A Procedure for Software Module Upgrades

by

Marvin Xuewen Li

B.Sc., Qinghua University, Beijing, China, 1984

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. Daniel M. Hoffman, Supervisor (Dept. of Computer Science)

[redacted]
Dr. Mantis H. Cheng, Departmental Member (Dept. of Computer Science)

[redacted]
Dr. Lyle P. Robertson, Outside Member (Dept. of Physics)

[redacted]
Dr. Warren Little, External Examiner (Dept. of Elec. and Comp. Eng.)

Supervisor: Dr. Daniel M. Hoffman

Abstract

Although considerable research has been done in the re-engineering of software systems, an automatic re-engineering process is not feasible in the near future. Human understanding, human design activities, and human decision making are still necessary elements of the process. Further, re-engineering will likely be time-consuming and expensive, making it impractical to completely re-engineer an existing system.

However, we can divide re-engineering projects into categories, and, for each category, establish standard procedures for doing the work effectively and economically. Lessons learned from mistakes will be reflected in these procedures. The procedures can be applied to other projects in the same category, thus improving productivity and reducing errors.

This thesis presents a procedure for *module upgrades*. The term *module upgrades* is used to denote the process of collecting a set of related functions into a programming *work assignment* with a well designed and precisely defined interface. Our goal has been to define the procedure precisely and allow the subject system to be upgraded incrementally.

We have developed the module upgrades procedure, including a set of guidelines and constraints on the products to be produced. To evaluate and improve the procedure, we applied it to one module in a complex control software system. We kept our cost records to support more accurate cost estimation for future upgrade projects.

Examiners:

[redacted]

Dr. Daniel M. Hoffman, Supervisor (Dept. of Computer Science)

[redacted]

Dr. Mantis H. Cheng, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. Lyle P. Robertson, Outside Member (Dept. of Physics)

[redacted]

Dr. Warren Little, External Examiner (Dept. of Elec. and Comp. Eng.)

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 TRIUMF, for their time, assistance and product. Thanks to James Uhle for providing help that made the task of writing easier. My wife has given me much support over the years and deserve particular thanks for her support on this thesis. Thank you, Yun. Financial support in the form of University of Victoria Fellowship, University of Victoria Graduate Student Supplement, and research assistantship from Dr. Daniel Hoffman made this work possible.

Chapter 1

Introduction

1.1 The problem

As computers have become more powerful, software systems have become more complicated. The problem is even worse for software systems that were developed in the past and have been maintained for many years.

The original designers may not have known any software engineering principles. The system may have been decomposed into parts according to a flowchart, yielding a bad system structure, causing significant maintenance problems. Even disciplined programmers may produce a poor quality system, if they lack software engineering training. Problems such as unstructured code, code redundancy, unnecessary complexity, and lack of documentation may all be present.

Years of modifications introduce further problems. The system requirements documentation, if any, may be inconsistent with the actual software. When new features were added to the system, the documentation may not have been properly updated. Code in different parts of the system may have become incompatible after differ-

ent maintainers introduced changes. Hidden errors are introduced and cause ripple effects.

It might seem that the best approach is to completely replace the old software. Unfortunately, this is often impractical due to limited time and manpower. Developing a complex software system is expensive. If the new system is not carefully designed, implemented, and tested, it may not be much better than the existing one. Furthermore, compatibility problems can make rewriting the software extremely difficult. Features are often documented nowhere but in the software itself. After a system has been running for years, users may have built many applications that depend on the undocumented features, and even on the errors.

In summary, over time, software systems become more difficult to maintain, but we cannot throw them away. While a great deal is known about new software development, very little is known about making improvements to existing software systems. Because the installed base is huge and because new software development is extremely expensive, methods for improving existing systems are worth pursuing. Such methods would allow us to maintain a software system and also make internal improvements.

1.2 The approach

Software re-engineering has attracted considerable attention during the past few years. Commercial tools are available to extract program control or data flows, compute complexity measurements, and perform program restructuring [Laf90]. However, they can only provide services for syntactic analysis. Because human understanding, human design activities, and human decision making are still necessary elements of the re-engineering process, the process will likely be expensive and time-consuming, making it impractical to completely re-engineer an existing system.

