing process; it identifies what is being tested and describes how the testing is implemented. A test plan identifies the *critical state values* of the software and sketches an *strategy* for test case selection. Critical state values are a subset of all possible state values of the software and typically focus on boundary cases as determined by the specifications. The test plan also sketches an outline of the test *implementation*, including how the *drivers*, *stubs* and remaining items in the *test harness* are used to execute the test (described below). We try to keep the test plan as concise as possible since its primary goals are to guide and document the testing process, and not to specify individual test cases.

A test harness is the code, documents, and data necessary to execute a test plan. Building the test harness involves using the test plan and then developing the drivers and stubs to control the testing. A driver stimulates and controls the software under test; it typically requests actions that affect the software's state or behaviour and observes the actual behaviour. A stub provides services to the software under test.

Using drivers and stubs in place of the actual services addresses the issues of controllability and observability. A module is much easier to control and observe if a driver is used in place of

the entire system and the system is much easier to control and observe if it can be controlled automatically and separately from outside interference.

A *test case* consists of an *input* and *expected output* pair. An input is a stimulus to the software under test, designed to exercise some aspect of the software's behaviour. By using the test plan's test case selection strategy, the test developer develops the inputs that are applied against the testing software. *Output* is the observable behaviour of software as the result of a stimulus to the software. The expected output of the software is the result that the tester expects to observe as the result of a specific test input. The test developer must develop the expected outputs for each of the inputs. If a source for generating the expected results (a *test oracle*) is not available, and it typically is not, then the expected results must be generated by the test developer or tester using the specifications.

Once all the preparation has been done, the tester executes the test. For each test case, the tester stimulates the software under test using the input and recording the actual observable behaviour of the software (*actual output*).

The expected output and the actual output are compared. The test case has succeeded if the actual output of the software under test correctly corresponds to the expected output; otherwise the software has failed that particular test case. Test results are recorded noting the test cases passed and failed, and sometimes a numerical evaluation of these figures as a ratio against the total number of test cases.

It is not enough to just perform the test suite and report test results. It is necessary to evaluate the entire testing procedure to ensure that the goals of the testing process were achieved. When the test cases find failures the test developer must consider whether the test case failed because there was an error in the expected output or if there was an error in the test software. Sometimes, the test case fails because the test developer did not accurately determine the expected behaviour of the software. Nearly always failures are found. Evaluation determines whether there are too many failures; this varies from software to software and depends on such issues as the type of failures, and the acceptable number of failures for the software.

Even when the testing does not find any failures it is necessary to evaluate the entire testing procedure to ensure that the goals set for the testing are achieved and that the testing meets the characteristics of good testing as much as possible. From running the testing, it may be possible to see where improvements can be made to perform these tasks better or using more automation to perform the tasks. Each of the phases must be checked for accuracy, completeness and efficiency.

Testing Tools

Tools that allow for keystroke or mouse-use recording and playback are useful for recording the test input. These tools perform like conventional audio cassette tape recorders and record the actions

that are made on the computer through the user interface. When requested, the computer is capable of "playing back" this recording, using the record of actions in place of actual human interface actions. These tools are useful in recording the input for a test case.

Screen capture utilities allow you to store a screen image precisely on disk. If the observable behaviour is visual, these tools are useful for recording expected and actual output. File comparison tools are useful for comparing screen capture files representing the expected and actual output and determining if the test case has succeeded or failed.

Some tools are capable of generating random keystroke and mouse-use; these tools may be useful in different types of testing processes.

A test coverage tool would also be useful. Structure coverage describes the number of structures that exist in the software's source code and the number of times the structures are referenced during execution of the software. Test coverage typically refers to the number of lines that executed as a ratio of the number of lines in the code. This allows us some quantative estimate for the effectiveness of the testing; ideally all lines of code should be executed at least once. Unfortunately, we could not find any test coverage tools available for the Macintosh.

Chapter III

Background

In [9], Hoffman performed a case study of nineteen C modules in the GRADES system that involved dealing with standard call-based input and output modules, keyboard input modules and screen output modules. He demonstrated that modules present a variety of controllability and observability problems. In special cases, like keyboard input and screen output modules, automation may not provide assistance for testing. Due to controllability and observability problems, testing procedures must be evaluated for each item under test.

Brown and Hoffman [7] developed testing for a module which is difficult to control and observe. They describe the principles of testing including systematic maintainable testing, interface-based testing, isolation of the item under test, and the use of automation to reduce costs. They list a set of steps in the testing process, which we expanded on in Chapter II. They have an interactive and batch test program for their module under test, and present the costs and benefits of their testing.

In [18], Uren, Miller and Irwin examined several instances of using automation for executing tests. These examples had different test case selection strategies, and automation helped achieve their clients' goals in detecting failures and having procedures for

detecting failures. For these case studies, they list various statistics: numbers of programs written, tests conducted, functions tested, errors found, and cost of the tests. While Uren, Miller and Irwin attempt to provide useful information about the costs and benefits of testing, they are not consistent between the case studies in the information they report. They do not analyse their results, nor do they make any conclusions about them. Of the test case selection strategies, they discuss the *touch testing* strategy which requires that each function of a system is tested at least once. Touch testing is a compromise in the trade-off between the benefits and the costs of testing; the test suite is smaller, less complex and easier to maintain than a complete test suite, but it is not as thorough.

Ostrand, Sigal, and Weyuker [13], offer a framework to organise the *functionality units* of a specification-based system and the critical test cases of these units and describe a tool, SPECMAN, that provides interactive support of this framework during the test developer's analysis of a specification. They identify a functionality unit as a "single atomic operation of the system or conceptual higher-level groupings of operations [13, p. 41]." Units are identified with their associated test case name and implementation modules. Each test case name is defined separately by its input/expected output pair, purpose and tester. While Ostrand, Sigal, and Weyuker propose an interesting framework for developing test cases, they provide no information about how productive applying this framework is. SPECMAN is primarily a test developer's project management tool; it manages and records all the steps in the

testing process but performs none of these steps. It is limited by the assumption that there is a specification for the system to be tested. Evaluation of this tool's effectiveness is not discussed as the prototype of the tool was still being implemented.

Ostrand and Balcer propose another method for functional testing [12], which involves categorising the functional units of a specification, identifying the parameters and environmental conditions and partitioning the ranges of possible conditions for each unit. Selections from these final partitions are used as the test cases. Ostrand and Balcer describe a tool that guides the user through these steps. The user then develops individual *test frames*, which are documentation for a particular test case. The tester transforms the test frames into a test case. Again, they provide some support in choosing inputs but have not automated any step in the testing process; they provide a test case management tool.

In [6], Bird and Munoz investigated automatic generation of random self-checking test cases. They are interested in systems to generate the software for most of the steps in the test process: test harnesses, input, expected output, and test result evaluation. This system works well for compiler and sort/merge applications but they discovered that, for graphical software, it is not always possible or efficient to make the output self-checking. They propose some techniques to make manual checking easier; these techniques involve simplifying the screen output as much as possible, using test cases that present easy to recognise output, using grids on the screen, and querying the tester about the screen contents.

Rodrigues, Tracy and Miller developed the tool, AUTOmate, specifically to support testing systems that use complex interfaces such as bitmapped screens, the mouse and keyboard [16]. AUTOmate is a keyboard and mouse-action recorder and playback tool with screen capture, comparison functions, and script editing. AUTOmate's bitmap comparison functions are strict; a one pixel difference in the display causes the comparison to fail. They often detect irrelevant failures, as minute changes in the screen display may be within allowable requirements. AUTOmate uses a lot of disk space to record the scripts and expected output. AUTOmate addresses several steps in the testing process: the test harness, executing tests, recording actual output, and comparing test results. Unfortunately, AUTOmate is only available for Sun Microsystems computers and is not available for the Macintosh.

Hoffman describes how to use the SCR software development approach to specify the syntax and semantics of a module interface [10]. He provides several examples of modules that are specified using this method to illustrate the power and effectiveness of module specifications. We use the method to describe the modules we developed and tested.

Chapter IV

System Testing

We have developed and implemented system testing for the *Scalc* program of the *BB* system, and for *CaseTool* based on the steps outlined in the Testing Process section of Chapter II. We discuss our experiences and results with the testing process as applied to both of these programs. First, we present our experiences with *Scalc*. Our experiences with system testing *CaseTool* were two-fold; we present *Scheme One*, discuss its shortcomings and then present *Scheme Two*.

Building Blocks: Scalc

Since *Scalc* was originally developed and tested in C on UNIX and then translated to Object Pascal for the Macintosh, we performed system testing to ensure that the new version of *Scalc* met the requirements.

Approach

We discuss the testing of *Scalc* by investigating the actions at each step in the testing process. All documents pertinent to *Scalc* testing are in Appendix A.

Develop a Test Plan

The strategy for test case selection identifies three critical states of this stack based calculator: an empty stack, a partially full stack, and a full stack. The requirements document describes the calculator functions and the error messages. We employed touch testing to perform test case selection for exceptional behaviour; for each of the calculator commands, we generated all possible error messages. We used a slightly more comprehensive test case selection strategy to test normal behaviour: for each of the critical system states identified, we invoked each command once. The implementation of the test plan involved no stubs and the *makefile* contains the driver. The test cases are stored in the *input* and *exp,* directories and the actual output in the *act* folder.

Build the Test Harness

The test harness consists of the *makefile*, a folder of input files, a folder of expected output files, and a empty folder for actual output.

Choose Inputs

We developed the inputs from the Test Plan by placing each test input in a separate file in the *input* folder. Each file has a mnemonic name to identify the command tested with an *n* suffix denoting a normal operation test and a numeric suffix denoting an error message test.

Determine Expected Outputs

The expected outputs were developed using the UNIX version of *Scalc* as a test oracle. Each expected output was stored in a separate file in the *exp* folder using the same file name as the input file.

Execute the Tests

Since *Scalc* is an MPW tool and must be executed in the MPW environment, we executed the tests in the same environment. The tests were executed using the command: *buildprogram runtest*. For each file in the *input* folder, the *makefile* invoked *Scalc* and redirected standard input from that file. Standard output was redirected to a file in the *act* folder.

Compare the Expected and Actual Output

The *makefile* uses the MPW tool, *compare*, to detect differences between the two files. The *makefile* compares each test case after the test has been executed and reports differences to the screen.

Evaluate the Test Results

No failures in the software or the test cases were detected.

Tools

MPW provides all the facilities necessary to perform file creation, deletion and comparison.

Results

We performed *Scalc* testing once. We executed 48 test cases and detected no failures. The time involved in testing *Scalc* is shown in Figure IV-1. There are two measurements for evaluating the time taken for testing *Scalc*. Testing costs on the MPW version were small because most of the costs were incurred while developing the testing for the UNIX version. Had there been no UNIX version, the costs would have been only slightly less than the sum of the two columns. The additional time spent on the MPW version primarily consisted of learning the MPW features to perform the testing.

Step	Testing Phase	UNIX	MPW	Total
1.	Develop a test plan	3	0	3
2.	Build the test harnesses	2	.5	2.5
3.	Choose inputs	4	2	6
4.	Determine expected outputs	1	1	2
5.	Execute the tests	0.08	0.08	0.16
6.	Compare the expected and actual outputs			
7.	Evaluate the test results	0	0	0
	Total	10.08	3.58	13.16

Figure IV-1. Time in Hours Spent On System Testing for *Scalc*

Machine costs were negligible; the test plan, input files, expected output files, and actual output files occupied 75 Kb of disk space.

Advantages

The principal advantage of this testing approach is its simplicity. It is easy to understand and executes fully automatically. We became more familiar with MPW and the mechanisms for building programs and executing scripts in this environment.

Disadvantages

We are concerned about the costs of maintaining the testing if changes are made to the user interface. The inputs and expected outputs are spread out over 48 files and these files would require maintenance if the user interface changed.

If the system progresses from an MPW tool to a Macintosh application, *Scalc* may not be as easy to test. *Scalc* does not use the features of the Macintosh that *CaseTool* does. Nor does *Scalc* use a complex user interface or graphics, like *CaseTool*. The MPW tools that supported *Scalc* testing will not apply well to *CaseTool*.

CaseTool: Scheme One

Scheme One was the first test scheme designed and implemented for ABC to test *CaseTool*. The testing was scheduled to be performed once each week and to take from four to six hours to complete.

Approach

All documents referred to in this section are in Appendix B.

As described in Chapter I, *CaseTool* has controllability and observability problems. The lack of tool support for input redirection and output capture and comparison mean that little automated support is available for testing *CaseTool*. Also, because the system was under development and lacked a requirements document, the tester had difficulty verifying the results of the testing; the tester had no guide for determining the expected output. Part of our testing involved developing documents to use as the requirements.

Since we had no tools to control testing, we needed to take care in recording the steps we used at each stage of the testing. Detailed records of the testers' actions provided control over the process and allowed accurate observations of the results. The emphasis of this test scheme was on establishing standard testing tasks. Testing was controlled through the following documents:

1. Test Inventory
2. Detailed Inventory
3. *CaseTool* Standards
4. Master Test Plan
5. Weekly Test Plan
6. Error Report

Following the Macintosh desktop metaphor, *CaseTool* is a *noun* and *verb* oriented system. Systems are composed of nouns, or objects, and verbs, or functions, that can be applied to objects. Since we had no requirements document and insufficient knowledge

about the requirements of *CaseTool* we used the Test Inventory to define the scope of the testing by listing the system's objects and functions. The Test Inventory is composed of the *CaseTool* Standards document and a list of the *CaseTool* functions. In *CaseTool*, similar functions are placed in a single pop-down menu. In the Test Inventory, we grouped these functions into categories corresponding to these menus and palettes. The Detailed Inventory contains descriptions of the functions listed in the Test Inventory. It describes each function as observable by the user through the screen, mouse and keyboard: the conditions before, during and after the function is executed by the user. Each function also has an *Exceptions* section for describing exceptional behaviour, and is cross-referenced to other related sections in the test documentation. Together these documents served as test oracle. We used *CaseTool* Standards to identify the objects in *CaseTool*. These included items such as the mouse icons, the file icons, the report icons, and the draw-able graphics. There are also functions that can be applied in any state of *CaseTool*. To avoid redundancy, we described these *global functions* in *CaseTool* Standards.

Decisions about the Test Inventory, *CaseTool* Standards and Detailed Inventory were made by ABC. ABC decided what functions should be in the test inventory and what their expected behaviour was. We assumed full responsibility for maintaining these documents.

Develop a Test Plan

There are two test plans for testing *CaseTool:* the Master Test Plan and the Weekly Test Plan. The Master Test Plan identifies the test case selection strategy as touch testing. We used touch testing because of the large number of functions, objects and execution conditions in *CaseTool.* With only one tester, we could not afford to apply a more comprehensive strategy. For each category in the Test Inventory, the Test Plan describes a sequence of steps for test case selection.

Because the Test Inventory is too large to be completely tested in one test session, the Weekly Test Plan outlines which functions to test in the current test session and possible limitations in some functions or objects in *CaseTool.* The test plans were written through consultation with ABC. Often ABC requested that we paid particularly close attention to one function, as they knew the underlying code had changed. Other times, they informed us that changes were in progress on a function and it was not ready for testing.

The implementation of the test plan involved no stubs and no drivers. The test data folder contained a number of Pascal source code files for use as file input to *CaseTool.*

Build the Test Harness

There is no test harness except the test data folder of Pascal source code. We chose the *symtbl* folder from *BB* to use as the test source code. It provided Pascal source code and non-Pascal files, like

Makefile and symtbl.intspec, which were useful for exception testing.

