A SYSTEMATIC APPROACH TO INSTALLING A COMPUTER PRODUCT

By

SUE-FONG CHRISTINE SHEN

Bachelor of Science in Agriculture
National Taiwan University
Taiwan, R.O.C.
1974

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1980

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fullfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1984

Thesis
1984
S546s
cop. 2

A SYSTEMATIC APPROACH TO INSTALLING A COMPUTER PRODUCT

Thesis Approved:

Sharilyn A. Thoreson
Thesis Adviser

Michael J. Folk

J P Chandler

Norman N. Durham
Dean of Graduate College

1184247

PREFACE

This study is cocerned with the approach to installing a computer product. In separate chapters, it describes procedures to install a hardware device and a software package. An extensive discussion about techniques for testing and debugging, which are two important proceudres in a software package installion process, is also included.

The author wishes to express her sincere appreciation to her major adviser, Dr. Sharilyn A. Thoreson, for her invaluable guidance, assistance, and understanding throughout this study. Appreciation is also expressed to the other committee members, Dr. Michael J. Folk and Dr. John P. Chandler, for their advice since the proposing stage of this study.

Finally, a note of thanks is given to my husband, Zie-Chiang, who applied the admission for me from the beginning and kept "pushing" and "encouraging" me throughout my graduate studies at the Computing and Information Sciences Department of Oklahoma State University.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Since the world's first all-electronic computer went into operation over a quarter of a century ago, computers have had a more and more profound effect on our lives. Many people who have never used computers try to learn and use these products of modern technology, and more of those who have used computers try to get greater benefits from their computer systems. New computer products appear on the market every day and for those who like to get advantages from these products, installing them becomes an inevitable task.

Statement of the Problem

It is not uncommon that a computer product does not function as it is expected to after first being installed. Most of the time, when the product is found malfunctioning lots of efforts already been invested. Furthermore, detecting the causes of problems and correcting these problems consume even more effort.

Motivation and Purpose

For a successful and efficient installation, it is necessary for one to pay full attention to every step of the installation procedure. The problems encountered and the accumulated experiences of searching for solutions during the installation of a plotter and a software package in the Computing and Information Science Department of Oklahoma State University in the summer of 1983 have given me the motivation to write this thesis. The primary goal of this thesis is to explore a systematic approach to installing a computer product. I hope this thesis can give those who encounter similar problems a little help.

Definitions of Terms

A computer system consists of a collection of component elements, and every element performs its specific function. The system must be able to sense information from and provide information to its environment, that is, it must have inputs and outputs (3). The component elements which have independent functions, and without which the system can still execute some other functions, are called computer products.

The installation of a computer product is defined as the integration of the product into the computer system. Computer products can be various devices, such as computer peripherals; they also can be software packages.

Organization of Study

Chapter I introduces the problems, the motivation and purpose of the study, and some definitions of related terms.

Chapter II presents the procedures needed to install the hardware of a product. It also discusses interfacing - an important factor of installing the hardware. The final step of installing, which is the installation test, is described in three progressive sequences, namely static, dynamic standalone, and dynamic integrated testing.

Chapter III describes the portability of the software products, which contains the descriptions about the relationship among product designer, installer and the installation, and the techniques for enhancing the portability. This chapter also presents general procedures for installing a software product.

Chapter IV mainly talks about the software installation's final steps - testing, debugging and documentation. The description and comparison of testing and debugging techniques are presented. It also mentions the criterion of sound documentation for the software product, and what the installer should do about the documentation.

Chapter V is the summary and conclusion of the computer product installation.

CHAPTER II

HARDWARE INSTALLATION

A computer product can be many things. It may be pure hardware or has little interaction with software, for instance, a modem, a CRT display device, a line printer, etc., or it may be pure software, such as a software package. In this chapter, the installation of the products which have little interaction with software is presented.

The installation of products, especially those new and attractive products appearing on the market everyday, may or may not be in the initial system design plan. It sometimes didn't come to the designer's knowledge when the system was designed. Therefore, it is possible that a desired product and the computer mainframe are from different manufacturers, which may make the installation more demanding.

Installation Plan

The installation plan should be started as soon as the decision is made to purchase a product. The purchaser's specified requirements should be met. To attain this goal, it is necessary to have a study about the best-fit products. When the product is ordered and in transit, a suitable environment for it should have been prepared. A computer product needs a cool, clean and safe environment to avoid damage (11).

Installation Procedures

<u>First Check-out</u>

The individual parts of the product are supposed to have been inspected before the unit was shipped to the buyer, but it never be too careful to inspect the device and accessories for any physical damage sustained in transit. One also needs to make sure all the items that should accompany the device are present.

<u>A Consideration of Hardware Interface</u>

An interface is a system which connects the computer and its peripherals. Selection of appropriate interface hardware has a direct impact on ease of system integration and efficiency of communication between the computer and the outside world. Ideally, all peripherals and computers should conform to some standard that specifies all the characteristics of their connections, making all such devices plug-to-plug compatible. Unfortunately, no such standard exists at the present time. In actuality, there are four areas of compatibility that must be satisfied in order to successfully interface a device and a computer. They are:

1. Mechanical Compatibility

Mechanical compatibility implies that male connector and female connector on the computer and peripheral device respectively be of the same number of pins and fit to each other.

2. Electrical Compatibility

Data is passed between devices over the data lines using two voltage levels to represent the two possible states (1 or 2) of a binary digit or bit. Electrical compatibility means that the voltage levels must be compatible for the two devices.

The line voltage selected for the device should have been identified on the device panel. It can be changed, but usually by qualified service personnel only.

3. Data Compatibility

Once an interface has made the computer and its peripheral device mechanically and electrically compatible, they are capable of exchanging messages in the form of electrical signals, but in order to understand and execute these messages, certain conventions must be followed regarding the formatting of the data to be exchanged. For internal communication, devices may use any data format, but each usually will input and output data in one of two standard representations, namely EBCDIC or ASCII. It is not the author's interest to discuss the difference between these two data representations, but using the same representation is a must to make two devices data compatible.

4. Communication Compatibility

Data transmission speeds of computers and their peripheral devices are often different. The peripherals are usually slower in their ability to execute instructions than computers are in their ability to generate them. It is necessary to provide some means to ensure the transmission and receipt of data between devices without loss of any data.

Baud rate is an important factor to consider in establishing communication compatibility between two devices. The baud rate is approximately equal to the number of bits transmitted per second. By setting the baud rates of the computer and its peripherals equal, the data transmission between the interfacing devices won't be lost.

It is necessary to refer to the computer's manual to determine its baud rate and set the peripheral's baud rate accordingly. For example, the baud rate selection switch of HP7470A plotter in Oklahoma State University is set at 4800 to synchronize with the Perkin-Elmer mini-computer.

So far, all these check-outs discussed are static check-outs.

Dynamic Standalone Check-out

After the principle aspects of the computer product's hardware being taken care of, the user can start operating the device. The following action, of course, is to plug the power cable, then turn on the power.

At this point, refer to the manual to check if the specified lights, cursor, ...etc. are functioning as expected. Sometimes some basic defects can be revealed and be replaced, avoiding unnecessary testing later on.