We can divide re-engineering projects into categories, and, for each category, establish a standard procedure for doing the work effectively and economically. As stated by Peter Freeman [Fre83], a pressing need in many organizations today is for a set of combined management and technical development guidelines that combine work product definitions, quality guidelines, and management review procedures. Lessons learned from mistakes will be reflected in these procedures. The procedures can be applied to other projects in the same category, thus improving productivity and reducing errors.

This thesis presents a procedure for *module upgrades*: the process of collecting a set of related functions into a programming *work assignment* with a well designed and precisely defined interface. These functions may be critical to the correctness of the subject system and their implementation may be widely scattered. The functions are chosen in such a way that they together provide a single service, or share access to a single data structure (or device). Based on the interface definition, other programmers can use the service provided without having to know the details of the data structure or device and the access methods employed. Hence, even if the data structure or access methods change, the rest of the program need not change. Only the work assignment that provides the service need change.

The module upgrade procedure was designed so that the upgrade process can be reviewed at each step. In order to be able to maintain the software, the maintainer must have complete representations of the information that make up any particular part of the design. If part of a design is stored only in the designer's head, we cannot reliably reuse it in the future. Our goal has been to define the procedure precisely and allow the subject system to be upgraded incrementally. Applying this procedure to a portion of a software system is not free, but the future maintenance cost for that portion may be significantly reduced.

1.3 The TICS system

We applied the module upgrade procedure to the Device Control Block List (*DCBL*) access code of the **TI**SOL Control System (*TICS*). TISOL is the Test Isotope Separator On Line facility at TRIUMF. DCBL is one of the critical parts of TICS. TICS was developed at TRIUMF to provide a low cost solution for control and data logging applications on IBM-PCs. It is geared towards control situations in a particle accelerator environment, e.g., vacuum system control, beam line control, etc. TICS provides a stand-alone control system and was designed to run under MS-DOS on an IBM-PC XT or AT or a fully compatible machine. Limited multitasking and real-time facilities are provided within TICS. The TICS software system is built around the TICSLanguage, an application-oriented data base description language. It is used to describe the configuration of a control system and its display in ASCII readable data base files. Binary tables are constructed from this data base by a compiler utility and are used by an application package to perform the actual control and to provide the user interface.

The TICS system consists of the following programs:

- the TICSLanguage compiler TICSCMPL,
- the TICS control program,
- the start up facility,
- the interpolation table builder TICSTBL,
- the log retrieval utility TICSPLOT, and
- the off-line display editor TICSEDIT.

We focused on the TICS control program (hereafter just *TICS CP*). The TICS CP does the actual system control. The TICS CP operates cyclically, performing the following tasks, as required, in each cycle.

- process keyboard commands from the user,
- read devices and update the Device Value Database (DVDB),
- update the display,
- process alarms, warnings and device consistency checks,
- perform data logging and monitoring, and
- schedule user supplied tasks.

When execution begins, TICS CP reads all the devices and initializes the DVDB with the values read. The source code of the package consists of 133 `.c` files (75736 lines), 56 `.h` files (6358 lines), 25 `.bat` files (459 lines) and 30 `.mak` files (2693 lines), totaling 244 files and 85296 lines of code.

1.4 Thesis overview

This thesis contains a discussion of related research (Chapter 2), and terminology and fundamental issues (Chapter 3). Chapter 4 presents the module upgrade procedure, defining the scope of the procedure and the work products produced at each step. It also presents criteria for judging the quality of the products. Chapter 5 discusses our application of the module upgrade procedure to the DCBL module in the TICS system. We summarize our experience in Chapter 6. A summary of our work and future research directions are discussed in Chapter 7. Appendix A contains the full set of work product criteria, and Appendix B contains the work products we produced in our module upgrade project.

Chapter 2

Related Work

Our work is related to the following areas of research: software restructuring and re-engineering.