Choose Inputs

The Test Plan describes a sequence of steps for developing a test script of the individual test cases to execute. Inputs are the key strokes, mouse actions and source files. When getting source files for analysis, the tester used the *symtbl* folder files.

Determine Expected Outputs

Our background knowledge encompasses the expected output for the input action, but we used the Detailed Inventory to support and supplement our knowledge.

Execute the Tests

We executed the tests by hand.

Compare the Expected and Actual Outputs

We performed the output comparison. Discrepancies between the expected and actual output were recorded in the *Error Report.*

Evaluate the Test Results

Evaluation of the test results was done through meetings with ABC personnel. The Error Report was examined test case by test case to determine if the failure was caused by a fault in the testing or in the *CaseTool* source code. Test case failures that were caused by

faults in *CaseTool* were filed with ABC. We corrected the faults in testing.

Tools

Several tools were investigated to see if they could provide automated support for testing. Script recorders would be ideal for recording long input sequences and screen capture and comparison tools would be useful for comparing the bitmapped images on the screen. We investigated MacroMaker, for keyboard and mouse input recording, and Camera for screen capture.

Unfortunately, these tools are primitive, and are not designed with testing in mind. The scripts provide no time delays to ensure that the system has time to achieve a desired state before continuing the script. Scripts can not be modified, only extended, and must be recreated from the start every time a change is required. We could find no tools that resolved these problems. Furthermore, we could find no convenient way for comparing the saved screen output. Using screen capture would necessitate that we implement a tool to do this task or print out every screen image for visual comparison.

At the beginning of the testing process, the *CaseTool* user interface was changing frequently. The requirements of *CaseTool* were still being developed and the user interface could be substantially different from one week to another. Even if tools such as script recorders and screen capture had been available, we could not have used them: they would have been too expensive to maintain.

Later, as the interface and behaviour stabilised we again investigated the possibility of using these tools, but they could still not be efficiently used because of the limitations of the tools. MacroMaker could store and play the many menu operations to read test files and create test *CaseTool* documents, but, since it could not deal with functions that involved operating on selected objects, we did not use it.

Results

We performed testing on *CaseTool* using Scheme One nineteen times over a period of approximately seven months. During the first two months tests were executed every other week, and then increased to every week. The results of our testing are summarised in Figure IV-2.

Developing the requirements documents necessary to understand *CaseTool* and determine expected output took a large portion of the time incurred. Due to the lack of automation, we incurred few costs building the test harness. The complexity of executing the tests caused the four steps to be performed as if they were one step.

Step	Testing Phase	Total Hours
0.	Develop the requirements documents	55.5
1.	Develop a test plan	15
2.	Build the test harnesses	1
3.	Choose inputs	82.75
4.	Determine expected outputs	
5.	Execute the tests	
6.	Compare the expected and actual outputs	
7.	Evaluate the test results	35
	Total	188.25

Figure IV-2. Time in Hours Spent On System Testing for *CaseTool: Scheme One*

Advantages

Each stage of the testing process yielded interesting information about *CaseTool* and testing. *CaseTool* system testing began early in the development of the product when there was no requirements document. ABC did not have a clear definition of the final product. Early system testing forced ABC to clarify *CaseTool's* requirements; design decisions were recorded and inconsistencies in system behaviour were more obvious when function definitions were explicitly recorded. The Test Inventory, Detailed Inventory and Standards were used by ABC in developing the user's guide [1].

Because we used a defined set of steps to execute the test cases, the failures found were usually easy to reproduce. Sometimes, the people at ABC would know of a failure that they discovered through informal use of the *CaseTool*, but were unable to reproduce.

The number of failures found provided useful information about the reliability of the system. The test results showed that faults existed. The people at ABC had a surprising attitude towards their product; they thought that their software was reliable, and yet, they knew that they employed few software development techniques and that the product was probably riddled with faults. Testing brought them concrete evidence of the reliability of *CaseTool*.

We learned about the considerable task of system testing, particularly system testing on software the size and complexity of *CaseTool*. ABC also benefited from this knowledge and developed a more concerned view about system testing.

Disadvantages

The work involved in this testing process was substantial. Testing was expensive in personnel time at every phase of the testing process. Because testing was so expensive and difficult, noone else was willing to perform the testing. ABC had neither the time or resources to contribute substantially.

The lack of a requirements document meant that a large amount of time was spent maintaining the test documentation. Because of the time this task demanded, work suffered in other

areas of the test process, and the documentation was not updated as often as we would have liked. We caught ourselves taking advantage of the testers' assumed knowledge and the knowledge that we were the testers. We did not update the test documents as often as we should have, if the time budget on a given week was small.

An explicit test suite was never identified: only a sequence of steps to test case selection. This sometimes affected the ability to reproduce the failure consistently. Fewer tests were performed and fewer failures detected.

ABC is a small company and did not have the available personnel or equipment for testing. Even if ABC had these resources, the complexity of the test documentation and execution would have meant that few people at ABC would have been capable of performing the tests. The tests required someone with a great deal of knowledge about the Macintosh, *CaseTool*, Pascal and the test files.

CaseTool: Scheme Two

This new testing scheme addressed several of the weaknesses of the first scheme and took advantage of a more advanced stage in the development of *CaseTool*. Now that ABC had produced a user's guide, we used this document to define the requirements and had ABC responsible for keeping the document up-to-date. The many hours saved were used to implement a more concrete test implementation. This enhanced the reproducibility of the testing, and made the testing easier for the testers to perform. Due to the close link

between the documents for test development and the test execution documents in Scheme One, it was nearly impossible for the test developer and executor to be different people; Scheme Two eliminates this problem.

Under this new scheme, we continued in the role of test developer, but the role of testers was filled by many different people. We performed some testing every three or four weeks, but the primary testers were two community college students hired by ABC to work once a week performing the tests. ABC will eventually take over responsibility for both roles of test developer and tester.

Approach

As in Scheme One, we look at each step in the testing process. Test documents are in Appendix C.

We still had some of the problems of Scheme One, including controllability and observability problems and lack of tool support. We still required careful record keeping to control the testing through the testing documents. The test developer still needed a great deal of knowledge about *CaseTool*, but there was no reason that *all* the testers needed to know *all* there is to know about *CaseTool*. Scheme Two involved using different test plans to accommodate the different qualifications of our testers; we divided our testers into three different groups and developed three different test plans:

1. Macintosh Test Plan (M Plan)
2. Macintosh & CaseTool Test Plan (MC Plan)
3. Macintosh, CaseTool & Pascal Test Plan (MCP Plan)

The Summary Test Plan (S Plan) describes the breakdown between the different test plans, and common implementation information.

The M Plan was executed by testers that needed to be familiar with the Macintosh operating system [5] and typical Macintosh drawing applications such as MacDraw. Testers who executed the MC Plan had the same knowledge as the M Plan testers and an understanding of *CaseTool* objects and functions [1, sections 1 & 2]. The MCP Plan testers had the same knowledge as the MC Plan testers plus familiarity with Pascal, the test source code and the relationships between the identifiers in the source code.

Each test plan has two test suites:

1. M Plan
   Macintosh Partial Test Suite (M PSuite)
   Macintosh Full Test Suite (M FSuite)
2. MC Plan
   Macintosh &CaseTool Partial Test Suite (MC PSuite)
   Macintosh &CaseTool Full Test Suite (MC FSuite)
3. MCP Plan
   Macintosh, CaseTool & Pascal Partial Test Suite (MCP PSuite)
   Macintosh, CaseTool & Pascal Full Test Suite (MCP FSuite)

One test suite is a *partial* test suite that the testers could perform in less than one hour whereas the *full* test suite takes four hours to complete. This division allows the testers to schedule the amount of testing with respect to the amount of time available for testing.

The partial testing was performed frequently, every time a new version of *CaseTool* is compiled, and the full testing was

performed for the versions that were released outside the company. ABC compiled a new version of *CaseTool* every week and released *CaseTool* to various outsiders every month.

Develop a Test Plan

The Summary Test Plan contains information common to all the test plans. All of the test plans are implemented and executed the same way. Each test plan breaks test case selection into normal and exceptional behaviour. For each of the sections, the test plans identify the overall goals and the more detailed stimuli. Each stimulus is assigned a time estimate in minutes for both partial and full testing. The implementation section on the test plans specifies information particular to the individual test plan that was not listed in the Summary Test Plan.

Macintosh Test Plan

The M Plan focuses on testing Apple standards such as the File and Edit menus, the palette functions, the window functions, and text editing.

Macintosh & CaseTool Test Plan

The MC Plan addresses the project menu, the navigation palette, the report creation, the display menu on reports and CaseTool objects such as the source code, folder and report icons.

Macintosh, CaseTool & Pascal Test Plan

The reports generated by applying the tools functions to different kinds of Pascal identifiers are the focus of MCP Plan.

Build a Test Harness

The test harness was largely unchanged from Scheme One. We used the same set of test files, *symtbl* folder files, except we were a little more restrictive. We were only interested in a few files, which were listed in the plans. We also used some extra files and an extra disk for exception testing; again, these are listed in the plans.

Choose the Inputs

All inputs for each test plan are explicitly described in the corresponding test suites. The test suites are sequences of tables of *events*. Each event is an action that the tester performs on some specified object.

The tables are divided in two ways: by goal and by *group.* The goals directly correspond to the goals in the test plan; they appear in the same order and use the same wording. A group is a sequence of tables that facilitates recovery from a test failure. If a test failure causes *CaseTool* to crash, and the machine must be restarted, then it is necessary for the tester to recover and continue testing. The tester need only return to the beginning of the most recent group and repeat all of the steps except the one that caused the system crash.

Determine the Expected Outputs

Again, the testers are responsible for determining the expected output for each input action. Each test plan lists the references that provide the background knowledge.

Execute the Tests

The tests are executed by hand. Testing begins with the partial test suite, executing the events in each table, and working the tables left to right, top to bottom. Full testing assumes that partial testing has already been performed.

Compare the Expected and Actual Outputs

The tester performs the output comparison. Each event has a blank box associated with it, filled in with a √ for success or an *F* for failure. Discrepancies between the expected and actual output are recorded in the *Error Report* including the table number, row and column number of the failed event, and a description of the actual output.

Evaluate the Test Results

The new testers have provided useful comments that we have incorporated to make testing easier for the testers: better explanation of events, and failure recovery. Furthermore, we applied Fagan's inspection techniques [8] to write better test plans and test suites; we worked on completeness, readability, and understandability. As a result, we used simpler graphics to make evaluation easier [6] and decomposed complex tables into many smaller tables. Test results are discussed and evaluated with ABC, as in Scheme One.

Tools

We did not investigate tools any further; our test process and *CaseTool* had not changed enough to make them useful.

Results

The students performed most of the testing during a two month period and they located a total of 136 failures. Figure IV-3 tabulates the results for testing using Scheme Two. The M, MC, and MCP designations correspond to the Macintosh tester, the Macintosh & *CaseTool* tester, and the Macintosh, *CaseTool* & Pascal tester, respectively. We began with the M testing, as the students were newly hired and did not have the necessary knowledge to perform the MC, or MCP tests. Time constraints restricted them to the partial test suites.

We incurred no costs in developing the test harness as we used Scheme One's harness. After the first two tests, we were no longer involved in evaluating the test results. The test suites and error reports were sent directly to the programmers for evaluation.

Step	Testing Phase	M (7 Tests)	MC (4 Tests)	MCP (2 Tests)	Total
1.	Develop a test plan	6	8.5	4	18.5
2.	Build the test harnesses	0	0	0	0
3.	Choose inputs	16.83	8.33	5.33	30.49
4.	Determine expected outputs				
5.	Execute the tests	7.16	2.92	1.75	11.83
6.	Compare the expected and actual outputs				
7.	Evaluate the test results	2	0	0	2
	Total	31.99	19.75	11.08	62.82

Figure IV-3. Time In Hours Spent On System Testing for *CaseTool.*: *Scheme Two* (as of May 15, 1990)

Advantages

One advantage of this revised method is its simplicity. It is standardised, and the rules are straightforward to apply. Testing was easier to develop and is easier to execute. The test developers spend no time dealing with the requirements of *CaseTool.* The test developer has a set of documents that are easy to maintain. The smaller test plans decompose the testing into manageable subsets. The testers have a detailed script; they know exactly what the inputs are and the tests are simple to execute, and compare results. Our scheme of grouping tests for restarting test execution after system crashes helps testers recover from system crashes quickly

and continue testing. The results of the testing are more reliable, and ABC has a precise record of the events that caused a failure.

As an indication of the increased usability of the testing, ABC has been more interested in addressing the failures detected and ensuring that they provide equipment and personnel to perform the testing. Furthermore, now that the test documentation is under control it is possible to allocate time to maintain and extend the testing.

As the product stabilises, it should be possible to use *CaseTool* to generate the expected output screens for advanced testers. Requiring the tester to have detailed knowledge of the test files would not be necessary and the testing could be performed by testers with less background.

Disadvantages

The execution of the test cases was still done by hand, and w
expect to automate the process. Such testing is more expens
more error prone than automated testing. The test suites ar
tedious to maintain.

Manual testing
- expensive
- error prone
- test suites are difficult to maintain.

Chapter V

Module Testing

We have developed a scheme for module testing and have used it to test the *symtbl* module of the *BB* system, and the *Space* module of *CaseTool*. We present the scheme and discuss our experiences and results in applying it to both of these modules.

A Scheme for Module Testing

Our scheme is modeled on *pgmgen* [9], a test program generator for C code on UNIX. *Pgmgen* takes a file of test cases as input and produces a test driver written in C. Due to *pgmgen's* dependency on the Unix tools YACC and LEX, porting it to Macintosh Object Pascal would have been expensive. Instead, we developed modules to provide services similar to those provided by *pgmgen*. Our approach requires some additional effort from the tester. We discuss our scheme below. Appendix D contains all pertinent documents for this scheme.

Approach

Test programs are implemented as a sequence of test cases. Test cases are written in Pascal and take the form:

```
sTCase(tCaseLabel);
trace
sCheckExc(expExc);
```

or

```
sTCase(tCaseLabel);
trace
sCheckT(expExc, expVal, actVal);
```

such that:

`sTCase` marks the beginning of each test case and associates with it *tCaseLabel*.

tCaseLabel is a string that identifies a test case.

trace is Pascal code containing access program calls on the module under test.

`sCheckExc` compares the actual occurrences of exceptions to *expExc* and reports any differences.

expExc denotes an integer constant representing an exception that the trace is expected to generate. The integer constants, `DontCare` and `NoExc`, denote cases where the tester does not care if an exception is signaled and when no exception is expected, respectively.

`sCheck`*T* compares the actual occurrences of exceptions to *expExc* reporting any differences, and then compares *actVal* and *expVal* again reporting any differences. *T* is one of `Int`, `Str`, or `Bool`.

expVal and *actVal* are expressions that evaluate to the expected and actual output.

Reporting a test failure involves printing to standard output the *tCaseLabel* and the expected and actual results of the test case.

These routines also maintain statistics to count the test cases, the exception and value failures. The procedure `gSum` prints these statistics.

Because we write the test cases in Pascal, we are able to include other Pascal source code. This includes constants, procedures, functions, or in-line code to support iteration over the test cases.