If everything goes fine so far, keep carrying on the procedures stated on the device manual. Usually the manual will give straight-forward instructions to operate, for instance, a plotter. The manual will indicate how to use pens correctly, load paper, etc.

Before the device integrates with other devices, it is wise to have a confidence test about the device. The confidence test provides the user a visual indication that the device is operating properly. This confidence test can always be repeated whenever the hardware defect of the device is suspected.

For example, when the HP7470A plotter is first installed in OSU, this self-test checks the mechanical and electronic functions. It does so by selecting alternative pens, moving between scaling points 1 and 2, drawing an asterisk with the second pen, and moving specified spaces along the X-axis. The second pen is then stored and the pen holder returns to the first pen, indicating the confidence test is completed.

Dynamic Integrated Check-out

After the device is done with the dynamic standalone check-outs without any problem, it should then be integrated with the other devices, for instance, the mainframe, to ensure that its dynamic operation is correct when it is driven by the rest of the system. This systematic progression from static check-out to standalone dynamic check-out to integrated dynamic check-out provides an efficient way to detect, isolate, and correct errors in the hardware.

It is obvious that identifying and correcting problems is easier when it is in static check-out stage, than it is in dynamic check-out stage; it is cost-effective that problems being taken care of as early as possible.

CHAPTER III

SOFTWARE INSTALLATION

Overview

Previously mentioned check-out procedures are enough to complete the installation for computer products which consist of pure hardware, that is, products that have little or no interaction with the software. Disc drives, line printers, card readers, graphic displays, etc., which are I/O devices interact less with the software and often contain a large amount of pure hardware belong to this group, but many of the computer products are pure software and are called software products, or software packages. In this thesis, the terms software product and software package will be used interchangeably.

Software products capture what practitioners of a field need, in a form that requires far less knowledge of computers, and of the field itself, than that required to program a problem directly. Obviously, such facility is why they have become so popular in recent years.

Before an end user is able to use a software product, he/she encounters a problem in installing it into the computer system. An installation guide which describes installation procedures step by step always accompanies the product when the product is delivered to a user; this makes the installation an easier task. As mentioned in the last chapter, new computers and computer products appear on the market everyday; therefore the product designer can not possibly have the designs updated to fit a list of target computers into which the product may be installed. This would give the future installer difficulty in installing the product. It also may decrease the product's popularity.

Portability of the Software Products

The pace of change in computer hardware technology is such that computing machinery becomes obsolete long before the programs which execute on that machinery (19). Besides, huge sums of money are spent on moving programs from machine to machine. Therefore, it is very important that programs be written in such a manner that they may be implemented under more than one computer/operating system configuration.

In Poole and Waite's paper (16), portability is defined as a measure of the ease with which a program can be transferred from one environment to another; they consider if the effort required to move the program is much less than that required to implement it initially, and the effort is small in an absolute sense, then that program is highly portable.

Although it is desirable to have programs written in such a way that they do not depend on the underlying computer hardware/operating systems, in practice, complete independence is impossible. Portability's definition, as we may have noticed, does not exclude rewriting small parts of programs.

Relationship Among Product Designer, Installer and the Installation

The product designer's responsibility is to develop the portable software, document it, provide support material, and transmit the results to the installer. Theoretically the designer has the responsibility for providing the installer with all the information about the portable software that is needed to carry out the installation. In actuality, the designers of typical software products are experts in the application field, but often not so expert in computers (7). The designer must have at least general knowledge about the properties of the intended range of target machines and their environments. He/she should not only to be able to properly design the software product itself, but also to provide the material needed to support the installation process. Without this knowledge, the designer may very possibly make fundamental errors in design, and overlook small, but crucial matters affecting portability.

Unlike the designer, the installer may not be able to study the software product in order to understand its portability before he/she receives the product. The installer has much less flexibility and fewer options than the designer. He/she must accept what is provided and do the best to make the installation efficient.

Information must be exchanged in the installation process, this includes program material and supporting documentation being provided for the installer by the designer. However, there is frequently a discrepancy between what is offered and what is needed. Usually what is missing is the implicit knowledge that the designer has about the transportable software product and it is often not made explicit to the installer. Two-way communication between the designer and the installer can speed installation, especially for products which are defective in design or documentation (6).

Techniques for Enhancing Portability

The techniques for enhancing portability are primarily those generally applicable in software development. Because of the nature of the installation process, the emphases are some what different. Listed in the following subsections are desirable qualities which the software product should posses to attain portability.

1. Simplicity

A good software product should meet the "simplicity" criterion. The ideal is, after the user has read the manual he/she is able to employ the product for application without further reference to any written document. In practice, this goal has not yet successfully been attained. Something a designer can do is concentrate all the necessary information for normal use of the product onto a single page.

Another important aspect is that the designer does not try to write complex packages providing a wide range of services and satisfying all user's fantasies. Instead, only those performance requirements and design features necessary to meet the requirements of usability should be included in the design. Extra features that might be nice to have, but are not really needed can be deleted (12).

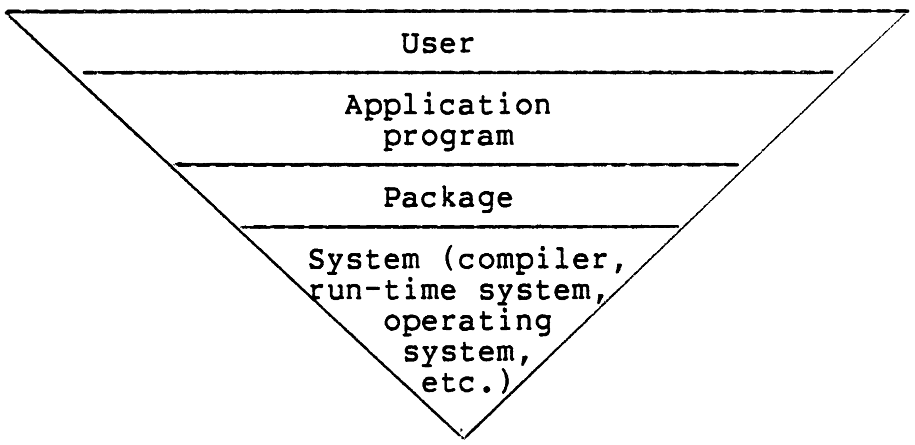
2. Self-contained

A characteristic of a portable program is that it be self-contained. The program should not rely on the existence of some external agency to supply required functions. In practice, complete self-containment is almost impossible to achieve and the designer intending to produce a portable product must compromise by isolating necessary references to the external environment. When the environment is changed, those dependent parts of the program can be easily identified and modified.

For example, the software package HP-ISPP contains 22 subroutines; 13 of them are user callable and 9 are internal subroutines which are not callable by the users. In these 9 subroutines, one called ZZINIT specifies local configuration parameters for the host processor. When the computer to which the HP-ISPP is installed changes, the installer needs only concentrate on the ZZINIT subroutine to have all the parameters changed.