2.1 Software restructuring

2.1.1 Concept

Restructuring [Arn86] is reorganizing a system while preserving its external behavior. It is the modification of software to make the software easier to understand and to change. Restructuring can be applied to documentation, design, and implementation. For example, to transform a poorly written program into a structured (goto-less) program is a kind of restructuring. An understanding of the meaning of the code is not necessary for many types of restructuring. They can be done just with a knowledge of structural form. Restructuring does not normally involve the modifications to a system because of new requirements. Restructuring is often used as a form of preventive maintenance. It may involve adjusting the system to meet new environmental

constraints. Note that software changes for other purposes, such as code optimization, are not included in this definition of software restructuring. Code optimization is typically for the purpose of improving program execution speed. The purpose of software restructuring is to decrease the programmer effort needed to maintain the software later on.

2.1.2 Approaches

Following Robert Arnold [Arn86], approaches to software restructuring can be divided into two categories: approaches not involving code changes and approaches involving code changes.

Sample approaches in the first category are: (1) buying a package to replace an old system [Can84], and (2) upgrading documentation, such as making comments consistent and up to date. These approaches are safer and usually cheaper. Making changes to code can cause hidden errors, which will be expensive to detect and fix. Therefore, modifications to code should be made only if the maintainer knows exactly what he (or she) is doing. The source code usually does not come with the package. This presents a flexibility problem, but it leaves the maintenance burden to the package developer. If a suitable package can be found, buying may be the best solution.

The second category can be divided further into two groups: tools and techniques. The tools group consists of approaches for which software has been written to mechanize the approaches. The techniques group consists of methodologies that are designed for, or may be applied to, restructuring software.

There are tools that take a program and automatically report which software standards the program does and does not meet. Then the program can be improved

by removing the standard violations [Arn86]. Of course, the standards must be precise enough to be checked automatically. There are also tools that takes a program and produce a formatted version. Normally the logic of the code is not changed. Program transformation systems [PS83], also considered to be tools, are based on rule-based systems of artificial intelligence. A set of rules are established and recorded. Then changes are made automatically according to these rules. For example, this approach can be used to change all lower case global variables to upper case or vice versa. This approach is limited to those situations where the rules can be explicitly stated and are precise enough to be automated. Unfortunately not many systems can be transformed this way. However, commercial tools are available to assist restructuring COBOL programs. Examples are Structured Retrofit [Lyo81], SUPERSTRUCTURE [Mor84], and RECODER [Bus85].

One typical example of approaches in the techniques group is logical retrofit with the Warnier methodology [Par84] [War78]. It is to make the program control structure analogous to the structure of the data the program operates on. Typically the data structures the program operates on are well defined and documented. By making control structure analogous to the data structure it operates on, the program will be easier to understand. Also, changes to control structure caused by the data structure changes are likely to be local. Another approach in the techniques group, called software renewal [M.84], focuses on upgrading the system documentation, system specification, and system tests. Code may also be modified.

There are techniques that are not primarily designed for software restructuring, but that can be applied to software restructuring. Approaches using software metrics (e.g., [Bas80] [HMKD82]) are sample approaches of this kind. The idea is to (1) measure the software with the software metric, (2) answer the question: "Is the software property measured by the metric satisfactory?", and (3) if not, restructure

the software and go to (1). This approach is more useful when tools are available to apply the metrics. Another approach is restructuring with a preprocessor [KP76]. The idea is to selectively restructure software with statements recognized by the preprocessor, while using the rest of the code without change.

2.2 Re-engineering

Re-engineering is the examination and alteration of a system to reconstitute it in a new form and the subsequent implementation of the new form. This may include modifications with respect to new requirements not met by the original system. It usually includes two phases: reverse engineering phase and forward engineering phase.

Reverse engineering [CI90] is the process of analyzing a system to identify its components and their interrelationships, and to present the system at a higher level or in a more abstract form. The purpose of reverse engineering is to make the future maintenance or reuse of the system easier. It generally involves extracting design artifacts and building or synthesizing abstractions that are more implementation independent. It does not change the system. It is a process of examination. There are lot of subareas of reverse engineering. *Redocumentation* [CI90] and *design recovery* [CI90] are two subareas that are widely referred to.

Redocumentation involves creating a semantically equivalent representation, as an alternate view for a human audience. Redocumentation is the simplest and oldest form of reverse engineering. Some common tools used to perform redocumentation are pretty printers, which display a code listing in an improved form; diagram generators, which create diagrams directly from code, reflecting control flow or code structure; and cross-reference listing generators. The goal of these tools is to provide easier ways to visualize relationships among program components.