Most test cases either check for exceptional failures or output failures; it is possible to write test cases that check both. Using *symtbl,* described in Chapter I, an example of each type of test case is:

```
sTCase('**** Delete ids not in the table ****');
s.sInit;
s.sDel(-1);
sCheckExc(notLegId);

sTCase('**** Empty str not in empty table ****');
s.sInit;
sCheckBool(NoExc,false,s.gLegSym(''));
```

The first case checks that the trace `s.sInit;s.sDel(-1);` signals `notLegId`. The second case, after executing the trace `s.sInit;`, checks that `s.gLegSym('')` returns false and also that no exceptions occurred; by specifying `NoExc`, `sCheckBool` confirms that no exception did occur. The string label makes it easier to locate a particular test case in a large file of test cases.

Implementation

Our testing scheme uses two modules: Batch Test (*BTest*) and Batch Exception (*BExc*). *BTest* provides driver support for keeping track of

individual test cases, comparing the test results, printing test failures to standard output, and printing test statistics to standard output. *BExc* provides tracking for exception handling. It records when an exception has been signaled, how many exceptions have been signaled and if an exception has been signaled. The modules are implemented as Pascal units: *BTest* and *BExc.*

Use

In order to use these two modules effectively, we developed a scheme for using them. The module under test is implemented as a Pascal unit with a separate unit for exception handling. The tester implements the exception handling unit using the *BExc* unit. When implementing testing for some module *X*, the tester must implement a Pascal unit called *XExc* using the following skeleton:

```
unit XExc;
uses BExc;
interface
     {each exception has a procedure header}
     function gName (exc:int) : string;
const
     {each exception has a named constant assigned a
     consecutive integer beginning at 1}
     numExc = {number of exceptions}
implementation
     {each exception implemented as follows}
     procedure y;
     begin
          sExc( {constant corresponding to y} )
     end;
     function gName (exc:int) : string;
     {returns the string corresponding to the name of
     the exception}
end.
```

By implementing exception handling as calls to *BExc* it becomes possible to record when exceptions have been signaled.

We expect testers to implement drivers for a module *X* using the following skeleton:

```
program XBTest
uses BTest, BExc, XExc, X;

{global definitions}

begin
      {initialise all units}
      {test case 1}
      {test case 2}
      {test case 3}
      ...
      gSum;
end.
```

The Pascal program is compiled and executed. Test failures are reported to standard output, and summary statistics are reported when all tests have been executed.

Building Blocks: symtbl

Since *symtbl* was developed and tested in C on UNIX and translated to Object Pascal for the Macintosh, we performed module testing to ensure that the new version of *symtbl* met the specification. As stated in Chapter I, the *symtbl* module is implemented in the Pascal unit, *symtbl,* and the exceptions that *symtbl* can signal are implemented in the Pascal unit, *symtblExc.*

Approach

We discuss the testing of *symtbl* by investigating the actions at each step in the testing process. All documents pertinent to *symtbl* testing are in Appendix E.

Develop a Test Plan

The test plan for *symtbl* is based on a test plan for testing the C version of *symtbl*. The test plan makes an important assumption about the module. The interface specification for symtbl does not state the method of allocating identifiers for symbols; this information is discovered through inspection of the source code. We need to know this allocation policy in order to effectively test this module. Since this allocation policy is likely to change in future implementations, we recorded the allocation policy as an assumption in the test plan.

The test plan identifies two critical module state characteristics: the number of table entries and the length of the symbols. The critical numbers of symbol table entries are an empty table, a partially full table, and a full table. Symbols have two critical lengths: a short length, and the maximum length of a symbol. The test case selection strategy includes exceptional and normal cases; exceptional cases include testing that each exception for each access program is signaled and normal cases include testing every access program once for each critical state value.

Using our scheme described earlier, Symtbl Batch Exception (*symtblBExc*) is the exception signaling unit and Symtbl Batch Test (*symtblBTest*) is the driver.

Build the Test Harness

SymtblBTest implements the test plan using the test skeleton described above. *SymtblBTest* uses *symtbl* and the units from our testing scheme: *BTest* and *BExc*. *SymtblBTest* consists of many test cases with extra Pascal source code to implement iteration over the critical module states. The test plan outlines the additional routines needed and implemented as global definitions. Descriptions of the critical module states are stored in an array. *SymtblBTest* executes the test cases in a while loop, iterating once for each critical module state.

Choose Inputs

The inputs are based on the test plan, the test harness routines, and the *pgmgen* script for the C version of *symtbl*. The test plan specifies some critical inputs to check, such as empty strings or specific sets of identifiers. For other symbols, we developed the function `tMksym(`*i, len*`)` which returns a symbol consisting of the integer *i* converted to a string and padded to length *len* with asterisks. The ability to easily create unique symbols of a given length helps write the test cases. By being able to generate the strings, we do not have to maintain large numbers of string constants; this reduces the likelihood of human error in maintaining the test cases. The *pgmgen* script provided guidance.

Determine Expected Outputs

We determined the expected output, using the *symtbl* specification and the identifier allocation policy that we described earlier. Without this assumption, we could not determine the expected identifiers for the symbols that we add to *symtbl*.

Execute the Tests

All modules are implemented as Object Pascal units in MPW and executed in the MPW environment. The *makefile* for the *symtbl* unit compiled the testing programs when we executed the command: *buildprogram symtblBTest*. We ran the tests by executing the command: *symtblBTest*.

Compare the Expected and Actual Output

BTest provides harness support for comparing expected and actual output. We wrote the test cases to call these routines for comparing the expected and actual output, and reporting test failures to standard output.

Evaluate the Test Results

We investigated failures by hand. We determined that some test failures were in the test implementation and that others were in the *symtbl* unit. We located the faults and corrected them. Once all failures were resolved, we evaluated the test plan and test implementation by hand using Fagan's inspection techniques [8]. We had inspected the C version of the testing on UNIX, and we were

checking that no errors were made in the translation, and that the translation accurately tested the Pascal module.

Tools

We developed the tools that we used to implement the testing for this module: *BTest* and *BExc*. We described them in detail earlier. MPW provided the environment to compile and execute the test program.

Results

We performed *symtbl* testing until no failures were detected. The time involved in testing *symtbl* is shown in Figure V-1. The Testing Tools costs included the development of the interface specifications and implementation of the *BTest* and *BExc* modules. The *Symtbl* Testing costs included the development of the test plan, and the implementation of the *symtblBExc* and *symtblBTest* modules.

Developing the test harness routines for *symtblBTest* took longer than we expected due to our unfamiliarity with Macintosh programming. Testing *Scalc* and translating *BB* to Pascal saved us some time as we had a better understanding of Pascal and MPW. Nonetheless, we still spent time learning about the Pascal compiler to perform such tasks as substituting the *symtblBExc* unit for the *symtblExc* unit, and using the libraries to implement the `tMksym` routine. We were able to save time on the test evaluation as we had

evaluated the effectiveness of the testing for the C version, and we only had to ensure that the new testing continued to be effective.

Step	Testing Phase	Testing Tools	*Symtbl* Testing	Total
1.	Develop a test plan	N/A	2	2
2.	Build the test harnesses	11.5	11	22.5
3.	Choose inputs	N/A	2	2
4.	Determine expected outputs	N/A		
5.	Execute the tests	N/A	0.08	0.08
6.	Compare the expected and actual outputs	N/A		
7.	Evaluate the test results	1	2.75	3.75
	Total	12.5	17.83	30.33

Figure V-1. Time in Hours Spent On Module Testing for *symtbl*

We discovered several failures while testing *symtbl*: three failures in the test harness, and five failures in *symtbl*. Two of the harness failures detected were due to faults in the *symtblBTest* routines. When we located the faults we discovered that we had not initialised the testing correctly and we had not implemented `tMksym` correctly. The third test harness failure had detected a fault in the Boolean conditions that compared test results. The five failures in *symtbl* uncovered two faults in checking a parameter input and three faults in incorrectly written Boolean conditions.

Advantages

Every failure detected in *symtbl* resulted in a system crash. The causes of system crashes on the Macintosh can be difficult faults to locate. Because this scheme effectively isolated *symtbl*, we were able to locate the faults quickly.

Regression testing this unit should be much less expensive as the tools already exist and it costs almost no human time to execute. Choosing the inputs for symtbl was simple, and the resulting code is easy to understand, execute and maintain. The entire execution of the test cases was automated; test execution was quick. The tests were easy to re-execute and test results were easy to reproduce.

Development of the testing tools was only one third of the total development time for *symtbl*. The testing for *symtbl* was expensive, in this respect, but we now have tools that could be applied to other modules with relative ease. This should save development time for testing other modules.

Disadvantages

The major disadvantage with this scheme is that we are limited in the type of modules that we can test with this method. This scheme is only effective with call-based modules which use an exception scheme that is easy to monitor. Furthermore, if we are to test a module that uses other modules, the test plan and test harness

become more complex as they might include stubs for these additional modules.

Another disadvantage to this testing scheme was the initial development time in designing and implementing the testing tools and harnesses. Spending this amount of time on testing one small unit is not cost effective for a system with more than a few modules. However, we account some of this extra time as time invested towards tool development and personnel training. We increased the knowledge of the test developers and testers. We do not expect to incur these same costs again.

CaseTool: Space

Since ABC had no specifications for *Space,* we wrote specifications and ABC confirmed their accuracy. *Space* manages *CaseTool's* memory and provides access programs to increase the amount of memory available to *CaseTool*, store and retrieve information in memory, and free memory. Surprisingly, for testing purposes it is similar to *symtbl*; both modules manage a table of objects and provide routines to add and access objects in the table. We wanted to apply our scheme for module testing to this module. We adapted the implementation of *Space* to use exception handling as our module testing scheme expects.

Approach

We discuss the testing of *Space* by investigating the actions at each step in the testing process. All documents pertinent to *Space* testing are in Appendix F.

Develop a Test Plan

The test plan for *Space* is based on a test plan for *symtbl*. We enumerated the critical number of entries in the table, as for *symtbl*. We also identified the critical sizes of zones in the memory. These correspond roughly to the lengths of symbols stored in *symtbl*. The test case selection strategy lists critical tests. Exceptional cases include testing that the FailNil exception is signaled for each critical state value. Normal cases include testing every access program once for each critical state value.

Build the Test Harness

SpaceBTest (Space Batch Test) implements the test plan using the test skeleton described earlier for *symtbl* testing. The test plan outlines the additional routines needed and implemented as global definitions. As earlier, descriptions of the critical module states are stored in arrays.

We need the ability to request memory from the operating system to copy in and out of the *Space*; we developed an array type, `tStrType`, and a pointer to the array type, `tStrPtr`, to accomplish this. Since these types do not have support in *BTest*, we provided this support in *SpaceBTest*.

Choose Inputs

We chose the inputs using the test plan, and the test harness routines. We developed the routine `tMkStr(`*c len*`)` which returns an array of chars where the first *len* positions consist of the character *c*. The ability to easily create unique memory addresses of a given length helped us write the test cases. This provides a similar service as `tMkSym` in *symtbl*, reducing the likelihood of human error in the testing process.

Determine Expected Outputs

We determined the expected output to correspond to the inputs, using the *Space* specification.

Execute the Tests

All modules are implemented as Object Pascal units in MPW and executed in the MPW environment. The *makefile* for the *Space* unit compiled the testing programs when we executed the command: *buildprogram SpaceBTest*. We ran the tests by executing the command: *SpaceBTest*.

Compare the Expected and Actual Output

BTest provides support for comparing expected and actual output for the integer, Boolean and string types. In *SpaceBTest*, we implemented the routines, `tSCheckStrPtr(`*expExc, expVal, actVal, len*`)` and `tPrtStrPtr(`*expVal, actVal, len*`)` to support comparison of expected and actual output for the `tStrPtr` type. The routine

`tSCheckStrPtr(`*expExc, expVal, actVal, len*`)` evaluates *expVal* and *actVal*; if they are the same pointer or if they are different pointers but point to identical characters for the next *len* positions in memory then the test case has succeeded otherwise `tSCheckStrPtr` reports a test failure.

Evaluate the Test Results

We investigated failures by hand. We determined that some test failures were in the test implementation and we have yet to locate the fault associated with our one remaining failure.

Tools

We used the tools that we developed and discussed earlier for this module: *BTest* and *BExc*. MPW provided the environment to compile and execute the test program.

Results

We are still implementing portions of the test plan. In particular, we have not tested any of the exceptional behaviour. We have implemented iteration over the critical module states for checking the `NewZone`, `Occupy`, `CloneOccupant` and `Free` routines.

We report the time, as of May 20, 1990, involved in testing *Space* in Figure V-2. We incurred 6.5 hrs developing a specification for *Space*. The test plan was straightforward to develop as we had a model in the *symtbl* test plan. We incurred fewer costs for

developing the test harness as *BTest* and *BExc* were already implemented. Developing the test harness was still time-consuming as we had to implement the additional support for the new type.

Step	Testing Phase	*Space* Testing
0.	Develop a specification	6.5
1.	Develop a test plan	6
2.	Build the test harnesses	11
3.	Choose inputs	1.5
4.	Determine expected outputs	
5.	Execute the tests	
6.	Compare the expected and actual outputs	.08
7.	Evaluate the test results	2
	Total	27.08

Figure V-2. Time in Hours Spent On Module Testing for *Space* (as of May 20, 1990)

We discovered several failures while testing *Space*: one failure in the test harness, and one failure that remains unresolved. The first failure involved pointers and object initialisation. The unresolved failure was based on the large demand for memory this test program generates. We knew that we needed to lower this demand, but we are puzzled by the behaviour of the *Space*. The exception, `FailNil`, did not get signaled when the amount of allowable memory was exceeded during normal testing. We do not know why and we are anxious to implement exception testing to locate the problem.

Advantages

Due to our experience with *symtbl*, *Space* testing was surprisingly easy to develop and execute. We had some difficulty working with our new types, but some scheme for filling memory had to be developed, regardless of the testing scheme we used. We believe these new types are the simplest.

Testing costs were modest considering the complexity of the module under test. We were able to take advantage of existing tools for developing the testing. Furthermore, regression testing this unit should be less expensive, as most of the costs incurred were at the harness and specification development phases.

As with *symtbl*, the test execution is automated and the test cases are explicit; they are executed quickly and the results are reproducible.

Disadvantages

We had difficulties developing testing for the *Space* module primarily because of the lack of specifications and the complexity of the module. We had to spend time learning how the module worked and finding out what it was supposed to do. We had to invest time in understanding memory allocation in the Macintosh Operating System.

Furthermore, the lack of a methodology for developing the module meant that the interface was not always practical [10].

Routines were developed to fit a particular purpose and we did not have as much access to the module state as we would have liked in order to observe the module's behaviour. This contributed to our difficulties in understanding test failures. The test harness was complicated by the need to develop an additional type.

Chapter VI

Conclusions

We present a summary of the work performed, the contributions of this work and the possibilities for future work in this area.

Summary of Work

We have presented a process for testing and applied this process to several different types of software. We used this process to test two software systems: *CaseTool* and *Scalc*, and two modules: *Space* and *symtbl*. Each item under test has different controllability and observability characteristics. For each of the items we tested, we developed schemes and test plans designed to meet the criteria defined in Chapter I. To confirm our expectations, we implemented the test plans and maintained careful records of the costs incurred.

The testing we developed for *Scalc* does have the characteristics of good testing. The tests are cost effective to execute, develop and maintain. We used automation to reduce human labour and chance for error to a minimum. We were able to use existing tools to further reduce development time.