We shall refer to the programs which call the software package as application programs, and users will be those individuals (or programs) who (which) run application programs

(12). The application program/ software package/ system hierarchy is pictured in Figure 1; of course other levels may exist. As can be seen, there is at least one level, that of an application program, between users and the software package, therefore, self-containment of the package is necessary. The more invisible and self-contained the package is to users the simpler the diagnosis would be if there has been an error.



Source: Myer, B., "Principles of Package Design." Communications of the ACM, Vol. 25, No. 7 (July 1982).

Figure 1. Hierarchy of Programs and Program Users

3. Use Standard Language Features

It is better to use only standard language features rather than to use a dialect unique to a particular installation. A program written in one dialect may have to be modified somewhat before being processed by a compiler for another computer or operating system. The work involved in implementing a package on more than a single system is significantly increased if non-standard language 'extensions' are used in the initial coding of the package.

It is suggested that the source code of a package be restricted to ISO (International Standard Organization) or ANSI (American National Standard Institute) standard to improve the portability (17).

4. Structured Internal Design and Others

If it is inevitable that some modifications be done when installing a software product, the readability and understandability of the product's source program usually makes the task easy. Methods to make a package easy to understand include dividing the package program into several smaller modules, using structured programming techniques, and organizing the documentation in terms of simplicity and clearness.

There are many structured programming techniques. In some languages, for example, PL/I, PASCAL, etc., structured programming means programming without GOTO's, in others it may mean top-down design.

Meyer (12) in his paper says that subprograms in his package written in Fortran are subroutines rather than functions, at least for the following reasons:

a. In many systems, Fortran functions cannot be called from COBOL programs while subroutines can.
b. A function type must be declared in the calling program, except when it is an integer or single-precision real and

follows the Fortran default rule. This would be a source of error in systems with no checking at link or load time.

This suggestion of Meyer needs to be considered in regard to the portability of packages written in Fortran.

Package Installation Process

As we know a detailed step-by-step installation manual is very helpful. Such manuals are not easy to write, but they are worthwhile. The installer must gain full understanding of the installation manual before he/she starts installing the product.

Since there are a bewildering variety of physical devices, recording techniques, tape densities, file formats, and so on, only general procedures of installation will be described:

Step 1: Loading or mounting the package:

For the products residing in magnetic tape media, load the package distribution tape files into appropriate disc files. For those residing in flexible diskette media, mount the diskette.

Step 2: Adapting the package to its environment:

For example, many parameters describing the host processor of the software package needs to be modified.

Step 3: Compiling the package subprograms:

The package subprograms may need to be compiled into a single object file. Some bugs may appear in this step.

Step 4: Preparing a relocatable library file containing the package module.

Step 5: Debugging and testing problems:
This step is usually the most difficult one, especially if the product and the computer are not compatible. It will be discussed extensively in Chapter IV.

Step 6: Documenting:
The installation procedure and the modifications made during the installing process should be recorded. This would make the future maintenance much easier. Chapter IV will have more detailed discussion about it.

CHAPTER IV

TESTING, DEBUGGING AND DOCUMENTING OF THE SOFTWARE PRODUCT

Overview

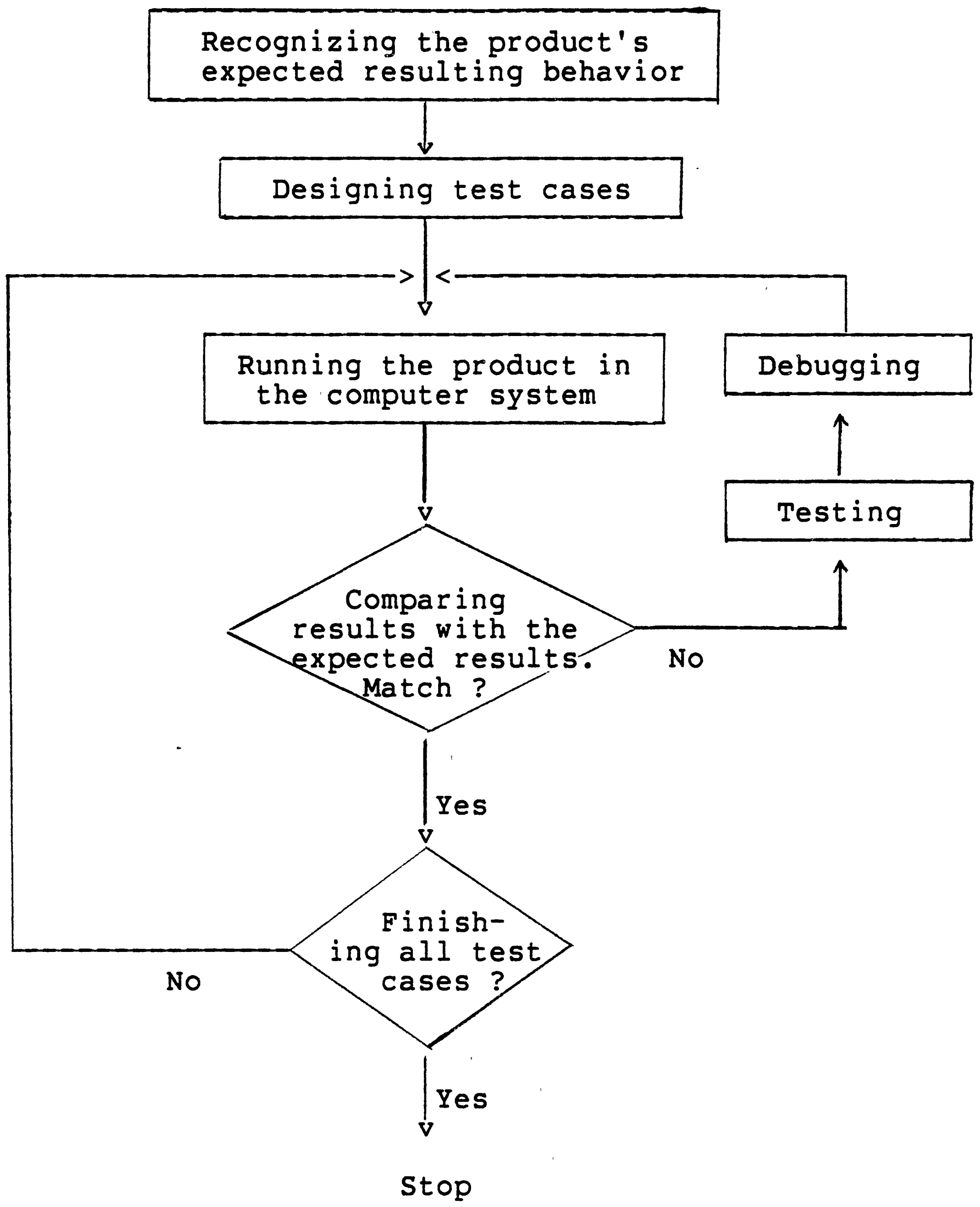
It is not desirable to use a software product without testing it after the product is installed. Although we assume the product was thoroughly tested before it ws delivered by the manufacturer, unless the product is designed for some specified computer configurations and is installed into those computer systems, there are almost always some errors.