According to Ted Biggerstaff [Big89], design recovery recreates design abstractions from a combination of code, existing design documentation, personal experience, and general knowledge about problem and application domains. Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, and so forth. Thus, it deals with a far wider range of information than found in the code alone.

Forward engineering is the traditional process that follows a top-down fashion from requirement specification to design and then implementation. It is to build a new system or add new functions into an existing system.

Re-engineering consists of enabling the migration, modification and/or documentation of existing software regardless of the state of the software and of its documentation. Typically, software to be re-engineered implements functionality that is of critical importance but was designed and implemented a long time ago using outdated technology. It may contain redundant and/or inconsistent code. It is unreliably documented and hard to maintain. It is expensive, sometimes impractical, to find out which part of documentation is inconsistent with actual implementation.

The current commercial tools can assist extracting program control or data flows and measuring software complexity. While useful, they can only provide syntactic services. Little or no consideration is given to the programming knowledge that went into the software or the meaning of the software in its domain of application.

Chapter 3

Key Concepts and Terminology

We divide the key terms and concepts into two groups. One group consists of terms and concepts related to the process of developing new software systems and the other terms and concepts related to improving the quality of an existing system.

3.1 Building a new software system

Following Parnas [PC86], our software development process consists of the following phases: (1) requirements analysis, (2) system decomposition, (3) module interface design, (4) module implementation, and (5) module testing.

The resulting documentation or source code are called *work products*. For example, the document recording the detailed requirements of a system, called *requirements specification*, is a work product. For each work product P, we have developed *work product criteria* [Hof90a] of P. Sample work product criteria can be found in Appendix A.

3.1.1 Requirements analysis

The requirements analysis phase defines the functionality of the system: the characteristics that will be externally observable. Its output document is the *Requirements Specification.* The ideal requirements specification contains everything you need to know to write software that is acceptable to the customer, and no more.

3.1.2 System decomposition

Unless the system to be built is small enough to be produced by a single programmer, we should carefully divide the work into work assignments, which we call *modules.* We base the module decomposition on *information hiding* [Par72a]. The information hiding principle states that each module should encapsulate a "secret": a design decision that is likely to change. The whole purpose is separation of concerns, the most important principle in software engineering. If the information hiding principle is diligently applied, we may end up with many modules. Therefore, we need to produce a work product that documents the module structure, the *Module Guide.*

3.1.3 Module interface design

The module guide defines the basic responsibilities of each module, but it is not precise enough to allow programmers to work independently. For this, we need module interface specifications. According to Parnas [PC86], a *module interface* is the set of assumptions the rest of the program can make about the module's behavior. An interface should be opaque in that it hides the module secret, and need not change even if the secret does. A *module interface specification* is a description of these assumptions. The functions provided by the module for the rest of the program to

call are called *access programs.* A module interface specification has a syntax part and a semantics part. The *syntax* part states the names of the access programs, their parameter and return value types, and the names of the *exceptions* that each access program may generate, where *exceptions* are indications of illegal usage of the access programs. An exception is *signaled* when the access program is invoked in an unexpected way, such as incorrect parameters are supplied. Any constants and types provided by the module for its users are also described. The *semantics* part states, for each access program call, the situations in which the call is legal, and the effect that invoking the call has on the legality and return values of other calls. We will discuss an interface specification in detail in Chapter 5. We call a sequence of access programs a *trace.*

3.1.4 Module implementation

The module implementation phase produces executable code, in some programming language. The executable code for a module is a work product, called the *module implementation.*

3.1.5 Module testing

Modules are components of software systems. *Module testing* [Hof89] checks that the implementation of the module meets its specification. Module testing focuses on one module at a time, using stubs to isolate the module from other parts of the system, if necessary. Test cases are selected to best exercise the module, and drivers are written to execute these test cases.