CaseTool was difficult to test; its complex user interface made controlling and observing *CaseTool* challenging. Since we discovered that our first scheme for testing was too costly, we

designed and implemented a second scheme. Our records show that this second scheme came closer to good testing. While Scheme One found more failures, Scheme Two had only being executed for two months; we expect Scheme Two to find more failures than Scheme One. The explicit test tables made testing straightforward to execute and the division of the testing into manageable subsets ensured that the tests were maintainable.

The *symtbl* module had a more useful interface than *Space* that aided software testing; we had more access programs that allowed us to control the module and observe its behaviour. Testing *symtbl* was cost efficient, and we applied that knowledge to *Space*, with relative ease. Unfortunately, the difficulties we encountered in controllability and observability meant that we could not execute as many test cases as we would have liked.

Contributions of Work

Without a complete understanding and without control of the costs and benefits of testing, testing is not performed satisfactorily; often testing is not performed at all. Money and time are wasted producing ineffective testing that is difficult to execute and maintain. The results are ignored and the software is not reliable. We focused our work on broadening our understanding of testing, increasing our control over the costs of testing and providing effective testing for software.

To accomplish this, we developed testing for four pieces of software: two systems and two call-based modules. We designed the testing to be cost effective to develop, execute and maintain, given the controllability and observability characteristics of each piece of software.

We have provided detailed figures on the costs incurred throughout the entire process. Of all the testing papers we read, few attempt to report similar information about the costs they incurred while testing. We attempt to understand the *entire* testing process: the steps involved in testing, the importance of developing a good test scheme and plan, and the roles that people and computers can play in the testing process. Many researchers look at only one aspect in this process and ignore the others. They ignore the costs that are incurred elsewhere as a result of their schemes. For example, some researchers investigate test input selection ([13], [12], [6]) but do not discuss how proposed tools or schemes will work in relation to the other steps. In the case of [12], the authors developed a complex process designed to test as many cases as possible, but the test cases their tools generate still must be implemented and maintained by hand, and the tools provide no test harness support at all.

Another expensive error in understanding the testing process is combining steps. This was our main problem with *CaseTool: Scheme One*. We combined the role of test developer and tester. We required testers with as much knowledge as the test developer. Since that knowledge needed to be extensive, we incurred unacceptably high personnel costs. We also combined several steps

in the testing process. This was primarily because we lacked a requirements document, and concrete test suite. This affected the number of tests we performed and the reproducibility of those tests.

Furthermore, many test developers fail to look at the testing process in the context of the larger process of software development. They do not design testing that can be effectively applied in the maintenance phase of the software life cycle. We demonstrate the importance of planning early. In all our test cases, we incurred most of our costs in test development. We developed sound testing plans, implementations and harnesses. We incurred few costs in the area most affected in the maintenance phase: choosing inputs, determining outputs and executing the tests. Since these steps are likely to be revised during regression testing, we expect these tests to incur fewer additional costs than methods that incur high costs in these steps during test development.

We succeeded in developing effective testing for *Scalc*, *CaseTool*, *symtbl*, and *Space*.

Future Work

There are several areas that we believe need further investigation:

1. Complete *Space* testing
2. System testing tools
3. Module testing tools
4. Further case studies

We only partially completed the module testing for *Space*; we would like to see this testing completed.

CaseTool testing is time consuming and utilises no automation. A script recorder/playback testing tool similar to AUTOmate [16] would be very helpful in testing systems that involve bitmapped screen and mouse interfaces.

We developed a scheme for batch testing call-based modules based on the pgmgen scheme. This scheme has proved successful, and usable as it is, but it it only a prototype. ABC has expressed an interest in incorporating testing tools in their *CaseTool* product.

We learned a great deal about the complexity of testing. Keeping records on costs incurred provided insight about where the costs occur in the testing process, but we tested only four pieces of software. Further case studies on different types of software with different controllability and observability characteristics should show that the testing process still applies well, and the schemes developed will expand the base of software we know how to test.

References

[1] ABC Inc, *CaseTool User's Guide: Software Engineering Tools,* Version 1.0, ABC Inc., Vancouver, British Columbia, 1989.

[2] Apple Computer Inc, *Human Interface Guidelines: The Apple Desktop Interface*, Apple Computer Inc., Cupertino, California, 1987

[3] Apple Computer Inc, *Macintosh Programmer's Workshop 3.0 Pascal*, Apple Computer Inc, Cupertino, California, 1988

[4] Apple Computer Inc, *Macintosh Programmer's Workshop 3.0 Reference, Volume 1 and Volume 2*, Apple Computer Inc, Cupertino, California, 1988

[5] Apple Computer Inc., *Macintosh System Software User's Guide,* Version 6, Apple Computer Inc, Cupertino, California, 1988.

[6] Bird, D.L., Munoz, C.U., Automatic generation of random self-checking test cases, *IBM Systems Journal 22*, 3 (1983), 229-245.

[7] Brown, P.A., Hoffman, D., *The Application of Module Regression Testing at TRIUMF*, University of Victoria, B.C., 1989.

[8] Fagan, M.E., Design and Code Inspection to Reduce Errors in Program Development, *IBM Systems Journal,* (1976).

[9] Hoffman, D., A CASE Study in Module Testing, *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society, 1989, 100-105.

[10] Hoffman, D., Practical Interface Specification, *Software - Practice and Experience 19*, 2 (Feb. 1989), 127-148

[11] Hoffman, D., *ABC's Aranda - A Technology Appraisal*, University of Victoria, B.C., 1989.

[12] Ostrand, T.J., Balcer, M.J., The Category-Partition Method for Specifying and Generating Functional Tests, *Communications of the ACM 31*, 6 (June 1988), 676-686.

[13] Ostrand, T.J., Sigal, R., Weyuker, E.J., Design for a Tool to Manage Specification-Based Testing, *IEEE*, (1986), 41-50.

[14] Parnas, D.I., On the Criteria to be used in Decomposing Systems into Modules, *Communications of the ACM 15,* 12 (Dec. 1972), 1053-1058.

[15] Parnas, D.L., Clements, P.C., A Rational Design Process: How and Why to Fake It, *IEEE Transactions on Software Engineering 12*, 2 (Feb. 1986), 251-257.

[16] Rodrigues, N., Tracy, W., Miller, G., AUTOmate: A New Approach to Software Verification, *Pacific Northwest Software Quality Conference Proceedings*, (Sept. 1989), 201-211.

[17] Schmucker, K.J., *Object-Oriented Programming for the Macintosh,* Hayden Book Company, New Jersey, 1986.

[18] Uren, E., Miller, E., Irwin, J., Automated Software Testing - Case Studies, *Proceedings from the 1987 Conference on Software Maintenance*, IEEE, (Sept.1987), 198-201.

Appendix A

System Testing for BB:Scalc

scalc

SCALC Requirements Specification

USER COMMANDS

SCALC runs interactively and performs arithmetic on non-negative integer values. SCALC has a push-down stack for storing values - the top of this stack serves as one operand in, and the destination for, all calculations. Initially, the stack is empty. SCALC prompts the user for input with the string "scalc>" just after SCALC has been invoked and after each user command has been processed.

The following commands are available to the user - each command must appear on a line by itself, with tokens separated by one or more blanks or tabs.

push <int>	pushes <int> on to the top of the stack
pop	removes the top stack element and displays the new stack element or, if the stack is empty, the message Stack is empty
top	displays the value of the top stack element
<operator> <int>	replaces the top stack element with the expression <oldtos> <operator> <int> and displays <newtos>, where <oldtos> and <newtos> are the values of the top stack element before and after execution of the command.
quit	terminates execution of SCALC

WHERE

<int> is is a 1 to 6 digit unsigned integer

<operator> is "p", "m", "t" or "d"
corresponding to plus, minus, times
and divide (truncating), respectively

ERROR HANDLING

If a command generates an error, then the ONLY effect of the command is the error message described below. No more than one error is reported for a given command.

Format errors — If the user input does not conform exactly to the format described above, then print the appropriate message chosen from the three below.

***** Error - first bad token is: <token>
where <token> is the leftmost token in error

***** Error - too few tokens

***** Error - too many tokens

After an input line which contains only blanks or tabs, just reprompt.

Stack empty — If pop, top or an arithmetic command is issued when the stack is empty, issue the following message:

***** Error - stack empty

Stack full — If the push command is issued when the stack already has 10 elements, issue the following message:

***** Error - stack full

Overflow — Overflow occurs when a calculation is requested that would result in a value larger than 6 digits, generating the following message:

***** Error - arithmetic overflow

Underflow — Underflow occurs when a calculation is requested that would result in a negative value, generating the following message:

***** Error - arithmetic underflow

Zero divide — Zero divide occurs when a division is requested with denominator zero, generating the following message:

***** Error - zero divide

scalc.tplan

Test Plan Scalc

Strategy
- Critical system state values
  - stack: empty, mid size, and max size
- Exception
  - generate unknown command message
  - for each command
    - generate all appropriate error messages
- Normal
  - for each critical system state

perform each allowable command once

Implementation

no stubs
directory structure:
scalc/
input/ - test cases stored one per file
naming convention:
<cmd>n: normal behaviour for <cmd>
<cmd>[1-9]: error behaviour for <cmd>
exp/ - expected result of test case (same file name)
act/ - actual results of test case (same file name)
driver: in Makefile
for each file f in input/
run: scalc < f > act/f
run: compare act/f exp/f

Appendix B

System Testing for CaseTool: Scheme One

Test Inventory

Test Inventory

Version: Test 0.19 **Date:** Feb., 16, 1990.

Introduction:

This document lists the items in CaseTool under test.

Inventory:

1. Standards document
2. CaseTool
   - Apple CaseTool
   - Start
   - Quit
3. Documents
   - New
   - Open
   - Close
   - Save and save as
4. Projects
   - Palette
   - Get source files
   - Define note forms
   - Forms
   - Folder
5. Tools
   - Contents
   - Classes
   - Imports
   - Used By
   - Modified By
   - Source Code
   - Flow Chart
   - Notes
7. Display
   - Enlarge and Reduce
   - Hide and Show

Filter
Display Selection

Master Test Plan

Master Test Plan

Version: Test 0.19 **Date:** Feb., 16, 1990.

Strategy:

Assumptions

The tester is familiar with the CaseTool requirements documents:
Test Inventory
CaseTool Standards
Detailed Inventory

Critical System States Values

Documents:

Documents can be formed through a cross product of their possible contents. Documents are composed of Reports, and Projects(Files, Folders, Forms, Palette).

Project

	Files	Folders	Forms	Palette
0	no files	none	default	none
1	interface file	one	blank	1
2	implementation file	two folders	1 palette object	MaxPalette/2 objects
3	interface & impl file	two deep	1 tool	MaxPalette obj.
4	program file	two x two	MaxPalette & MaxMoreTools	
5	many files			
6	1 non-Pascal file			
7	1 incorrect Pascal file			
8	1 empty file			

Denote a test document as a function T with 4 arguements where each parameter in T refers to files, folders, forms, and palette respectively and the value of the parm to the row position; e.g., T(1,0,0,3) is a test document that has one interface file, no folders, default forms, and all of the palette objects.

Tools:

each tool once on a valid identifier
each tool on each possible identifier

Display:

Filter:
no boxes checked, each box only checked once, all boxes checked

Hide and Show:
non-leaf labels, non-leaf identifiers

Enlarge and Reduce:
min size, normal - 1 size, normal, normal + 1 size, max size

Test Case Selection Strategy

Touch test; guarantee that each critical value will be touched at least once during testing.

For each testing session, select a subset of the documents table cross product, S. Selections are made based on the following criteria

1. Time
   keep testing to 4 hrs
2. Historical awareness of error troubles and testability
   emphasis on tools, palette, get source files
   de-emphasis folders
3. Requests from ABC Inc to test emphasis
   corresponds with historical error troubles

Exception

For each possible exception X
    generate X at least once

Normal

/* Create all the documents in the test set */

Documents

```
For each document D in S do
    Start
    New
Projects
    Make the new document have the same state as D
    Save as D
    Quit
```

/* Complete testing the remaining Critical Values */

Tools

```
For each saved D in S do
    Start
    Open D'
    For each critical value I in Identifiers do
        For each critical value T in Tools do
            If T can be applied to I then Do T
    Save
    Quit
```

Display

```
For each D in S do
    Start
    Open D
    For each report R in D do
        For each critical value F in Filter do
            Apply Filter with F
        For each critical value H in Hide and Show do
```

```
                                        Apply Hide and Show with H
                                    For each critical value E in Enlarge and reduce do
                                        Apply Enlarge and Reduce with E
                            Save
                            Quit

        /* Palette */

Projects/Palette    Start
                    For each palette item P in PaletteMenu do
                            Select P
                            Resize P from all 4 blips, and off edges of window
                            Move P in all directions and off edges of window
                            Apply each Edit menu command to P
                    If possible to group objects then
                            Group objects
                            Resize as above
                            Move as above
                            Apply each Edit menu command, as above
                    Quit

        /* Save */

Document            For each D in S do
                            Open D
                            View each possible view in D
                            Close
```

Implementation:

Test disks:

HARD DISK on the SE in the software testing lab
Almost Full floppy disk

Test source files in:

HARD DISK:CaseTool Testing:bb:

sysdef:	
sysdef.p	interface & implementation, no uses
symtbl:	
Makefile	
symtbl.p	interface
symtblImpl.p	implementation
symtblExc.p	interface & implementation
symtblTest.p	program

HARD DISK:CaseTool Testing:

empty.p
syntax.error.p

By hand and visual inspection:

Preconditions:
MacroMaker installed in the system folder

```
        Desktop has CaseTool Testing folder open and current
                Folder window is zoomed out with upper left zoom out box
                Window is in upper left most scrolled position
        MacroMaker opened to Testing Macros

    /* Create all the documents in the test set */
    For each document D in the test set do
        if the document creation has been successfully made into a macro
                select "build D" from macro menu
        else
                build D by hand
                save D
        quit CaseTool and return to desktop

    For each step S in the Test Case selection strategy
        Perform steps as required to attain the "Before" conditions in the
                requirements for S.
        Perform S; Confirm "During" conditions hold
        Confirm "After" conditions hold
```

Weekly Test Plan

Weekly Test Plan

Version: Test 0.19 **Date:** Feb., 16, 1990.

Strategy:

Assumptions

The tester is familiar with the CaseTool requirements documents:
Master Test Plan

Test Case Selection Strategy

Touch test; guarantee that each critical value will be touched at least once during testing.

Choose a set of items from the test inventory and test those according to the applicable parts of the Master Test Plan.

All test documents will exercise CaseTool and Documents sections from the Test Inventory (Start, Quit, New, Open, Save, & Close).

Test Documents	Test Features	Estimated Time (min)
T(0,0,0,0)	Apple CaseTool	5
T(0,0,0,3)	Palette	15
T(1,0,0,1)	Projects, Tools, Palette	15
T(2,0,1,0)	Get Source Files, Tools, Define Forms, Forms	45
T(3,0,4,0)	Get Source Files, Tools, Define Forms, Forms, Palette	45
T(4,0,0,0)	Get Source Files, Tools	45
T(5,0,0,0)	Get Source Files, Tools	45
T(6,0,0,0)	Get Source Files	5
T(7,0,0,0)	Get Source Files	5

T(8,0,0,0)	Get Source Files	5
	Total	230
		(3:50 hr)

Implementation:

Implement the test case selection strategy for each of the test documents list above using the implementation strategy in the Master Test Plan.