Myers (14) gives the definition of testing as follows: "Testing is the process of executing a program with the intent of finding errors." (p.5) It is noticeable that the goal of testing is to uncover an error, and it is always possible that undetected errors exist even after the most comprehensive testing. Therefore, it is important to realize that testing can never show that a program is correct (19).

Software testing can be very complex depending on the project's size and characteristics. Researchers claim that testing should be incorporated into each phase of a software development project, not just isolated in the final stage. The later errors are found the more costly they are to correct (1, 2, 18). It is sometimes considered that program testing and debugging are the same thing. Although closely related, they are actually distinct processes. As presented above, testing is the process of establishing the existence of program errors; while debugging is the process of locating where these errors occurred in the program and removing them (19, 21). There are three causes of an unsuccessful installation: human errors, hardware errors, and software defects. Before the installer starts testing and debugging software products, he/she must fully understand and do what the installation manual says. By doing so, he/she can avoid most errors caused by human mistakes (4). Next he/she must go through the checking steps of the hardware portion's installation to eliminate the hardware's errors, then do the testing and debugging work.

All testing methods involve determining the expected behavior of the program, actually or conceptually executing the program and observing its behavior, and finally comparing that behavior with the expected behavior (1). Testing a software product starts with recognizing the expected behavior of the product, running the product with a set of designed test cases in the computer into which it is installed, and then comparing the resulting behavior with the expected behavior. If they match, the installation is said to be successful. When the resulting behavior is not the same as what was expected, one starts testing the program.

Fig. 2 shows the procedure for testing a product.



Recognizing the product's expected resulting behavior
Designing test cases
Running the product in the computer system
Debugging
Testing
Comparing results with the expected results. Match ?
No
Yes
Finishing all test cases ?
No
Yes
Stop

Figure 2. Procedure for Testing a Software Product

Test Case Design

To attain the primary objective of testing, which is uncovering errors in the program, one may think that passing through every possible path in the program is a good method. The problem is that this method, which is called exhaustive testing, is often not feasible because of the infinite number of paths in a program (1, 14, 17). Consider a cited example (14): A short program with a loop having up to 20 iterations in it, and within this loop are a set of nested IF statements. Shown in Fig. 3 is its control-flow graph. The number of its possible paths is $5^{20} + 5^{19} + \ldots + 5^{1}$, approximate 10^{14}.

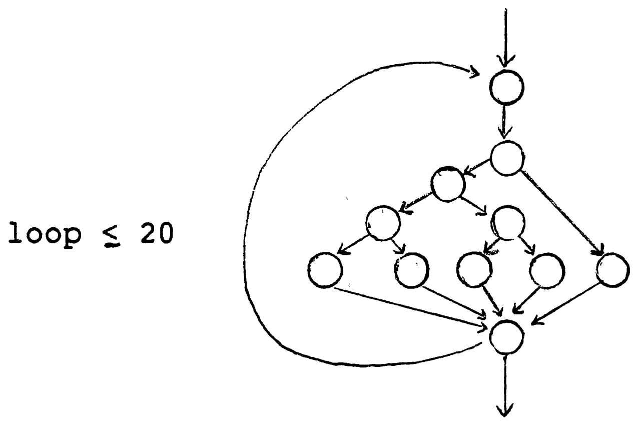


Figure 3. Control-flow Graph of a Program

If a processor can develop a test case, execute it, and evaluate the results in one millisecond, the processor would take 3170 years to test this program. From this example we can easily understand the importance of designing effective test cases when testing a program. In the next section some practical test case design methods will be introduced.

Black-Box vs. White-Box Method

In black-box testing, the tester is unconcerned about the internal organization, logic, control or data flow of the program being tested. He/she views the program as a black box. The opposite method, namely, white-box method, involves the tester's understanding of the internal structure of the program.

Black-box testing. If the function of the software program is known, one can conduct a black-box test which will demonstrate whether or not the function is fully operational. This method is used to design test cases which can demonstrate that input is properly accepted and then output is correctly produced; or input is invalid and output is erroneous. To find all errors in the program, one may consider using every possible input condition as a test case, which is exhaustive input testing, but very often that exhaustive input testing needs the tester to produce virtually an infinite number of test cases, and this is impossible in practice (1, 9, 14, 17). Since it is impossible to produce an infinite number of test cases, one has to search for a way to test a program which is equivalent to exhaustive input testing. "Equivalent" here is in a sense that it is reliable and valid; Goodenough and Gerhart (5) give explanations of reliability and validity as follows:

> In general, reliability refers to the consistency with which results are produced, regardless of whether the results are meaningful.... Validity, in contrast to reliability, customarily refers to the ability to produce meaningful results, regardless of how consistently such results are produced (p.19).

Three test data generation techniques, namely, equivalence partitioning, boundary-value analysis, and cause-effect graphing, which are considered "equivalent" to the exhaustive input testing are introduced in the following sections.

1. Equivalence partitioning

Since exhaustive-input testing of a program is impractical, one is limited to a small subset of all possible inputs. Selecting the subset which has the highest probability of finding the most errors becomes important. The principle of equivalence partitioning is that the input domain of a program can be partitioned into a finite number of equivalence classes such that a test of a representative of each class is equivalent to a test of the entire class. That is, if one test case in an equivalence class detects an error, all other test cases in the equivalence class are expected to find the same error.

The partitioning of input domain of the program is not easy. It depends on the requirement, the program domain, and the problem understanding of the tester. Myers (14) thinks that it is a heuristic process. The basic way to identify equivalence classes is to consider each input condition and partition it into at least two groups: valid equivalence classes and invalid equivalence classes. The former group represents valid input to the program, and the latter one represents all other possible states of the input condition. For instance, if an input condition specifies that the input value X is an integer and $a<X<b$, two groups of equivalence classes can be identified: one valid equivalence class ($a<X<b$), and two invalid equivalence classes ($X<=a$ and $X>=b$).

2. Boundary-value analysis

Boundary-value analysis leads to a selection of test cases that exercise bounding values. Many software errors occur just below, at, or just above the bounding value of indices, data structures, and scalar values. Therefore, test cases that explore boundary conditions have a higher probability for uncovering errors than test cases that do not.

This method differs from equivalence partitioning in the following respects:

a. Boundary-value analysis requires that one or more elements be selected so that each edge of the equivalence class is the subject of a test; while equivalence partitioning randomly selects on element in an equivalence class as a representative.

b. Instead of just considering the input conditions, its test cases are also derived by considering the output equivalence classes.

Consider the same example used in the equivalence partitioning subsection where the input value X is an integer and is specified as a<X<b. Boundary-value analysis would select X=a+1, and X=b-1 to represent valid input classes, and X=a and X=b as invalid input classes. If the outputs are expected to be greater than 0, considering the output equivalence classes, boundary-value analysis would also select the input values that can drive the outputs to be greater, less than, and equal to 0, respectively.