3.1.6 Inspection and verification

We use *inspection* [Fag76] to improve software quality. Inspection is a formal, efficient and economical method of finding *faults*. A fault is different from *failure* in that a fault is an error in the source, which may be either design or implementation, while a failure is an error in behavior, observable at run-time. For the work product to be inspected M, the inspection process can be summarized as follows

1. overview: The author of M presents an overview to the whole inspection team to help other team members to understand M.
2. preparation: Each team member studies M individually preparing for the inspection meeting.
3. inspection: The whole team meets to find faults in M. The team goes through M line by line, only for finding faults. How to fix the faults is not discussed.
4. rework: The author of M resolves the faults found by inspection.
5. follow-up: The inspection team leader checks that all faults, problems, and concerns have been resolved. If many faults have been found in M, M should be re-inspected after rework.

Program verification proves that a program meets its specification, based on the source code, without actually executing the implementation by machine. Program verification is usually very expensive for a large program. *Exception checking* is a weak form of verification that ensures the program will run to completion without exceptions such as illegal array subscript and division by zero.

3.2 Improving an existing software system

3.2.1 Internal improvements

The term *software upgrades* [HH91] is used to denote the process of internally improving existing software structure while adding new features or removing errors according

to user requests. A *module upgrade* is the process of collecting a scattered set of related functions into a centralized module with well designed and defined interface. These functions are chosen in such a way that they together provide a single service, or share access to a single data structure (or device). With an explicit interface, the rest of the program can use the service provided without depending on the details of the data structure (or device) and the access methods employed. Hence, even if the data structure or access methods change, the rest of the program need not change. Only the work assignment that provides the service need change.

3.2.2 External improvements

Software enhancement denotes the process of modifying the software in a way that affects the behavior of the software system. Enhencements can be requested by market demands, user error reports, etc. When a given piece of code must be changed to implement an enhancement, the code must be (1) carefully studied, (2) modified, (3) tested, and (4) installed in production. If we apply upgrades at the same time we implement enhancements, we can save lot of work, since much of the work of upgrades has to be done for the enhancements anyway. Thus, upgrades scheduled with enhancements will often be significantly cheaper than upgrades applied standalone.

Chapter 4

The Module Upgrade Procedure

The module upgrade procedure consists of three phases: *preparation*, *module development* and *module incorporation*, with several work products produced in each step. Figure 4.1 shows a list of the work products to be produced in each phase. For each work product, we have developed criteria. Our goals have been work product criteria both precise enough to make faults identifiable, and concise enough to be used in practice. Meeting these two goals is challenging.

Each work product criteria has five sections. The *Audience* and *Preconditions* sections state the intended readers and the backgound knowledge assumed. These are part of the work product criteria because they restrict the terminology that can be used and the assumptions that can be made about the knowledge the readers possess. The *Purpose* and *Additional criteria* sections describe the fundamental purpose and specific constraints, if any, on the work product. The *References* section (optional) names the references needed to understand the work product.

We will discuss the procedure phase by phase, step by step. For each step, we will discuss the work products to be produced and their work product criteria. For

I. Preparation Phase
- Project Overview
- Module Selection
- Detailed Plans
  - Module Interface Development Plan
  - Module Implementation Plan
  - Module Test Plan
  - Module Incorporation Implementation Plan
  - Module Incorporation Test Plan

II. Module Development Phase
- Module Interface Specification
- Module Implementation
- Module Test Implementation

III. New Module Incorporation Phase
- Module Incorporation Implementation
- Module Incorporation Test Implementation

Figure 4.1: work products by phase

those work products that will be referenced frequently in the subsequent chapter, their complete criteria are included. A full set of work product criteria can be found in Appendix A.

4.1 The preparation phase

The preparation phase is a planning phase. We first discuss the criteria for choosing which module to upgrade, and then describe the detailed planning required.

Before we discuss module selection and detailed planning , it is important to clarify the concept of *module* in this context. We should keep in mind that the subject system might have been written with no concept of software engineering at all. However, we may still identify modules, even though their implementations may be scattered. We

call them *old modules*. These old modules can be upgraded into *new modules* that meet our standards. The new module performs essentially the same function as the corresponding old module.

4.1.1 Choosing the module

This step consists of choosing what part of the system is to be upgraded. A long list of module upgrades may be required to bring the system into "good shape." A system is in "good shape," if it meets a set of standards describing the ideal condition of the system, covering production source code, documentation, and testing code. This list of module upgrades will usually be long and expensive to carry out. Therefore, we need to upgrade one, or a few, module at a time.