Appendix C

System Testing for CaseTool: Scheme Two

S Plan

Summary
Test Plan

CaseTool Version: ß1.2 **Date:** April 13, 1990

Strategy:

The test plan is divided into three smaller test plans; each category allows people with different levels of knowledge to participate in testing. No test appears in more than one of the smaller test plans although some steps may need to be repeated in order to achieve a different test. Testers are classified based on their assumed knowledge in the following areas:

1) Macintosh operating system
2) Macintosh operating system, & CaseTool
3) Macintosh operating system, CaseTool & Pascal

A tester in each of the three smaller test plans is expected to complete partial testing in one hour, and full testing in 4 hours. The overall goal in these test plans is "touch testing". We try to exercise each CaseTool feature at least once.

Implementation:

Test configuration:

Macintosh SE or higher with:

- 1 Mb RAM or more,
- either hard disk or blank formatted floppy disk in drive,
- printing facilities,
- CaseTool installed on disk.

Forms:

For each Test there are a set of Test Forms: a Test Plan, a Partial Test Suite and a Full Test Suite. Discrepancies between expected and actual output are recorded in the Error Report. For each Goal in the Test Plan there are one or more tables in the Test Suite. The Goal is repeated in the Suite and the Goals are in the same sequence in the Suite and the Plan. Tables are grouped to take approximately 10-20 min per group to perform.

For Partial Testing:

Perform the events on the Partial Test Suite form.

Work tables one row at a time left to right

If the results of the event are as expected

Mark empty boxes with an √

else

Mark empty boxes with an F
Record unexpected behaviour on the Error Report

Continue testing
If CaseTool must be restarted then
    Restart CaseTool
    Redo all events of the most recent Group
    omitting any events that failed and continue.
else Continue testing

For Full Testing:

1. Perform Partial Testing.
2. Work the Full Test Suite form the same way as Partial Testing.

M Plan

Macintosh
Test Plan

CaseTool Version: ß1.2 **Date:** March 1, 1990

Strategy:

Assumptions

The tester is familiar with:

- the operation of an Apple Macintosh
- typical Macintosh drawing applications like MacDraw
- the method for printing documents on the test machine

References

Apple Computer Inc., "Macintosh System Software User's Guide", Version 6, Apple Computer Inc, Cupertino, California, 1988.

ABC Inc, "CaseTool User's Guide: Software Engineering Tools", Version 1.0, ABC Inc., Vancouver, British Columbia, 1989, p. 8, p.63.

Test Case Selection Strategy

All time estimates are in minutes.

Normal Behaviour

Goal	Stimulus	Partial	Full
Start CaseTool	From the application icon	1	1
	From an CaseTool document		1
Apple menu	Select each menu item		10
File menu	*New* (repeated until failure)		10
	Save	1	1
	Save as	5	5
	Open	1	1
	Close	1	1
	Print		10
	Page Setup		1
	Quit	5	5

Windows	Resize using resize box	1	1
	Zoom using zoom box	1	1
	Scroll window	5	5
	Move window	1	1
	Close using close box	1	1
	Change active window	1	1
Palette	Draw each palette object	1	1
	Select single object	1	1
	Select multiple objects	1	1
	Resize objects	1	1
	Move objects	1	1
Edit on selected palette objects	*Cut*	1	1
	Copy	1	1
	Paste	1	1
	Clear	1	1
	Select All	1	1
	Group objects	1	1
Grouped Objects	Select as for palette objects		1
	Resize as for palette objects		1
	Move as for palette objects		1
	Edit menu as for palette objects		10
Text editing	Insert text	5	5
	Select portion of text within text object	5	5
	Edit menu for selected text	10	10
Edit/Undo	*Undo* on palette functions		10
	Undo on Edit menu functions		10
	Undo on grouped objects		10
	Undo on text objects		10
	Total	50	140

Exceptional Behaviour

Goal	Stimulus	Partial	Full
File menu	Create a new document after maximum created		10
	Save as' document that already exists		10
	Save as' to disk without enough space		10
	Open an already open document	1	1
	Close a document with changes	1	1
	Total	5	35

Implementation:

Test configuration:

"Almost Full" floppy disk containing:

filler:

junk files to take up all but 5 k of disk space

Forms:

Macintosh Partial Test Suite
Macintosh Full Test Suite
Error Report

MC Plan

Macintosh & CaseTool
Test Plan

CaseTool Version: ß1.2 **Date:** April 6, 1990

Strategy:

Assumptions

The tester is familiar with:

- the operation of an Apple Macintosh
- typical Macintosh drawing applications like MacDraw
- the method for printing documents on the test machine
- section 1 & 2 of the CaseTool User's Guide

The tester will be expected to apply tools and determine if the resulting report is of the correct kind (e.g., applying the Contents tool generates a Contents report), but will not need to evaluate the report to determine if the information in the report is correct.

References

Apple Computer Inc., "Macintosh System Software User's Guide", Version 6, Apple Computer Inc, Cupertino, California, 1988.

ABC Inc., "CaseTool User's Guide: Software Engineering Tools", Version 1.0, ABC Inc., Vancouver, British Columbia, 1989.

Test Case Selection Strategy

All time estimates are in minutes.

Normal Behaviour

Goal	Stimulus	Partial	Full
Project/ Get Source Files	Get one file Get multiple files Get files from different folders	1	1 5 5
File menu	*Save* and *Open* with files in project		5
Project/ Create Folder	Create a folder at Home folder Create a folder inside a non-Home folder Create two folders at the same folder		1 1 1
Project/ Find	Find an identifier in the project		2
Project/ Define Note Form	Select menu item		1

Tools menu	*Contents*	2	2
	Classes	2	2
	Used by	2	2
	Imports	2	2
	Modified by	2	2
	Source Code	2	2
	Flow Chart	2	2
	Notes	2	2
Source File Icons	Rename		2
	Move into folder		2
	Edit		5
Folder Icon	Rename		2
	Project/Open		2
	Move folder into another folder		2
	Navigate between folders		5
	Edit		5
Report Icons	Rename		2
	Project/Open	2	2
	Move into Folder		2
	Edit		2
Define Note Form Icons	Rename		2
	Project/Open		2
	Navigate between icons		2
Define Note Form Icon contents	Remove all objects from default		2
	Add all possible objects		5
Display menu	*Display selection*	5	5
	Filter		10
	Expand	2	2
	Reduce	2	2
	Page Breaks		1
	Clean up Window	2	2
Other Display Features	Hide	2	2
	Show	2	2
	Show More	5	5
Navigation palette	Home	5	5
	Up		5
	Previous	5	5
	Total	44	112

Exceptional Behaviour

Goal	Stimulus	Partial	Full
Project/ Get Source Files	Get empty source code file	1	1
	Get non-source code file	1	1
	Get source code file with syntax error	1	1
	Get files already in project	2	2
	Get mix of good and bad files		5
Project/ Find	Find identifier not in project		2
	Find identifier in project twice		2

Project/ Define Note Form	Create Folder		2
Tools menu	Identifier definition missing for Contents Identifier definition missing for Classes Identifier definition missing for Imports Identifier definition missing for Source Code Identifier definition missing for Flow Chart		2 2 2 2 2
Source File Icons	Rename with name that is too long *Project/Open*		2
Define Note Form icons	*Edit*		5
Define Note Form contents	*Edit/Cut, Edit/Copy*		5
Display menu	*Expand* source code report *Reduce* source code report		2 2
Other Display Features	Hide a leaf node in a report tree Hide any word in a source code report Show More in a source code report	1 1 1	
	Total	13	51

Implementation:

Test configuration:

"Almost Full" floppy disk containing:

filler:

junk files to take up all but 5 k of disk space

"Test Data" floppy disk containing test files:

symtbl:

symtbl.p
symtblImpl.p
symtblExc.p
symtblTest.p
Makefile

sysdef:

sysdef.p
Makefile

empty.p
syntaxError.p

Forms:

Macintosh & CaseTool Partial Test Suite
Macintosh & CaseTool Full Test Suite
Error Report

MCP Plan

Macintosh CaseTool & Pascal Test Plan

CaseTool Version: ß1.2 **Date:** April 13, 1990

Strategy:

Assumptions

The tester is familiar with:

- the operation of an Apple Macintosh
- typical Macintosh drawing applications like MacDraw
- the method for printing documents on the test machine
- section 1 & 2 of the CaseTool User's Guide
- the contents of the Pascal source files listed below

The tester will be expected to determine if the information in the report is correct.

References

Apple Computer Inc., "Macintosh System Software User's Guide", Version 6, Apple Computer Inc, Cupertino, California, 1988.

ABC Inc., "CaseTool User's Guide: Software Engineering Tools", Version 1.0, ABC Inc., Vancouver, British Columbia, 1989.

Test Case Selection Strategy

All time estimates are in minutes.

Normal Behaviour

Goal	Stimulus	Partial	Full
Project/ Define Note Forms	For each note form icon do *Open* the icon Create forms using one of each possible tools and object		10
File Icon or Identifier	*Contents*	2	2
Program Identifiers	*Content*	2	2
	Imports	2	2
	Source Code	2	2
	Flow Chart	2	2
	Notes	2	4
Unit Identifiers	*Contents*		2
	Used by		2
	Imports		2
	Source Code		2
	Notes		2

Object Identifiers	*Classes*	2	2
	Used by		2
	Source Code		2
	Notes		2
Type Identifiers (system, user - defined)	*Used by*	2	4
	Source Code	2	4
	Notes	2	8
Constant Identifiers	*Used by*	2	2
	Source Code	2	2
	Notes	2	4
Routine Identifiers (Method, Procedure & Function)	*Used by*	2	6
	Imports	2	6
	Source Code	2	6
	Flow Chart	2	6
	Notes	2	12
Variable Identifiers (global, local, instance, formal parameters)	*Used by*	2	6
	Modified by	2	6
	Source Code	2	6
	Notes	2	12
	Total	44	132

Exceptional Behaviour

Goal	Stimulus	Partial	Full
File Icon or Identifiers	*Classes*	1	1
	Used by		1
	Imports		1
	Modified by		1
	Source Code		1
	Flow Chart		1
	Notes		1
Program Identifiers	*Classes*	1	1
	Modified by		1
Unit Identifiers	*Classes*		1
	Modified by		1
	Flow Chart		1
Object Identifiers	*Contents*	1	1
	Imports		1
	Modified by		1
	Flow Chart		1
Type Identifiers	*Contents*	1	1
	Classes		1
	Imports		1
	Modified by		1
	Source Code on system types		1
	Flow Chart		
Constant Identifiers	*Contents*		1
	Classes		1
	Imports		1
	Modified by	1	1
	Flow Chart		1

Routine Identifiers	*Contents*	1	1
	Classes		1
	Modified by (except functions)		1
Variable Identifiers	*Contents*		1
	Classes		1
	Imports	1	1
	Flow Chart		1
	Total	51	33

Implementation:

Test configuration:
"Test Data" floppy disk containing test files:
symtbl:
symtbl.p
symtblImpl.p
symtblExc.p
symtblTest.p
Makefile
sysdef:
sysdef.p
Makefile
empty.p
syntaxError.p

Forms:
Macintosh CaseTool & Pascal Partial Test Suite
Macintosh CaseTool & Pascal Full Test Suite
Error Report

M PSuite

Macintosh Partial Test Suite

Version: **Date:**

Tester: **Time Start:** **Time End:**

Machine: **# Failures Found:**

Normal Behaviour

Group: 1

Goal: Start CaseTool
Table: 1

Row	Event	Success (√) or Failure (F)
1	Double Click on CaseTool icon	

CaseTool should start up and the resulting picture on the screen should be the same as the picture in Figure 2-1 on page 8 of "CaseTool User's Guide."

Goal: *File* menu
Table: 2

Row	Event	Success (√) or Failure (F)
1	*Save as* T1	
2	Draw a rectangle	
3	*Save* T1	
4	*Close* T1	
5	*Quit*	
6	Start CaseTool (see above)	
7	*Open* T1	

Goal: Windows
Table: 3

Row	Event	Success (√) or Failure (F)
1	Resize window larger	
2	Resize window smaller	
3	Zoom window	
4	Move window	
5	*File/New*	
6	Close using close box	
7	Change Active window	
8	Change Active window back to T1	

Table: 4
It may be necessary to resize window to make palette scroll bar active.

		Event (scroll using)		
	Col	1	2	3
Row	Scroll	Arrows	Box	Bar
1	Vertical			
2	Horizontal			

Table: 5

Row	Event	Success (√) or Failure (F)
1	*File/Close* T1	

Group: 2

Goal: Palette
Table: 6

It will be easier to perform the remaining tests if the objects are not too large (<= 3 inches across) and spaced close together, but not overlapping.

		Event	
		Draw	
	Col	1	2
Row	Object	Entirely within the visible window area	Finish outside the visible window area
1	Text		
2	Line		
3	Rectangle		
4	Sausage		
5	Shadow Box		
6	Ellipse		

Table: 7

		Event	
		Select Single Object	
	Col	1	2
Row	Object	Clicking	Dragging
1	Text		
2	Line		
3	Rectangle		
4	Sausage		
5	Shadow Box		
6	Ellipse		

Table: 8

Row	Event	Success (√) or Failure (F)
1	*File/Save as* T2	
2	*File/Close*	

Group: 3

Table: 9

Row	Event	Success (√) or Failure (F)
1	*File/Open* T2	

Table: 10

It may be necessary to de-select some objects in order to perform the tasks in each column.

Event
Select Multiple Objects

	Col	1	2
Row	Object	Shift Clicking	Dragging
1	Text		
2	Text & Line		
3	Text, Line & Rectangle		
4	Text, Line, Rectangle & Sausage		
5	Text, Line, Rectangle, Sausage & Shadow Box		
6	Text, Line, Rectangle, Sausage, Shadow Box & Ellipse		

Table: 11

		Event			
		Resize			
	Col	1	2	3	4
Row	Object	Top Left Blip	Top Right Blip	Bottom Left Blip	Bottom Right Blip
1	Text				
2	Line			N/A	N/A
3	Rectangle				
4	Sausage				
5	Shadow Box				
6	Ellipse				

Table: 12

		Event	
		Move	
	Col	1	2
Row	Object	Entirely within the visible window area	Finish outside the visible window area
1	Text		
2	Line		
3	Rectangle		
4	Sausage		
5	Shadow Box		
6	Ellipse		

Table: 13

Row	Event	Success (√) or Failure (F)
1	*File/Save*	
2	*File/Close*	

Group: 4

Goal: *Edit* on selected palette objects
Table: 14

Row	Event	Success (√) or Failure (F)
1	*File/Open* T2	

Table: 15

		Event				
	Col	1	2	3	4	5
Row	Object	*Cut*	*Paste*	*Copy*	*Paste*	*Clear*
1	Text					
2	Line					
3	Rectangle					
4	Sausage					
5	Shadow Box					
6	Ellipse					

Table: 16

Row	Event	Success (√) or Failure (F)
1	*Select All*	
2	*Group*	
3	*Ungroup*	
4	*Select All*	
5	*Clear*	

Table: 17

Row	Event	Success (√) or Failure (F)
1	*File/Save*	
2	*File/Close*	

Group: 5

Goal: Text editing
Table: 18

Row	Event	Success (√) or Failure (F)
1	*File/New*	

Table: 19

Row	Event	Success (√) or Failure (F)
1	Draw text object	

2	Insert typewriter printable character 2 numbers 2 letters 2 other symbols	
3	Insert text at beginning of object	
4	Insert text at end of object	
5	Insert text in the middle of object	
6	Select text within object (double click)	
7	Select text within object (dragging)	
8	*Cut*	
9	*Paste*	
10	*Copy*	
11	*Paste*	
12	*Clear*	
13	*File/Quit* (do **not** save changes)	

Exceptional Behaviour

Group: 6

Goal: *File* menu
Table: 20

Row	Event	Success (√) or Failure (F)
1	Start CaseTool	
2	*Open* T1	
3	*Open* T1	
4	*Quit*	

MC PSuite

Macintosh & CaseTool Partial Test Suite

Version: **Date:**

Tester: **Time Start:** **Time End:**

Machine: **# Failures Found:**

Normal Behaviour

Group: 1

Goal: *Project/Get Source Files*
Table: 1

Row	Event	Success (√) or Failure (F)
1	Double Click on CaseTool icon	
2	*Project/Get Source Files*	
3	Change to Test Data:symtbl	
4	Add symtbllTest.p	
5	Done	

Goal: *Tools* menu
Table: 2

For each report generated, scroll to the top right corner, the bottom right corner, the bottom left corner and back to the top left corner using the scroll arrows. You may need to scroll the window in order to locate some of the identifiers.