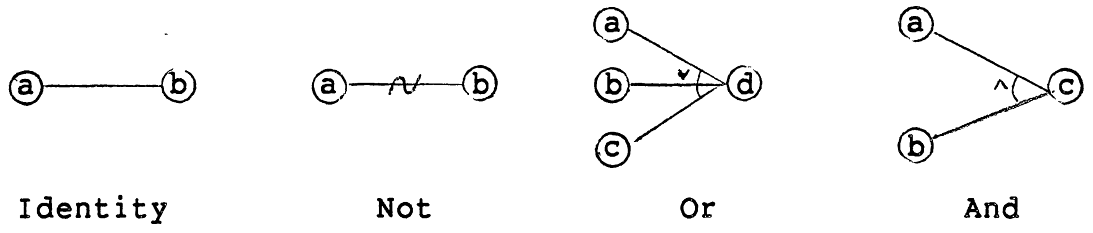
3. Cause-effect graphing

Cause-effect graphing is a technique for developing test cases for programs from the high-level specifications. It provides a concise representation of logical conditions and corresponding actions (1, 14, 17). This technique follows four steps:

1) List causes (input conditions) and effects (actions) for a module, and assign each an identifier.
2) Develop a cause-effect graph.
3) Convert the graph to a decision table.
4) Convert decision table rules to test cases.

One may refer to Myers (14) for details of these steps.

Fig. 4 shows the basic cause-effect graph symbols. Notice that nodes on the left-hand side represent causes, and those on the right-hand side represent effects.



Source: G. J. Myers, The Art of Software Testing (1979).

Figure 4. Basic Cause-effect Graph Symbols

Cause-effect graphing explores combinations of input circumstances, while the previous two techniques do not. It requires the translation of a specification into a Boolean logic network; this translation gives one an additional insight into the specification, and is a good way to uncover ambiguities and incompleteness in specifications. But it is difficult to convert the graph into the decision table. The whole process of cause-effect graphing is inefficient in every respect; especially when testing a software product which usually does not bother to perform such a series of complex tasks.

White-box testing. This method involves the understanding of the internal structure of the program being inspected. Generally speaking, it is more complex than black-box testing. Software products are usually lengthy, and their purpose is to serve the end users so that they can use the product without understanding its internal logical structure. Therefore, one generates test cases by the white-box testing method only when the black-box testing method is used and it uncovers errors, but can not help locating errors in the product.

White-box testing concerns the degree to which test cases exercise or cover the logic of the program. To test the program structure completely, the test data chosen should, ideally, cause the execution of all paths. Because many paths in a program are not finite, as explained before (see page 22), some other methods with similar spirit to exhaustive-path testing are studied.

1. Statement coverage method

This method is concerned with the generation of the test data set whose execution would cover every statement of the program. That is, by using this data set, every statement will be executed at least once. Some researchers claim that it is a weak method to detect errors in the program (5, 14). The following example shows its weakness. Suppose

```
          IF((A ≥ B) & C=0) THEN X=Y;
```

is a statement in a PL/I program, and its Boolean operator '&' should be '|'. If the test data satisfy the condition that A>B and C=0, then the statement is executed, but we can see the error is undetected. So the decision coverage method is proposed.

2. Decision coverage method

This method requires that the test cases are designed such that each decision has a true and false outcome when executing the program. There are still some weakness within the method; for instance, using the same statement used in the statement coverage method section as an example, the '&' still was intended to be a '|'. If two test cases are designed so that they satisfy the following conditions respectively:

1) $A < B$, $C \neq 0$

2) $A = B$, $C = 0$

then the result in the decision outcome for 1) is false, and 2) is true. These results are the same as those of testing

```
IF ((A ≥ B) | C=0) THEN X=Y;
```

That is, using these two test cases to test an erroneous statement one gets the same results as that when he/she tests the correct statement. Obviously, the error is not detected by using this method.

3. Condition coverage method

In this method, one designs enough test cases to make each condition in a decision take on all possible outcomes at least once. Again we use the same example, but add one test case as follows:

3) $A > B$, $C = 0$

thereby, each condition in a decision takes on all possible outcomes at least once. We still find that the error is undetected.

4. Combination of decision and condition coverage method

This method requires sufficient test cases to ensure that each condition in a decision takes on all possible outcomes at least once, and each decision takes on all possible outcomes at least once. A set of test cases for previous example is shown as follows:

1) $A > B, C = 0$

2) $A > B, C \neq 0$

3) $A = B, C = 0$

4) $A = B, C \neq 0$

5) $A < B, C = 0$

6) $A < B, C \neq 0$

By using this set of test cases, which were designed by the combination of decision and condition coverage method, the error is detected. Goodenough et al. further suggest using condition tables to analyze condition combinations (5).

It seems that we have come out with a good test-case-design strategy, but it is widely admitted that no testing can be thorough enough to test all the errors in a program. Methods introduced in this white-box testing section are not able to detect coincidental correctness, or missing path errors; this is just one of the many examples (22).

Testing Techniques

It is easy to realize that the objective of testing a software product is to make sure that the product works as expected in the computer system into which the product is installed. This is analogous to removing all the bugs in the product. For this special objective, the techniques adopted to test a software product are not as numerous as those adopted to general software development projects.

Static vs. Dynamic Analysis

Static analysis may involve some form of conceptual execution, but not actual program execution, whereas dynamic testing does involve the actual execution of program code.

Static Analysis: Desk checking, code inspection and walk-throughs are examples of static analysis. Desk checking is a method in which a person reads a program, checks it with respect to an error list, and/or walks test data through it (14). Code inspection also involves a step-by-step reading of the program, with each step checked against a predetermined list of criteria, but it is done by a team. Walkthrough involves a person leading the team through a manual simulation of the program, but teams are composed of a software designer and other persons who are involved in the program's implementation. These manual methods are found to be effective in finding from 30% to 70% of the logic design and coding errors in typical programs (15). They are usually performed during the product construction stage. If it is a small sized product, or a product at retesting stage, and the testing object is already narrowed down, then these methods are applicable.

Two other methods, which are different from previous ones, are also classified by Adrion et al. (1) as static methods. They are flow analysis and symbolic execution.

Flow analysis consists of data-flow and control-flow. Both methods use graphical representation. In control-flow analysis, the program graph has nodes, representing a statement or segment, that possibly end in a branch predicate. The edges represent the allowed flow of control from one segment to another. The control-flow is used to analyze the program behavior, to locate instrumentation breakpoints, to identify paths, and to perform static analysis activities. Errors about unreachable program statements are usually detected by control-flow analysis. In data-flow analysis, each node corresponds to a variable, and the edges indicate the dependence between variables. By tracing the behavior of program variables as they are initialized and modified during the program execution, data-flow analysis is able to discover program anomalies such as undefined or unreferenced variables, or inconsistent interfaces among modules, etc.