Let M be the selected module. M is a good choice for upgrading if it satisfies the following criteria.

1. Upgrading M will substantially improve the maintainability of the subject system or provide a better base for an expected future extension of the system, and M can be upgraded with reasonable cost.

2. The modules used by M, if any, are in good shape. The bottom line is that after M is upgraded, it should be stable, and unaffected by upgrades to other modules. If module A uses module B, then it is preferable to upgrade module B before A, because if module B's interface changes, A will have to be changed as well.

3. M corresponds to a critical part of the system; hence it has a higher priority. The criticalness of a module can be measured in terms of its contribution to system safety, performance, and resource requirements.

4. If M is likely to change, should we upgrade it? Probably most people will say no. However, we would say yes. If M is not upgraded, future changes will cause frequent changes to other modules, because the interface of the old M was not well designed. On the other hand, if we upgrade the module and adhere to the module interface specification criteria in designing module interface, future changes will be hidden within the module. Hence, it will cause no change to other modules.
5. M corresponds to error-prone part of the system and it also should be given a higher priority.

The procedure is designed to allow project progress to be regularly reviewed through disciplined inspections. Therefore, the project team is required to produce a work product for each step.

For this step, the work product *Module Selection* should be produced. The major work product criteria of module selection can be stated as follows:

1. The module selection must follow the above module selection criteria.
2. The reasons of selection are documented.
3. The secret of the module is clearly documented.
4. The services offered by this module are briefly described.
5. The new or old modules used, if any, are listed.

4.1.2 Develop detailed plans

The detailed planning step consists of developing the *Project Overview*, dividing the project into smaller work units, called tasks, and developing a detailed plan for each

task.

The purpose of the Project Overview is to briefly describe the project to managers, upgrade team members and other software maintainers. It should name the tasks in the project, and for each task, give

- a unique task identification for easy reference throughout the project,
- a brief description of the task, and
- the estimated amount of time needed for each task.

To produce this work product, the author must understand the software module upgrade concept well, and have experience developing software in the given environment. Otherwise, it will be difficult to give a reasonablly accurate time estimate. In practice, the time estimate may be inaccurate until the detailed plans for each task in the project overview have been developed.

The detailed planning should be

1. detailed enough to support estimates for each task,
2. detailed enough to find errors, if any, which may cause the project to fail,
3. detailed enough so that the implementors can carry out tasks effectively, and
4. inexpensive enough that it is affordable to stop the project at this step.

The second point above is to find obstacles in the future, and the third is to find the "best" way of doing the work. Both of them aim at saving cost. It is also very important to ensure the cost effectiveness of the planning itself. If the planning itself becomes very expensive then we lose one of the fundamental purposes of planning:

stop the project early if we have to. On the other hand, in practice, the work may be so simple that it is not worth spending days in planning and writing. That is why we need some criteria to judge when to carry on and when to stop.

The factors to be considered during the planning step differ from project to project. However, experience shows that the key question to answer is "Will all the necessary resources, including human resources, be available to and easily accessible by the upgrade team?"

The qualifications of the upgrade team members, such as an understanding of information hiding, and the background to understand the system structure, is the first factor to consider. Besides the upgrade team, there is another human resource to consider: source of help information. The system being upgraded typically has little or no documentation. The upgrade team needs help from someone who knows the system well.

For the detailed planning step, the following work products should be produced: Project Overview and one Detailed Plan for each task in the project overview.

We have talked about the criteria of project overview. The detailed task plans have much in common in terms of work product criteria. For example, all task plans have the same purpose: to support detailed planning. Therefore, we list the major work product criteria for all detailed task plans in the following. Each task plan should contain the following sections:

purpose describe the purpose of the task briefly

implementor describe the knowledge, such as information hiding, and qualifications required to perform the task, and the list of persons who will perform the task

resources needed

- list hardware particularly needed, if any, for this task, such as an emulator or bus controller
- list the utilities needed for this task, if any, such as GREP, DIFF, PMAKE, YACC, and TCOV
- designate where should the task be performed, such as under Unix or MS-DOS, on a local drive or a networked remote drive
- list the documents that will be used
- list the files that will be used
- list the files that will be modified

strategy describe the major steps in carrying out the task

The work product criteria presented above is for all the task plans. It is sufficient for most of them. However, we can tailor the criteria to best suit one specific task. For example, the test plan plays an extremely important role in module testing. Experience has shown that it is very difficult to achieve adequate and economical testing without careful planning. If this planning is not recorded, it is hard for others to review, making the test code difficult to maintain. Note that the test plan is refined after the module interface specification is developed and the module is implemented.