Row	Event	Success (√) or Failure (F)
1	*Contents*	
2	*Classes* on selected TSymtblTest	
3	Previous	
4	*Used by* on selected nextcall	
5	Previous	
6	*Imports* on selected nextcall	
7	Previous	
8	*Modified by* on selected reply	
9	Previous	
10	*Source Code* on selected symtbllTest	
11	Previous	
12	*Flow Chart* on selected symtbllTest	
13	Previous	
14	*Notes* TSymtblTest	
15	Home	
16	Is Contents symtbllTest.p icon visible ?	
17	Is Classes TSymtblTest icon visible ?	
18	Is Used by nextcall icon visible ?	
19	Is Imports nextcall icon visible ?	
20	Is Modified by reply icon visible ?	
21	Is Source Code symtbllTest icon visible ?	
22	Is Flow Chart symtbllTest icon visible ?	
23	Is Notes TSymtblTest icon visible ?	
24	*File/Close* (do **not** save changes)	

Group: 2

Goal: Report Icons
Table: 3

Row	Event	Success (√) or Failure (F)
1	*File/New*	

2	*Project/Get Source Files* symtbllTest.p	
3	*Tools/Contents*	
4	Home	
5	*Project/Open*	

Goal: *Display* menu
Table: 4

Row	Event	Success (√) or Failure (F)
1	*Display Selection* on selected Type	
2	Scroll to the right most side of report	
3	Select gDump	
4	Scroll to the left most side of report	
5	*Display selection*	
6	Scroll to top left most corner of report	

Table: 5
Use scrolling to ensure that some part of the graphic is always in view.

Row	Event	Success (√) or Failure (F)
1	*Expand* until report does not get larger	
2	*Reduce* until report does not get smaller	
3	*Enlarge* until report is legible	
4	Home	

Table: 6

Row	Event	Success (√) or Failure (F)
1	*Clean Up Window*	
2	*Close* (do **not** save changes)	
3	*File/New*	
4	*Project/Get Source Files*	
5	Get symtbl.p	
6	*Clean Up Window*	
7	*Project/Get Source Files*	
8	Get symtbllmpl.p	
9	*Clean Up Window*	
10	*File/Close* (do **not** save changes)	

Group: 3

Table: 7

Row	Event	Success (√) or Failure (F)
1	*File/New*	
2	*Project/Get Source Files*	
3	Get symtbllTest.p	

4	*Tools/Contents*	
5	*Hide* subtree below symtbllTest	
6	*Show* subtree below symtbllTest	
7	*Hide* subtree below TSymtblTest	
8	*Hide* subtree below Type	
9	*Show* subtree below Type	
1 0	*Show* subtree below TSymtbTest	

Table: 8

Row	Event	Success (√) or Failure (F)
1	*Tools/Flow Chart* on selected symtbllTest	
2	*Hide* subtree below symtbllTest	
3	*Show* subtree below symbtbllTest	
4	*Hide* subtree below CASE reply	
5	*Hide* subtree below WHILE	
6	*Show* subtree below WHILE	
7	*Show* subtree below CASE reply	
8	Previous	

Table: 9

Row	Event	Success (√) or Failure (F)
1	Show More symtbllTest	
2	Show More CONST	
3	Show More quit	
4	*Tools/Flow Chart* on symtbllTest	
5	Show More While ...	
6	Show More CASE reply	
7	Show More quit	
8	Show More reply := ...	

Table: 1 0

Normal Behaviour testing has been completed.

Row	Event	Success (√) or Failure (F)
1	*File/Close* (do **not** save changes)	

Exceptional Behaviour

Group: 4

Goal: *Project/Get Source Files*

Table: 1 1

Row	Event	Success (√) or Failure (F)
1	*File/New*	
2	*Project/Get Source Files* empty.p	

3	*Project/Get Source Files*	
4	Click on the .p to display non-.p files	
5	Get symtbl:Makefile	
6	Get syntaxError.p	
7	Get symtbl:symtbllTest.p	
8	Get symtbl:symtbllTest.p	

Table: 12

Row	Event	Success (√) or Failure (F)
1	*Contents* on symtbllTest.p	
2	*Hide* reply	
3	*Source Code* gDump	
4	*Hide* subtree below gDump	
5	*Show* More on gDump	
6	*File/Quit* (do **not** save changes)	

MCP PSuite

Macintosh CaseTool & Pascal Partial Test Suite

Version: **Date:**

Tester: **Time Start:** **Time End:**

Machine: **# Failures Found:**

Normal Behaviour

For each report generated, scroll to the top right corner, the bottom right corner, the bottom left corner and back to the top left corner using the scroll arrows. You may need to scroll the window in order to locate some of the identifiers.

Group: 1

Goal: File Icon or Identifier
Table: 1

Row	Event	Success (√) or Failure (F)
1	Start CaseTool	
2	*Project/Get Source Files* symtbl.p	
3	*Project/Get Source Files* symtbllmpl.p	
4	*Tools/Contents* on symtbl.p	
5	*Tools/Contents* on Include file symtbllmpl.p	

6	Home	
7	*File/Close* (do **not** save changes)	

Goal: Program Identifiers
Table: 2

Row	Event	Success (√) or Failure (F)
1	*File/New*	
2	*Project/Get Source Files* symtbllTest.p	
3	*Tools/Contents* on symtbllTest.p	
4	*Tools/Imports* on symtblTest	
5	*Tools/Source Code* on symtbllTest	
6	*Tools/Flow Chart* on symtbllTest	
7	*Tools/Notes* on symtbllTest	
8	Home	
9	*File/Close* (do **not** save changes)	

Group: 2

Goal: Object Identifiers
Table: 3

Row	Event	Success (√) or Failure (F)
1	*File/New*	
2	*Project/Get Source Files* symtbl.p & symtbllTest.p	
3	*Tools/Contents* on symtbl.p	
4	*Tools/Classes* on TSymtbl	
5	Previous	

Goal: Type Identifiers
Table: 4

Row	Event	Success (√) or Failure (F)
1	*Tools/Used by* on stTTbl	
2	*Tools/Source Code* on stTTbl	
3	*Tools/Notes* on stTTbl	
4	Home	
5	*File/Close* (do **not** save changes)	

Group: 3

Goal: Constant Identifiers
Table: 5

Row	Event	Success (√) or Failure (F)

1	*File/New*	
2	*Project/Get Source Files* symtbl.p & symtblImpl.p	
3	*Tools/Contents* symtbl.p	
4	*Tools/Used by* on stMaxSyms	
5	*Tools/Source Code* on stMaxSyms	
6	*Tools/Notes* on stMaxSyms	
7	Home	
8	*File/Close* (do **not** save changes)	

Group: 4

Goal: Routine Identifiers
Table: 6

Row	Event	Success (√) or Failure (F)
1	*File/New*	
2	*Project/Get Source Files* symtbl.p symtblImpl.p & symtblTest.p	
3	*Tools/Contents* symtbl.p	
4	*Tools/Used by* on sAddsym	
5	*Tools/Imports* on sAddsym	
6	*Tools/Source Code* on sAddsym	
7	*Tools/Flow Chart* on sAddsym	
8	*Tools/Notes* on sAddsym	
9	Home	
10	*File/Close* (do **not** save changes)	

Group: 5

Goal: Variable Identifiers
Table: 7

Row	Event	Success (√) or Failure (F)
1	*File/New*	
2	*Project/Get Source Files* symtbl.p symtblImpl.p & symtblTest.p	
3	*Tools/Contents* on symtbl.p	
4	*Tools/Used by* on fTblCnt	
5	*Tools/Modified by* on fTblCnt	
6	*Tools/Source Code* on fTblCnt	
7	*Tools/Notes* on fTblCnt	
8	Home	
9	*File/Close* (do **not** save changes)	

Exceptional Behaviour

Group: 6

Goal: File Icon or Identifiers
Table: 8

Row	Event	Success (√) or Failure (F)
1	*File/New*	
2	*Project/Get Source Files* symtbl.p symtbllmpl.p & symtbllTest.p	
3	*Tools/Contents* on symtbl.p	
4	*Tools/Source Code* on symtbl.p	
5	*Tools/Classes* on symtbllmpl.p	
6	Home	

Goal: Program Identifiers
Table: 9

Row	Event	Success (√) or Failure (F)
1	*Tools/Contents* on symtbllTest.p	
2	*Tools/Source Code* on symtbllTest	
3	*Tools/Classes* on symtbllTest	
4	Home	

Goal: Object Identifiers
Table: 10

Row	Event	Success (√) or Failure (F)
1	*Project/Open* on Source Code symtbl	
2	*Tools/Contents* on TSymtbl	
3	Home	

Goal: Type Identifiers
Table: 11

Row	Event	Success (√) or Failure (F)
1	*Project/Open* on Source Code symtbl	
2	*Tools/Classes* on stTTbl	
3	Home	

Goal: Constant Identifiers
Table: 12

Row	Event	Success (√) or Failure (F)
1	*Tools/Modified by* on stMaxSyms	

Goal: Routine Identifiers
Table: 13

Row	Event	Success (√) or Failure (F)
1	*Tools/Contents* on sAddsym	

Goal: Variable Identifiers
Table: 13

Row	Event	Success (√) or Failure (F)
1	*Tools/Imports* on fTbl	
2	Home	
3	*File/Close* (do **not** save changes)	

Error Report

Error Report

Version: **Date:**

Tester:

Test Suite (P/F): **Table #:** **Row: #** **Column #:**
Behaviour:

Test Suite (P/F): **Table #:** **Row: #** **Column #:**
Behaviour:

Appendix D

A Scheme for Module Testing

BTest.intspec

Batch Test (BTest) Interface Specification Semantics

ASSUMPTIONS

Batch Test uses Batch Exception and SymtblBExc.p and expects XBExc.p to implement XExc unit by supplying constants [1,numExc] and implement the exception handlers as calls to beSExc using one of the constants in [1, numExc] for each exception handler.

STATE VARIABLES

```
tCaseCount, excErrCount, valErrCount : int
tCaseLabel : string
```

SET CALL EFFECTS

```
sInit:
        tCaseCount = 0
        excErrCount = 0
        valErrCount = 0

sTCase(label):
        tCaseCount = tcaseCount + 1
        tCaseLabel = label

sValErr(val):
        valErr = val

sExcErr(val):
        excErr = val

sCheckExc(expExc):
        if not(gCheckExc(expExc)) then gPrtExc(expExc)

sCheckT(expExc,expVal, actVal):
        if not(gCheckExc(expExc)) then
                gPrtExc(expExc)
        else
                if not(expVal = actVal) then
                        gPrtT(expVal,actVal)
        where T in [Int,Bool,Str]
```

GET CALL RETURN VALUES

```
gTCase = tCaseLabel

gValErr = valErr

gExcErr = excErr

gCheckExc(expExc) =
        if expExc = DontCare then
                true
        if expExc = NoExc then
                if beGExc(expExc) then
                        false
                else true
        if (beGCnt > 0) or (not(beGExc(expExc)) then
                false
        else
                true

gPrtExc(expExc):
        excErrCount = excErrCount + 1
        print tCaseLabel
        (forall  i)  (
                if i in [1, beGNumExc] then begin
                        if ((i <> expExc)  and  beGExc(i)) then
                                print unexpected exception stGName(i)
                        else if ((i = expExc)  and  not(beGExc(i)) ) then
                                print expected exception stGName(i)
        )

gPrtT(expVal,actVal):
        valErrCount = valErrCount + 1
        print an T value fault and tCaseLabel
        where  T  in  [Int,Bool,Str]

gSum:
        print the statistics accumalated including tCaseCnt, expErrCnt,
        and valErrCnt

CONSTANTS

NoExc = 0
DontCare = -1
```