Symbolic execution uses variable names that hold the

```
        INTEGER I,J,X
        READ,  I, J
        X = I+J
        IF (X.GE. 0) GO TO 10
        X = X * X
        GO TO 20
  10    X = X + X
  20    STOP
        END
```

Figure 5. A Fortran Program

input values, instead of actual data values, as input values. The effect of assignments during a symbolic execution is to replace the value of the left-hand side variable by the unevaluated expression on the right-hand side, and all variable manipulations and decisions are made symbolically. For example, symbolic execution the Fortran program in Fig. 5 will result in the following expressions:

```
if (I+J) ≥ 0 then X = (I+J)+(I+J)
else X = (I+J)*(I+J)
```

As we can see, if the program is a little larger or more complicated, the result of its symbolic execution may become strings of complex expressions. In addition, all decision points are indeterminate; that forces the execution to go through every possible path and makes the program very lengthy and difficult to compute. In general it was found to be difficult to apply symbolic execution to all but the modules at the lowest level of the software program. A symbolic execution is considered to be reliable in catching errors if the symbolic output for a selected path revealed the error in an obvious way. If the output is erroneous but is in the same symbolic form as it appears in the path, then the errors are not obvious for the tester and are very possible to be missed. This reveals the unreliability of the symbolic execution for catching errors. Although the method has the previously mentioned drawbacks, it can be relied on for catching some subtle errors. For instance, the same sequence of statements may compute correct answers for some data but not for others, and the symbolic execution can

```
Original program:            IF(X.GT.0) GO TO 20
                             DO 10 J=M,N,I
                                  .
                                  .
                     10      CONTINUE
                             GO TO 30
                     20      statement
                                  .
                                  .
                     30      statement


Program inserted             KTBRNH=0
with counters:               IF(X.GT.0) GO TO 20
                             KTLOOP=0
                             DO 10 J=M,N,I
                             KTLOOP=KTLOOP+1
                                  .
                                  .
                      10     CONTINUE
                             GO TO 30
                      20     KTBRNH=KTBRNH+1
                             statement
                                  .
                                  .
                      30     statement
```

Figure 6. A Program Inserted With Counters

catch this error (8).

Dynamic Analysis. The dynamic analysis procedure usually includes static analysis and actual program execution. Adrion et al. (1) considered program execution as instrumentation of the program, execution of the instrumented program, and analysis of the instrumentation data.

Instrumentation of the program means "to tell what's going on inside it" (13, p.4). One instrumentation technique is inserting codes at appropriate places in the program, for instance, inserting a counter at a branch predicate, inside or outside of a loop construct. The Fortran program in Fig. 6 is an example in which KTLOOP is a loop counter and KTBRNH is a branch counter.

Code insertion techniques also can make the maximum and minimum values of variables, the initial and final values, etc., transparent to the observer. During the testing process of installing the previously mentioned package HP-ISPP, the installer has used this technique very often, and located a potential area of errors by observing the change of some variables' values.

An alternate implementation is to insert calls to routines in place of actual counters. Some commands are also inserted in the code. The instrumentation is enabled when the correct commands are set. Stucki (20) introduced another similiar method with dynamic assertions. These assertions can be considered as comments. But when the commands are set, the specific assertions are enabled and the preprocessor generates the instrumentations.

Techniques about execution of the instrumented program and analysis of the instrumentation data are numerous. They will be included in the following sections which are of different technique categories.

Incremental vs. Non-incremental Testing

One of the characteristics of software products is that it is large - tens of thousands of lines of code are usual (7). To ease the testing, experienced programmers all know that it is better to test the smaller building blocks of the program than to test the program as a whole. The process of testing the individual subprograms, subroutines, or procedures in a program is called module testing. In this section, two approaches to performing the process of module testing, namely, incremental and non-incremental approaches are discussed.

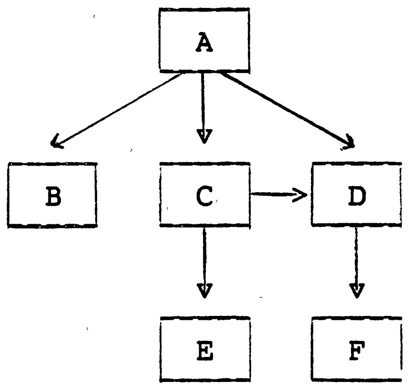


Figure 7. Hierachy Structure Between Modules in a program

Incremental Testing. If one combines the next module to be tested with the set of previously tested modules before it is tested, this is incremental testing. Consider the program in Fig. 7 as an example. Each rectangle represents a module, and each arrow specifies the control hierachy between modules. In this diagram, module A calls module B, C, and D; module C calls modules D and E; and module D calls module F. Incremental testing starts either from the top module A, or bottom module F. Once satisfied with the testing of this first module the next module, in this case, B or D, respectively, to be tested is combined with the first module, then tested as a system. The process continues until all six modules have eventually been integrated into a complete system. If a module is introduced at some stage in this process and tests which previously did not detect system errors now detect system errors, it is certain that these errors are due to the introduction of the new module. The source of the error is localized, which simplifies the task of debugging.

When one discusses incremental/non-incremental testing, the concept that the testing of each module requires a special driver module and one or more stub modules must be introduced. For instance, to test module F in Fig. 7, a driver module must be set. It accepts test case data, passes such data to module F, and displays the results produced by F. While testing module A, it is necessary to set three stub modules which simulate the functions of module B, C, and D, to receive control from A.

There are two philosophies, namely, top-down and bottom-up, for performing incremental testing. The top-down method starts with testing the top module, for example, module A in Fig. 7, and proceeds with its subordinate modules, in this example, B, C, and D, and their subordinate modules. To accomplish this, one must first write stub modules representing B, C, and D, and these stubs are expected to return meaningful results to A. After the top module has been tested, one of the stubs is replaced by an actual module, and the stubs required by that module are added. If they are tested sequentially, many sequences are possible. For instance, ABCDEF or ACEDFB, etc. are possible sequences for the program in Fig. 7. If they are tested in parallel, some other alternatives are possible, and several programmers may be involved. For instance, one programmer tests the combination A-B, another tests A-C, and the third tests A-D.

The bottom-up method starts with the terminal modules, which call no other modules in the program. Terminal modules in Fig. 7 are modules B, E, and F. These modules may be tested serially or in parallel. To do so, each module needs a special driver to supply inputs, call the module to be tested, and display the outputs. No multiple versions of a driver are needed; even the module being tested may have several superordinate modules, because the driver can iteratively call this module; but in top-down strategy, if a module being tested has several subordinate modules, it is usually necessary to set multiple versions of the stub.

It is widely admitted that creation of drivers is easier than creation of stubs (14, 19), but it does not necessary mean the bottom-up is better. One of the reasons is that when adopting the bottom-up strategy, if errors exist in high level modules they will not be detected until a later stage in the incremental test (10). In the former situation making corrections will involve rewriting and consequent retesting of lower level modules. For testing a software package, if one suspects the errors are hidden in higher level modules, adopting the top-down method can be advantageous; otherwise, adopting the bottom-up method is efficient.

Nonincremental testing. For nonincremental testing, one tests a program by testing each module independently and then combining the modules to form the program. Consider the program in Fig. 7 as an example. First, at least five driver modules and six stub modules must be prepared. Next, test each module, with a necessary driver and/or stubs, as a stand-alone entity. Finally, the modules are combined to form the program.

At the module-testing phase, nonincremental testing can test all modules at the same time, which is efficient. The disadvantage of nonincremental testing is that if there is an error related to intermodule interfaces, the error will not be uncovered until the entire program has been combined. This drawback sometimes makes pinpointing the error difficult.

In testing a software product, it is usually reasonable to assume that the product itself is free of error if it is not installed in the computer system, since it had been tested thoroughly before being released form the manufacturer. If errors are revealed after the software product is installed, one may be right in assuming that the errors come from product modules which have connections with the environment. Therefore, one should perform the module testing only on the selected modules which have connections with the environment, then perform the nonincremental testing for a complete testing process. If the relationships between the modules in the software package are complicated then it may be necessary to employ incremental testing. It is meaningless to conclude that one method is superior to the other for testing a software product, because it is a case-dependent matter.

Debugging Techniques

Debugging includes two processes, locating the error and fixing the error. It is an activity which closely follows the testing. Sometimes debugging and testing are performed rotationally until no more errors are found.

Since fixing errors is a program, or a programmer dependent matter, the purpose of this section will be to focus on the techniques of locating errors. Three categories for debugging techniques are proposed as follows:

1. Brute force

This category includes many methods: core dumps, symbolic dumps and insertion of print statement in the program. The former two methods are not efficient, because they dump massive amounts of data to be analyzed. Many times the dump is not produced at the exact point of the error, which does not help much in locating errors.

Inserting print statements which output important data values at appropriate places in the program is superior to the other two methods. It is efficient in isolating errors, especially when debugging a software product.

2. Cause elimination

Cause elimination is manifested by induction or deduction. The summary of their steps is as follows (14):

Steps for induction:

1) Locate the pertinent data.
2) Organize the data.
3) Devise a hypothesis.
4) Prove the hypothesis.

Steps for deduction:

1) Enumerate possible causes or hypotheses.
2) Use the data to eliminate all but one of the possible causes.
3) Refine the remaining hypothesis.
4) Prove the remaining hypothesis.

3. Backtracking

Backtracking is considered effective for locating errors of small programs (14, 17). It starts at the point in the program where the incorrect result was produced, then the source code is traced backward manually until the cause is found.

Debugging the software product usually starts with erroneous output, then one uses backtracking combined with cause elimination and inserting print statement methods. If the debugging tool is available in the computer system, one can even use that tool to do the job. For instance, under UNIX, a debugging program is called adb; adb allows the programmer to request the values of variables by names, to exam the contents of the machine registers and to display the results in various formats.

Documentation

In the final phase of installing a software product, documentation is unexciting but essential. The installer may think that finishing the testing and debugging of the product means the installation process is finished; yet, it is not. The reason the installer must pay attention to the product's documentation is that maintainability is considered to be one of the most important characteristics of a software system, and it can be accomplished only if the software system includes sound documentation (17, 19).

The documentation of a software product can be organized into two categories: user documentation and logic documentation. User documentation is made up of those documents which relate to the functions of the product, without reference to how these functions are implemented. Logic documentation, on the other hand, describes all aspects of the design, implementation, and testing of the product.

User documentation usually contains these documents:

1) A functional description, which outlines what the system can and can not do.
2) An installation guide which describes how to install the product and modify it for particular hardware configurations.
3) An introductory manual which explains how to use the product.
4) A reference manual which describes in detail the facilities available to the user and how to use these facilities.
5) An operator's guide (if the software product requires an operator), explaining how the operator should react to situations which arise while the product is in use.

These documents are usually separated from the source code of the software product.

Logic documentation should describe:

1) An overall product specification showing how the requirements are decomposed into a set of modules, and what the global variables are in the program.
2) The functions of each module, and what the local variables are in the module.
3) Some acceptance test cases, which serve as the criteria for a successfully installed and usable product.

They may be written as either internal or external documents, depending on which one will make the program more understandable and more convenient for the users to refer to.

The task for the installer is to update these documents. During the installation process if he/she finds that there are some omissions in the documentation, he/she should fill them. No matter what the modifications are, whether they are major or minor, in the source code of the product, in the installation instruction, or in the test cases, etc., it is the installer's responsibility to record the modifications, and upgrade these documents. After the installer has fullfilled that responsibility then, if there are any new errors revealed, the end user can have a faster fixation since he/she has got clues from the documentation.

The installation process is completed when the documentation updating, which follows the testing and debugging, is finished.

CHAPTER V

SUMMARY, CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

Installing a computer product is a task that is frequently encountered by the computer owner, who may be a person, a school, an agency, etc. If installation is in the computer system development plan, it usually has fewer problems compared to one which is not in the plan. Although manufacturers of products supply installation manuals, most of the time if not always, it is buyers who actually do the installing job and face the problems. This chapter summarizes a systematic approach to installing a computer product. It also summarizes the techniques used in testing and debugging which are two important procedures during the installation.

Approach to Installing the Hardware

Preparing a suitable environment before the product arrives is important. After the product arrives, it is necessary to perform the following steps:

1. Perform static check-out
   a. Inspect the device and accessories for any physical damage.

   b. Adjust the interface between the product and the computer, make sure they are compatible in at least four aspects: mechanically compatible, electrically compatible, use same data representation, and same communication speed.

2. Perform dynamic standalone check-out
   a. Turn on the power.
   b. Refer to the manual supplied by the manufacturer to check the device's functions.
   c. Perform the confidence test.
3. Perform dynamic integrated check-out

Integrate the product with the other devices in the specified computer system and check the product's functions.

Approach to Installing the Software

A portable software product is a program which can be transferred from one computer system to other systems without requiring much effort to modify the program. Therefore, portability is an important characteristic for a software product. The product designer plays a more important role in obtaining portability than the installer does. The designer must have general knowledge about the properties of the intended range of target machines, and explicitly transfer that knowledge to the installer by offering the installer a complete installation manual.

Techniques for Enhancing Portability

1. Keep the product simple but include all necessary functions.
2. Write the program in a self-contained manner.
3. Use standard language features.
4. Use structured programming techniques.

Process for Installing a Software Product

1. Read through and understand all of the installation manual.
2. Load or mount the product.
3. Adapt the product to its environment.
4. Compile the subprograms of the product.
5. Prepare a relocatable library file containing the product modules.
6. Test and debug the product.
7. Update the documentation.

Testing, Debugging and Documenting

Testing, debugging and documenting are works performed in the later stages of software installation. They may require lots of the installer's efforts. Testing and debugging are hot topics in the software engineering field. The purpose of testing is to uncover errors in the program; the purpose of debugging is to locate and correct the errors. Although the testing and debugging of the software product are not as complex as those of the software development project, it still demands good techniques to accomplish the efficiency. The procedures of testing a software product are:

1. Recognize the product's expected resulting behavior.

   This includes recognizing what the correct results are when inputs are valid, and what they are when inputs are invalid.

2. Design test cases.

   a. Black-box method

      The programmer sees the product as a black box. He/she designs valid inputs and invalid inputs to test the product. Methods in this category are: (1) equivalence partitioning, (2) boundary-value analysis, and (3) cause-effect graphing. The last one is a little complex for testing a software product; the first two methods are used most often.

   b. White-box method

      The programmer needs to understand the internal structure of the software product being tested. This demands more effort than the black-box method does, but this method is able to give more clues about the nature of the errors in the product's source code, while the black-box method sometimes can not.