BTest.p

```
UNIT bTest;

        INTERFACE

                USES BExc, {$U symtblBExc.p} symtblExc;

                CONST
```

```
        btNoExc = 0;
        btDontCare = - 1;

    PROCEDURE btSInit;

    PROCEDURE btSTCase(lab: string);
    FUNCTION btGTCase: string;

    PROCEDURE btSValErr(newValErr: integer);
    FUNCTION btGValErr: integer;

    PROCEDURE btSExcErr(newExcErr: integer);
    FUNCTION btGExcErr: integer;

    PROCEDURE btSCheckExc(expExc: integer);
    FUNCTION btGCheckExc(expExc: integer): boolean;

    PROCEDURE btSCheckBool(expExc: integer;
                           expVal,actVal: boolean);
    PROCEDURE btSCheckInt(expExc: integer;
                           expVal,actVal: integer);
    PROCEDURE btSCheckStr(expExc: integer;
                           expVal,actVal: string);

    PROCEDURE btGPrtExc(expExc: integer);
    PROCEDURE btGPrtBool(expVal,actVal: boolean);
    PROCEDURE btGPrtInt(expVal,actVal: integer);
    PROCEDURE btGPrtStr(expVal,actVal: string);

    PROCEDURE btGSum;

IMPLEMENTATION

    VAR
        tCase,excErr,valErr: integer;
        tCaseLabel: string;

        { local functions }

    PROCEDURE prtInt(expVal,actVal: integer);
        BEGIN
            valErr := valErr + 1;
            writeln('***** Value error, ',tCaseLabel,'.');
            writeln('   Expected value:',expVal,'  Actual
value:',actVal);
        END;

    PROCEDURE prtBool(expVal,actVal: boolean);
        BEGIN
            valErr := valErr + 1;
            writeln('***** Value error, ',tCaseLabel,'.');
            writeln('   Expected value:',expVal,'  Actual
value:',actVal);
        END;
```

```
        PROCEDURE prtStr(expVal,actVal: string);
            BEGIN
                valErr := valErr + 1;
                writeln('*****  Value  error,  ',tCaseLabel,'.');
                writeln('   Expected value:',expVal,'  Actual
value:',actVal);
            END;

        PROCEDURE prtExc(expExc: integer);
            VAR
                i: integer;
            BEGIN
                excErr := excErr + 1;
                writeln('*****  Exception  error,  ',tCaseLabel,'.');
                FOR i := 1 TO beGNumExc DO
                    BEGIN
                        IF ((i <> expExc) AND beGExc(i)) THEN
                            writeln('   Unexpected exception
',stGName(i),'  occurred.')
                        ELSE IF ((i = expExc) AND NOT
(beGExc(i)))  THEN
                            writeln('   Expected exception
',stGName(i),' did not occur.');
                    END;
            END;

        FUNCTION foundExc(expExc: integer): boolean;
            BEGIN
                CASE expExc OF
                    btDontCare:
                        foundExc := true;
                    btNoExc:
                        IF (beGCount <> 0) THEN
                            foundExc := false
                        ELSE
                            foundExc := true;
                    OTHERWISE
                        BEGIN
                            IF ((NOT beGExc(expExc)) OR
(beGCount <> 1)) THEN
                                foundExc := false
                            ELSE
                                foundExc := true;
                        END;
                END;
            END;

        { access programs }

        PROCEDURE btSInit;
            BEGIN
                tCase := 0;
                excErr := 0;
```

```
                valErr := 0;
                tCaseLabel := '';
                beSInit(numExc);
        END;

PROCEDURE btSTCase;
        BEGIN
                tCase := tCase + 1;
                tCaseLabel := lab;
                beSClear;
        END;

FUNCTION btGTCase;
        BEGIN
                btGTCase := tCaseLabel;
        END;

PROCEDURE btSValErr;
        BEGIN
                valErr := newValErr;
        END;

FUNCTION btGValErr;
        BEGIN
                btGValErr := valErr;
        END;

PROCEDURE btSExcErr;
        BEGIN
                excErr := newExcErr;
        END;

FUNCTION btGExcErr;
        BEGIN
                btGExcErr := excErr;
        END;

PROCEDURE btSCheckExc;
        BEGIN
                IF NOT (foundExc(expExc)) THEN
                        prtExc(expExc);
        END;

FUNCTION btGCheckExc;
        BEGIN
                btGCheckExc := foundExc(expExc);
        END;

PROCEDURE btSCheckInt;
        BEGIN
                IF NOT (foundExc(expExc)) THEN
                        prtExc(expExc)
                ELSE IF (NOT (expVal = actVal)) THEN
                        prtInt(expVal,actVal)
```

```
        END;

    PROCEDURE btSCheckBool;
        BEGIN
            IF NOT (foundExc(expExc)) THEN
                prtExc(expExc)
            ELSE IF (NOT (expVal = actVal)) THEN
                prtBool(expVal,actVal)
        END;

    PROCEDURE btSCheckStr;
        BEGIN
            IF NOT (foundExc(expExc)) THEN
                prtExc(expExc)
            ELSE IF (NOT (expVal = actVal)) THEN
                prtStr(expVal,actVal)
        END;

    PROCEDURE btGSum;
        VAR
            correct: integer;
        BEGIN
            correct := tCase - excErr - valErr;
            writeln('Statistics:');
            writeln('   Number of tests: ',tCase);
            writeln('    Number correct: ',correct);
            writeln('    Percentage correct: ',(100.0 * correct /
tCase): 7: 2);
            writeln('   Number of exception errors: ',excErr);
            writeln('   Number of value errors: ',valErr);
        END;

    PROCEDURE btGPrtExc;
        BEGIN
            prtExc(expExc);
        END;

    PROCEDURE btGPrtInt;
        BEGIN
            prtInt(expVal,actVal);
        END;

    PROCEDURE btGPrtBool;
        BEGIN
            prtBool(expVal,actVal);
        END;

    PROCEDURE btGPrtStr;
        BEGIN
            prtStr(expVal,actVal);
        END;

END.
```

BExc.intspec

Batch Exception (BExc) Interface Specification Semantics

STATE VARIABLES

```
tbl: sequence[1..syMaxExc] of boolean
numExc:int
```

SET CALL EFFECTS

```
sInit(num):
        numExc = num

sClear:
        (forall  i)(
                if i in [1,numExc] then
                        tbl[i] = false
        )

sExc(exc):
        tbl[exc] = true
```

GET CALL RETURN VALUES

gCount = number of tbl entries set to true

gExc(exc) = tbl[exc]

gLegExc(exc) iff exc in [1, numExc]

gNumExc = numExc

EXCEPTIONS

```
sInit(num):
        (maxExc, num > syMaxExc)

sExc(exc)
        (notLegExc, not gLegExc(exc))

gExc(exc)
        (notLegExc, not gLegExc(exc))
```

BExc.p

```
UNIT BExc;

        INTERFACE
```

```
        PROCEDURE beSInit(num: integer);
        PROCEDURE beSClear;
        PROCEDURE beSExc(exc: integer);
        FUNCTION beGExc(exc: integer): boolean;
        FUNCTION beGCount: integer;
        FUNCTION beGLegExc(exc: integer): boolean;
        FUNCTION beGNumExc: integer;

IMPLEMENTATION

        CONST
                MaxExc = 10;

        TYPE
                actExcType = ARRAY [1..MaxExc] OF boolean;

        VAR
                actExc: actExcType;
                numExc: integer;

        PROCEDURE beSInit;
                VAR
                        i: integer;
                BEGIN
                        IF (num IN [1..MaxExc]) THEN
                                BEGIN
                                        numExc := num;
                                        FOR i := 1 TO numExc DO
                                                actExc[i] := false;
                                END
                        ELSE
                                writeln('maxExc');
                END;

        PROCEDURE beSClear;
                VAR
                        i: integer;
                BEGIN
                        FOR i := 1 TO numExc DO
                                actExc[i] := false;
                END;

        PROCEDURE beSExc;
                BEGIN
                        IF (exc IN [1..numExc]) THEN
                                actExc[exc] := true
                        ELSE
                                writeln('notLegExc');
                END;

        FUNCTION beGExc;
                BEGIN
                        IF (exc IN [1..numExc]) THEN
                                beGExc := actExc[exc]
```

```
                ELSE
                        writeln('notLegExc');
        END;

FUNCTION beGCount;
        VAR
                i,cnt:  integer;
        BEGIN
                cnt := 0;
                FOR i := 1 TO numExc DO
                        IF  (actExc[i])  THEN
                                cnt := cnt + 1;
                beGCount := cnt;
        END;

FUNCTION beGLegExc;
        BEGIN
                beGLegExc := exc IN [1..numExc]
        END;

FUNCTION beGNumExc;
        BEGIN
                beGNumExc := numExc
        END;
END.
```

Appendix E

Module Testing for BB:Symtbl

symtbl.intspec

Symbol Table Interface Specification

Note: sInit must be called before any other call.

STATE VARIABLES

```
tbl: set of tuple of (
        sym: string;
        id: int;
)
```

SET CALL EFFECTS

```
sInit():
        tbl = {}

sAddsym(sym):
        add (sym,id) to tbl where
                id is in [1,MAXSYMS] and
                id is not the identifier of any other symbol in tbl

sDel(id):
        delete the element of s with identifier id
```

GET CALL RETURN VALUES

```
gCnt() = siz(tbl)

gLegid(id) iff id is an identifier in in tbl

gLegsym(sym) iff sym is a symbol in tbl

gSym(id) = the symbol in tbl with identifier id

gId(sym) = the identifier in tbl with symbol sym
```

EXCEPTIONS

```
sAddsym(sym):
        (maxlen, length(sym) > ST_MAXSYMLEN)
        (Legid, gLegid(sym))
        (Tblfull, gCnt = ST_MAXSYMS)
```

```
sDel(id)
        (Notlegid, not gLegid(id))

gId(sym):
        (Notlegsym, not gLegsym(sym))

gSym(id):
        (Notlegid, not gLegid(id))
```

symtbl.p

```
UNIT symTbl;

        INTERFACE

                USES sysDef,symtblExc,objIntf;

                CONST
                        stMaxSyms = syMaxNams;
                        stMaxSymLen = syMaxNamLen;

                TYPE
                        stTTbl = ARRAY [1..stMaxSyms] OF
                                                RECORD
                                                        sym: string[stMaxSymLen];
                                                        inUse: boolean;
                                                END;

                        TSymtbl = OBJECT
                                                fTbl: stTTbl;
                                                fTblCnt: integer;
                                                PROCEDURE sInit;
                                                FUNCTION gCnt: integer;
                                                PROCEDURE sAddsym(sym: string);
                                                PROCEDURE sDel(id: integer);
                                                FUNCTION gLegId(id: integer): boolean;
                                                FUNCTION gLegSym(sym: string): boolean;
                                                FUNCTION gId(sym: string): integer;
                                                FUNCTION gSym(id: integer): string;
                                        END;

        IMPLEMENTATION

                {$I symtblImpl.p }

END.
```

symtbl.tplan

Symtbl Test Plan

Assumptions

MAXSYMS >= 6
id allocation policy
when a symbol is added, it is given the smallest id available

Test Case Selection Strategy

Critical Module State Values
number of symbols in table: 0,MAXSYMS/2,MAXSYMS
symbol length: short, MAXSYMLEN

Exceptional Behaviour
let N = the number of symbols currently in the table
let BADIDS = {-10**6,-1,0,N,10**6}

for each critical module state
add overlength symbols, boundary and extreme
if the table is full
add a symbol not in the table
request symbols for, and delete, BADIDS
request ids for symbols not in table
add every symbol in the table

Normal Behaviour
ensure that the empty string is not in the empty table
add the empty string to an empty table
check that a very long symbol is illegal
for each critical module state
check table length
for each i in [1,MAXSYMS]
if i in [1,tSiz]
check tSym(i) legal, with correct id
check i legal, with correct symbol
check sDel deletes i and tSym(i)
check that gCnt is tSiz-(i+1)
else
check tSym(i) and i illegal

Implementation Strategy

Files:
symtblBTest.p is a hand coded driver using BTest and BExc
symtblExc is re-defined and implemented in symtblBExc.p
no stubs
Pascal functions to support iterating over the critical module states, viewed as a sequence:
procedure tInit: set position to before first state
function tNext: load next state into symtbl and return true
function tEnd: true iff position is after last state
function tSiz: number of symbols in current state
function tSym(i): symbol with id i in current state
function tMksym(i,len): string consisting of i converted to ASCII, padded right with *'s to length len

Test File Skeletons:
Unit symtblBExc

```
        uses BExc
        interface
            {same as for symtblExc}
            function gName (exc:int) : string;
            const
                {one per exception assigned a consecutive integer
                    beginning at 1}
                numExc = {number of exceptions}
        implementation
            {for each exc procedure in xExc.p implement again here}
            procedure y;
                sExc( {constant corresponding to y} )
            function gName (exc:int) : string;
end.

program symtblBTest
    uses sysDef, BTest, BExc, symtblBExc, symtbl;

{symtbl dependent global definitions}

begin
    {initialise all units}
    {test cases of form}
        sTCase(tCaseLabel)

        {trace}

        sCheckInt(expExc, expVal, actVal)
    {end test case}

    {looping mechanism can be placed here too}

end.
```

symtblBTest.p

```
PROGRAM symtblBTest;

USES objIntf,PackIntf,sysDef,symTbl,BTest, {$U symtblBExc.p} symTblExc;

CONST
    tMkSymMax = 255;
    tFillChar = '*';
    tTblSiz = 5;

TYPE
    tTblRecType = RECORD
                    siz: integer;
                    symlen: integer;
                END;
    tTblType = ARRAY [1..tTblSiz] OF tTblRecType;

VAR
    s: TSymtbl;
```

```
        sInt: integer;
        sBool: boolean;
        sStr: string;
        i,tCur: integer;
        tTbl: tTblType;

PROCEDURE sInit;
        BEGIN
                tTbl[1].siz := 0;
                tTbl[1].symlen := 0;

                tTbl[2].siz := stMaxSyms DIV 2;
                tTbl[2].symlen := 0;

                tTbl[3].siz := stMaxSyms DIV 2;
                tTbl[3].symlen := stMaxSymLen;

                tTbl[4].siz := stMaxSyms;
                tTbl[4].symlen := 0;

                tTbl[5].siz := stMaxSyms;
                tTbl[5].symlen := stMaxSymLen;

                new(s);
        END;

FUNCTION tMkSym(i,len: integer): string;
        VAR
                buf: str255;
                j: integer;
        BEGIN
                NumToString(i,buf);
                WHILE (length(buf) < len) DO
                        BEGIN
                                insert(tFillChar,buf,(length(buf) + 1));
                        END;
                tMkSym := buf;
        END;

PROCEDURE tInit;
        BEGIN
                tCur := 0;
        END;

PROCEDURE tNext;
        VAR
                i: integer;
        BEGIN
                tCur := tCur + 1;
                IF (tCur >= 1) AND (tCur <= tTblSiz) THEN
                        BEGIN
                                s.sInit;
                                FOR i := 1 TO tTbl[tCur].siz DO
                                        s.sAddsym(tMksym(i,tTbl[tCur].symlen));
```

```
                END;
    END;

FUNCTION tEnd: boolean;
    BEGIN
        tEnd := tCur > tTblSiz;
    END;

FUNCTION tSiz: integer;
    BEGIN
        tSiz := tTbl[tCur].siz;
    END;

FUNCTION tSym(i: integer): string;
    BEGIN
        tSym := tMksym(i,tTbl[tCur].symlen);
    END;

BEGIN

    sInit;
    btSInit;

    {****exceptions****}
    tInit;
    tNext;

    WHILE (NOT tEnd) DO
        BEGIN

        btSTCase('**** Add over length symbols, #1 ');
        s.sAddsym(tMkSym(0,stMaxSymLen + 1));
        btSCheckExc(maxLen);

        btSTCase('**** Add over length symbols, #2 ');
        s.sAddsym(tMkSym(0,tMkSymMax));
        btSCheckExc(maxLen);

        IF (tSiz = stMaxSyms) THEN
            BEGIN
            btStCase('**** Add symbol to full table ');
            s.sAddsym('x');
            btSCheckExc(tblFull);
            END;

        btSTCase('**** Delete ids not in the table, #1 ');
        s.sDel( - 1);
        btSCheckExc(notLegId);

        btSTCase('**** Delete ids not in the table, #2 ');
        s.sDel(tSiz + 1);
        btSCheckExc(notLegId);

        btSTCase('**** Request symbols for ids not in the table, #1 ');
```

```
        sStr := s.gSym( - 1);
        btSCheckExc(notLegId);

        btSTCase('**** Request symbols for ids not in the table, #2 ');
        sStr := s.gSym(tSiz + 1);
        btSCheckExc(notLegId);

        btSTCase('**** Request ids for symbols not in the table, #1 ');
        sInt := s.gId(tMkSym(tSiz + 1,0));
        btSCheckExc(notLegSym);

        btSTCase('**** Request ids for symbols not in the table, #2 ');
        sInt := s.gId('');
        btSCheckExc(notLegSym);

        FOR i := 1 TO tSiz DO
                BEGIN
                btSTCase('**** Add every existing symbol ');
                s.sAddsym(tSym(i));
                btSCheckExc(legSym);
                END;

        IF (tSiz > 0) THEN
                BEGIN
                btSTCase('**** Delete a symbol; ');
                s.sDel(1);
                btSCheckExc(btNoExc);

                btSTCase('**** Then request deleted symbol');
                sInt := s.gId(tSym(1));
                btSCheckExc(notLegSym);

                btSTCase('**** Then request deleted id');
                sStr := s.gSym(1);
                btSCheckExc(notLegId);
                END;

        tNext;

        END;

{*****normal   case*****}

btSTCase('**** Check for empty string in empty table ');
s.sInit;
btSCheckBool(btNoExc,false,s.gLegSym(''));

btSTCase('**** Check that empty string can be added, part A ');
s.sInit;
s.sAddsym('');
btSCheckBool(btNoExc,true,s.gLegSym(''));

btSTCase('**** Check that empty string can be added, part B ');
btSCheckBool(btNoExc,true,s.gLegId(1));
```

```
tInit;
tNext;

WHILE (NOT tEnd) DO
        BEGIN
        btSTCase('*** Check table length ');
        btSCheckInt(btNoExc,tSiz,s.gCnt);

        btSTCase('** Check that a very long symbol is illegal ');
        btSCheckBool(btNoExc, false, s.gLegSym( tMkSym(0, tMkSymMax)));

        FOR i := 1 TO stMaxSyms DO
                BEGIN

                IF (i <= tSiz) THEN
                        BEGIN

                        btSTCase( '*** Check that t_sym(i) is legal');
                        btSCheckBool(m btNoExc, true, s.gLegSym(tSym(i)));

                        btSTCase( '*** Check that t_sym(i) has correct id');
                        btSCheckInt(btNoExc, i, s.gId(tSym(i)));

                        btSTCase('*** Check that i is legal' );
                        btSCheckBool(btNoExc, true, s.gLegId(i));

                        btSTCase('*** Check that i has correct symbol');
                        btSCheckStr(btNoExc, tSym(i), s.gSym(i));

                        btSTCase('*** Check that s_del deletes i');
                        s.sDel(i);
                        btSCheckBool(btNoExc, false, s.gLegSym(tSym(i)));

                        btSTCase('*** Check that sDel deletes symbol');
                        btSCheckBool(btNoExc, false, s.gLegId(i));

                        btSTCase('*** Check table length ');
                        btSCheckInt(btNoExc,tSiz - i, s.gCnt);

                        END
                ELSE
                        BEGIN

                        btSTCase('*** Check that tSym(i)is not legal' );
                        btSCheckBool(btNoExc, false, s.gLegSym(tSym(i)));

                        btSTCase('*** Check that i is not legal');
                        btSCheckBool(btNoExc,false, s.gLegId(i));

                END

                END;
```

```
            tNext;

            END;

        btGSum;
END.
```