      The white-box method includes the following methods: (1) statement coverage, (2) decision coverage, (3) condition coverage, and (4) a combination of decision and condition coverage. The last method is considered to be the best among the four.

3. Run the product in the computer system.

4. Compare results with the expected resulting behavior.

When two results do not match, the tester needs to perform the step 5 to locate the errors. When they match, the tester can go on to test the next case until finishing all test cases.

5. Test the product.

When one suspects the product have errors, he/she may need to use some techniques to reveal those errors. The techniques applicable for testing a software product are summarized as follows:

a. Static analysis

It includes desk checking, code inspection, and walkthroughs, data-flow and control-flow analysis, and symbolic execution. The first three methods are manual methods; and they can be used for the small-sized product, or when the possible range of errors in the product is limited to certain modules. The flow analysis methods involve graphical representation. Control-flow analysis can detect unreachable program statements, while data-flow analysis can discover program anomalies more easily. The symbolic execution method is reliable in catching errors which may not be caught if the inputs are real values.

b. Dynamic analysis

It involves the actual program execution, while the static method involves conceptual program execution at best, or perhaps not any execution at all. Dynamic analysis includes the following aspects:

(1) Instrumentation of the program.

Techniques involved are code insertions, and call insertions, etc.

(2) Execution of the instrumented program.

(3) Analysis of the instrumented data.

Techniques for the second and the third aspects can be divided into incremental and non-incremental categories. There are two philosophies for incremental techniques, one is top-down, the other is bottom-up. One adopts top-down strategy when the errors are suspected to be hidden in higher level modules, otherwise, one adopts bottom-up strategy.

Debugging Techniques

Testing and debugging are rotationally performed as one big procedure in the process of installing a software product. But since their objectives are different, as stated in the preceding section, their techniques are different. A summary of debugging techniques is as follows:

1. Brute force

It includes core dumps, symbolic dumps, inserting print statement. Among them, only the inserting print statements method is considered efficient.

2. Cause elimination

Use either induction or deduction.

3. Backtracking

Trace from erroneous results to the source code of the product.

Conclusions and Suggestions for Further Research

The hardware installation is more straightforward than the software installation, and whether the software installation is efficient or not depends heavily on whether the testing and debugging is efficient. There are numerous techniques for testing and debugging, and each technique has its own characteristics. It is hard to conclude which technique is the best. Only the combination of the installer's knowledge, the software product's characteristics, and the computer system into which the product is to be installed can make a fair judgement possible.

A SELECTED BIBLIOGRAPHY

(1) Adrion, W.R., M.A. Branstad and J.C. Cherniavsky, "Validation, verification and testing of computer software." Computing Surveys, Vol .14, No. 2 (June 1982), 159-192.

(2) Boehm, B.W., "Seven basic principles of software engineering." Softtware Engineering Techniques. Infotech State of the Art Report, Infotech, London, 1977.

(3) Freedman, M.D. and B.E. Lansing, Designing Systems with Microcomputers: a Systematic Approach. Prentice-Hall, Inc., Englewood Cliffs, N.J, 1983.

(4) Gilb, T., Reliable EDP Application Design. Petrocelli Books, N.Y. City, 1974.

(5) Goodenough, J.B. and E.I. Gerhart, "Toward a Theory of Test Data Selection." IEEE Transactions on Software Engineering, (June 1975), 156-173.

(6) Griswold, R.E., "Engineering for Portability." Software Portablity, an Advanced Course. Cambridge University Press, Cambridge, London, 1977.

(7) Hamlet, R.G. and R.M. Haralick, "Transportable Package Software." Software-Practice and Experience, Vol. 10 (1980), 1009-1027.

(8) Howden, W.E., "An Evaluation of the Effectiveness of Symbolic Testing." Software-Practice and Experience, Vol. 8 (1978), 381-397.

(9) Huang, J.C., "An Approach to Program Testing." Computing Surveys, Vol. 7, No. 3 (Sep. 1975), 113-128.

(10) Lauesen, S., "Debugging Techniques." Software-Practice and Experience, Vol. 9 (1979), 51-63.

(11) Longbottom, Roy, Computer System Reliability, J. Wiley ltd., 1980.

(12) Meyer, B., "Principles of Package Design." Communications of the ACM, Vol. 25, No. 7 (July 1982),

419-428.

(13) Miller, E. "Introduction to Software Testing Technology." Tutorial: Software Testing & Validation Techniques, 2nd ed., IEEE Computer Society. 1981, 4-16.

(14) Myers, G.J., The Art of Software Testing, John Wiley & sons, Inc., 1979.

(15) Myers, G.J., "A Controlled Experiment in Program Testing and Code Walkthroughs/inspections." Communications of the ACM, Vol. 21, No. 9 (1978), 760-768.

(16) Poole, P.C. and W.M. Waite, "Portability and Adaptability." Software Engineering, An Advanced Course, Lecture Notes in Computer Science. Springer-Verlag, 1975.

(17) Pressman, R.S., Software Engineering: A Practitioner's Approach. McGraw-Hill, Inc., 1982.

(18) Scherr, A.L., "Developing and Testing a Large Programming Systems, OS/360 Time Sharing Option." Program Test Methods, ed. by W.C. Hetze. Prentice-Hall, Inc., N.J. 1973, 165-180.

(19) Sommerville, I., Software Engineering, Addison-Wesley Co., 1982.

(20) Stucki, L.G. "New Directions in Automated Tools for Improving Software Quality." R. Yeh (Ed.), Current Trends in Programming Methodology, Vol. II - Programming Validation, Prentice-Hall, Englewood Cliffs, N.J., 1977, 80-111.

(21) Van Tassel, D., Program Style, Design, Efficiency, Debugging and Testing. Prentice-Hall, Englewood Clilffs, N.J., 1974.

(22) White, L.J. and E.I. Cohen, "A Domain Strategy for Computer Program Testing." IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980.

VITA 2

Sue-Fong Christine Shen

Candidate for the Degree of

Master of Science

Thesis: A SYSTEMATIC APPROACH TO INSTALLING
A COMPUTER PRODUCT

Major Field: Computer Science

Biographical:

Personal Data: Born in Taichung, Taiwan, R.O.C., January 1, 1952, the daughter of Wan-Li and Guei-In Lin. Married to Zie-Chiang Shen.

Education: Graduated from Provincial Taichung Girls' Middle School, Taiwan, R.O.C., in June, 1969; received Bachelor of Science degree in Agriculture from National Taiwan University in June, 1974; received Master of Science degree in Family Relations and Child Development from Oklahoma State University in June, 1980; completed requirements for the Master of Science degree in Computing and Information Sciences at Oklahoma State University in May, 1984.

Professional experience: Social worker of Christian Children's Fund, Inc.; Taiwan, July, 1974 to December, 1977. Teacher of Stillwater Child Development Center, Stillwater, Oklahoma, January, 1980 to January, 1981. Student clerk of Admon Low Library, Stillwater, Oklahoma, May, 1981 to May, 1984.