symtblBExc.p

```
UNIT symtblExc;

    INTERFACE

        USES BExc;

        CONST
            legSym = 1;
            notLegId = 2;
            maxLen = 3;
            notLegSym = 4;
            tblFull = 5;
            numExc = 5;

        FUNCTION stGName(exc: integer): string;
        PROCEDURE stLegSym;
        PROCEDURE stNotLegId;
        PROCEDURE stMaxLen;
        PROCEDURE stNotLegSym;
        PROCEDURE stTblFull;

    IMPLEMENTATION

        FUNCTION stGName;
            BEGIN
                IF (exc = legSym) THEN
                    stGName := 'legsym';
                IF (exc = maxLen) THEN
                    stGName := 'maxlen';
                IF (exc = notLegId) THEN
                    stGName := 'notlegid';
                IF (exc = notLegSym) THEN
                    stGName := 'notlegsym';
                IF (exc = tblFull) THEN
                    stGName := 'tblfull';
            END;

        PROCEDURE stLegSym;
            BEGIN
                beSExc(legSym);
            END;

        PROCEDURE stNotLegId;
            BEGIN
```

```
                beSExc(notLegId);
        END;

    PROCEDURE stMaxLen;
        BEGIN
                beSExc(maxLen);
        END;

    PROCEDURE stNotLegSym;
        BEGIN
                beSExc(notLegSym);
        END;

    PROCEDURE stTblFull;
        BEGIN
                beSExc(tblFull);
        END;

END.
```

Appendix F

Module Testing for CaseTool:Space

Space.intspec

Space Interface Specification Semantics

STATE VARIABLES

fTable : sequence [1..kMaxSectorID] of handle
fNextFreeZone : longint

SET CALL EFFECTS

Occupy(zone,src,size):
convert zone to pointer and
copy next size bytes from src to pointer

CloneOccupant(zone,dest,size):
convert zone to pointer and
copy next size bytes from pointer to dest

IRelativeSpace:
fTable = {} and
fNextFreeZone = kMaxSectorSize + 1

Free:
release all memory associated with TReleativeSpace object
release self

NewZone(size,addr):
if (fNextFreeZone + size) > kMaxSectorSize then
fTable[length(fTable)+1] = GetFreeHandle
and
fNextFreeZone = (fNextFreeZone + size) rounded up to the next nearest even number

GET CALL RETURNS

CurrentAddress(zone) = zone converted to pointer

NewZone(size,addr) = x iff
x[1] = length(fTable) and
x[2] = fNextFreeZone and
addr = fTable[x[1]].theSector^ + x[2]

CONSTANTS

```
        NullSector = 0
        kMaxSectorId = 1024
        NoZone = 0
        kSectorSizeinK = 8
        kMaxSectorSize = 1024 * kSectorSizeinK

TYPES

        ZoneID = sequence [1..2] OF INTEGER;

EXCEPTIONS

        NewZone:
                (FailNil, NewHandle(kMaxSectorSize) = NIL)
```

Space.p

```
UNIT Space; { Interface Syntax Specifications }

        INTERFACE

                USES ObjIntf,Memory,Types,PasLibIntf,Files,SpaceExc;

                CONST
                        { exported }
                        NullSector = 0;
                        kMaxSectorID = 1024;
                        NoZone = 0;
                        kSectorSizeinK = 1 {originally 8 - changed for machine
limitations};
                        kMaxSectorSize = kSectorSizeInK * 256;

                TYPE

                        { local }
                        FTableSize = 1..kMaxSectorID;
                        ZoneID = PACKED ARRAY [1..2] OF INTEGER;

                        SectorInfo = RECORD
                                                theSector: HANDLE;
                                        END;

                        { exported }
                        TRelativeSpace = OBJECT (TObject)

                                 { local }
                                fTable: ARRAY [FTableSize] OF SectorInfo;
                                fNextFreeSector: integer;
                                fActiveSector: integer;
                                fNextFreeZone: LONGINT;

                                 { exported }
                                PROCEDURE IRelativeSpace(theBlockSize: INTEGER);
```

```
                    FUNCTION NewZone(size: INTEGER;
                            VAR addr: UNIV Ptr): ZoneID;
                    PROCEDURE Occupy(zone: ZoneID;
                            occupant: UNIV Ptr;
                            size: INTEGER);
                    PROCEDURE CloneOccupant(src: ZoneID;
                            dest: UNIV Ptr;
                    size: INTEGER);
                    PROCEDURE CurrentAddress(x: ZoneID;
                            VAR addr: UNIV PTR);
                    PROCEDURE Free; OVERRIDE;

                     { local }
                    FUNCTION NewSector(size: INTEGER): handle;
                    FUNCTION GetFreeHandle: Handle;

            END;

    IMPLEMENTATION

        {$I SpaceImpl.p}

END.
```

Space.tplan

```
                        Space Test Plan

Test Case Selection Strategy

    Critical Module State Values
        number of entries in table: 0,kMaxSectorID/2,kMaxSectorID-1
        zone sizes: 0,1,kMaxSectorSize/2,kMaxSectorSize

    Exceptional Behaviour
        for each s in zone size
            for each critical number entires in the table
                check FailNil

    Normal Behaviour
        check correct initialization
        for each s in zone size
            for each critical number of entries in the table
                check NewZone, z, of size, s, beginning at address, a
                check Occupy of z from ptr of str1, results of tMkStr(i,s)
                check CloneOccupant of z to ptr of str2
                check str1 = str2
                compact heap
                check contents of z
                check correct disposal of memory

Implementation Strategy
```

Files:

SpaceBTest.p is a hand coded driver using BTes and BExc based on symtblBTest example
SpaceBExc implements FailNil based on symtblBExc example
no stubs

Pascal functions to support iterating over the critical table and zone values, viewed as a sequence:

procedure tInitTable: set position to before the first state
procedure tInitZone: set position to before the first zone
function tNextTable: load next table state size and return true
function tNextZone: set next zone size and return true
function tEndTable: iff position is after last state
function tEndZone: iff position is after last zone

Pascal functions to set machine state:

procedure sHeapFull: allocate all available space on application heap
proceudre sHeapEmpty: dispose of all space allocated by sHeapFull

Pascal support for additional test type:

```
type    tStrType = packed array [ 1 .. kMaxSectorSize ] of char;
                tStrPtr = ^tStrType;
procedure tSCheckStrPtr(expExc,expVal,ActVal,len);
procedure tGPtrStrPtr(expVal,actVal,len);
```

Pascal functions to support creating variables of additional test type:

function tMkStr(i,len): returns tStrType consisting of i's for length len

SpaceBTest.p

```
PROGRAM SpaceBTest;

    USES objIntf,PackIntf,Memory,SpaceMgr,BTest, {$U SpaceMgrBExc.p}
SpaceMgrExc;

    CONST
        tMkSymMax = 255;
        tFillChar = '*';
        tSizTable = 3;
        tSizZone = 4;

    TYPE
        tTableType = ARRAY [1..tSizTable] OF integer;
        tZoneType = ARRAY [1..tSizZone] OF integer;
        tStrType = PACKED ARRAY [1..kMaxSectorSize] OF char;
        tStrPtr = ^tStrType;

    VAR
        s: TRelativeSpace;
        sInt,Zsize: integer;
        sBool: boolean;
        sStr: string;
        z: ZoneID;
        a,sStrPtr1,sStrPtr2: tStrPtr;

        i,tCurTable,tCurZone: integer;
        tTable: tTableType;
```

```
        tZone: tZoneType;

PROCEDURE tGPrtStrPtr(expVal,actVal: tStrPtr; len: integer);
        VAR
                i: integer;
        BEGIN
                btSValErr(btGValErr + 1);
                writeln('***** Value error, ',btGTCase,'.');
                write(' Expected value:');
                IF (expVal = NIL) OR (len = 0) THEN
                        write('NIL')
                ELSE
                        FOR i := 1 TO len DO
                                write(expVal^[i]);
                write('  Actual value:');
                IF (actVal = NIL) OR (len = 0) THEN
                        write('NIL')
                ELSE
                        FOR i := 1 TO len DO
                                write(actVal^[i]);
                writeln;
        END;

PROCEDURE tSCheckStrPtr(expExc: integer; expVal,actVal: tStrPtr;
                                        len: integer);
        VAR
                i: integer;
                equal: boolean;
        BEGIN
                IF NOT (btGCheckExc(expExc)) THEN
                        btGPrtExc(expExc)
                ELSE IF (NOT (expVal = actVal)) THEN
                        BEGIN
                                i := 1;
                                equal := true;
                                WHILE (i < len) AND equal DO
                                        BEGIN
                                                equal := expVal^[i] = actVal^[i];
                                                i := i + 1;
                                        END;
                                IF NOT (equal) THEN
                                        tGPrtStrPtr(expVal,actVal,len);
                        END;
        END;

PROCEDURE sHeapFull;
{ allocate all available space on the application heap }
        BEGIN
                {KDF code this }
        END;

PROCEDURE sHeapEmpty;
{ dispose of space allocated by sHeapFull }
        BEGIN
```

```
            {KDF code this }
        END;

PROCEDURE sInit;
        BEGIN
            tTable[1] := 0;
            tTable[2] := kMaxSectorID DIV 2;
            tTable[3] := kMaxSectorID - 1;

            tZone[1] := 0;
            tZone[2] := 1;
            tZone[3] := kMaxSectorSize div 2;
            tZone[4] := kMaxSectorSize;
        END;

FUNCTION tMkStr(i: char; len: integer): tStrPtr;
        VAR
            j: integer;
            temp: tStrPtr;
        BEGIN
            new(temp);
            FOR j := 1 TO len DO
                temp^[j] := i;
            tMkStr := temp;
        END;

PROCEDURE tInitTable;
        BEGIN
            tCurTable := 0;
        END;

PROCEDURE tInitZone;
        BEGIN
            tCurZone := 0;
        END;

PROCEDURE tNextTable;
        VAR
            i: integer;
            z: ZoneID;
            a: tStrPtr;
        BEGIN
            tCurTable := tCurTable + 1;
            IF (tCurTable >= 1) AND (tCurTable <= tSizTable) THEN
                BEGIN
                    new(s);
                    s.IRelativeSpace(0);
                    FOR i := 1 TO tTable[tCurTable] DO
                        z := s.NewZone(tZone[tCurZone],a);
                END;
        END;

PROCEDURE tNextZone;
        VAR
```

```
                i: integer;
        BEGIN
                tCurZone := tCurZone + 1;
        END;

FUNCTION tEndTable: boolean;
        BEGIN
                tEndTable := tCurTable > tSizTable;
        END;

FUNCTION tEndZone: boolean;
        BEGIN
                tEndZone := tCurZone > tSizZone;
        END;

BEGIN

        sInit;
        btSInit;

        {*****normal   case*****}
        tInitZone;
        tNextZone;
        WHILE (NOT tEndZone) DO
                BEGIN

                Zsize := tZone[tCurZone];
                tInitTable;
                tNextTable;

                WHILE (NOT tEndTable) DO
                        BEGIN

                        btSTCase('*** Check NewZone');
                        z := s.NewZone(Zsize,a);
                        s.CurrentAddress(z,sStrPtr1);
                        tSCheckStrPtr(btNoExc,a,sStrPtr1,Zsize);

                        btSTCase('*** Check Occupy of z from ptr of str1');
                        sStrPtr1  :=  tMkStr('*',Zsize);
                        s.Occupy(z,sStrPtr1,Zsize);
                        s.CurrentAddress(z,a);
                        tSCheckStrPtr(btNoExc,a,sStrPtr1,Zsize);

                        btSTCase('*** Check CloneOccupy of z from ptr of str');
                        new(sStrPtr2);
                        s.CloneOccupant(z,sStrPtr2,Zsize);
                        s.CurrentAddress(z,a);
                        tSCheckStrPtr(btNoExc,a,sStrPtr2,Zsize);

                        btSTCase('*** Check CloneOccupy of z from ptr of str');
                        tSCheckStrPtr(btNoExc,sStrPtr1,sStrPtr2,Zsize);

                        btSTCase('*** Check Free');
```

```
                sInt := FreeMem + sizeof(s);
                s.Free;
                btSCheckBool(btNoExc,(sInt <= FreeMem),true);

                tNextTable;
                END;

        tNextZone;
        END;

    btGSum;
END.
```

SpaceBExc.p

```
UNIT SpaceExc;

    INTERFACE

        USES BExc;

        CONST
            smFailNil = 1;
            numExc = 1;

        PROCEDURE FailNil;
        FUNCTION gName(exc: integer): string;

    IMPLEMENTATION

        PROCEDURE FailNil;
            BEGIN
                beSExc(smFailNil); ;
            END;

        FUNCTION gName;
            BEGIN
                IF (exc = smFailNil) THEN
                    gName := 'FailNil';
            END;

END.
```

VITA

Surname:
Franklin

Given Names:
Katherine Diane

Place of Birth:
North York, Ontario

Date of Birth:
Dec 31, 1965.

Education Institutions Attended:

University of Victoria	1988 to 1990
Queen's University	1984 to 1988
Marianopolis College	1982 to 1984

Degrees Awarded:

B.Sc. (Honours)	Queen's University	1988
B.A.	Queen's University	1986
D.E.C.	Marianopolis College	1984

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying of publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

A Case Study in Software Testing

Author:

[redacted]

KATHERINE DIANE FRANKLIN

June 29, 1990

Date