Figure 4.2 shows the complete work product criteria for Module Test Plan. The critical module states and parameter values are best explained in terms of an example. Suppose we have a stack module, which implements a bounded integer stack. Its implementation state is

```
int siz;
int elements[MAX_SIZ];
```

For the module under test, M:

Audience
Those considering running or modifying M's testing

Preconditions
A thorough understanding of M's specification and some understanding of M's implementation

Purpose

- Serve as a planning tool for development and evaluation of the test input strategy
- Document the test implementation

Additional Criteria

- The test plan must be sufficient to provide an estimate of both the adequacy of the test inputs and the feasibility of the test implementation.
- The following sections are required:

  assumptions
  State any assumptions, not contained in M's interface specification, on which the testing depends

  test implementation strategy
  Describe the key aspects of the test implementation

  test case selection strategy
  Critical implementation states and parameter values are specified.
  Tests are planned to invoke every access routine in both normal and exceptional situations.
  100 percent statement coverage is expected.

- For each service module M, a simple interactive tester is provided.

Figure 4.2: work product criteria for module test plan

`siz` is the number of elements currently on the stack, `element` is an array storing the stack contents where only the first `siz` elements are valid and `MAX_SIZ` is the maximum stack size. The critical implementation states are:

- stack empty: `siz = 0`
- stack partially full: `siz` in `[1, MAX_SIZ-1]`
- stack full: `siz = MAX_SIZ`

For the partially full stack, it makes little difference whether the stack size is 5 or 50, as long as the stack is partially full. The values of the stack elements make no difference either, as long as they are integers. Therefore, it makes sense to select test cases where `siz` has values `0, MAX_SIZ/2, MAX_SIZ`. Typically the value of `MAX_SIZ` makes no difference, in terms of the effectiveness of testing. In another words, choosing `MAX_SIZ` to be 1000 than 10 will not provide significantly better testing. The tests should be planned to exercise each state as thoroughly as is practical.

4.2 The module development phase

In this section, we will discuss the steps needed to develop the selected module. First we will discuss the module interface specification development, and then the module implementation followed by module testing.

4.2.1 Develop module interface specification

An interface specification must be written for each module. It must be precise and provide a black box picture of each module. It is very important that the inter-

Audience
Module designer, implementor, tester and user

Preconditions
An understanding of the interface specification language

Purpose
Describe the assumptions users are permitted to make about module behavior, independent of the underlying implementation

Additional Criteria

- Describes the interface precisely and completely.
- Follows the interface specification language rules.
- Satisfies the interface design criteria, where practical.
- Is testable: no unjustifiable controllability or observability problems.

Figure 4.3: work product criteria for module interface specification

face specification provides enough information for the users of the module to use its facilities and no more.

Although probably only one or a few programmers will be involved in the implementation of the module, the interface specification is actually produced by a process of negotiation between the implementors, those who will be required to use it, and others interested in the design. The shape of the future system should always be kept in mind.

The work product criteria of module interface specification is shown in Figure 4.3.

4.2.2 Implement the module

We strongly emphasize that the correctness of a module should be based on careful design and review, using testing as a final check.

Our implementation criteria are based on three formal constructs: *abstraction function*, *implementation state invariant* and *loop invariant*. An abstraction function is a mapping from the implementation state, which is usually stored in the variables local to the module, to the specification state, which are objects manipulated by the user, as described in the interface specification.

An implementation state invariant is an assertion on the implementation state variables that must be true on entry to and exit from the code implementing each access program the module provides. For example, for the stack state

```
int siz;
int elements[MAX_SIZ];
```

its implementation state invariant is `siz in [0,MAX_SIZ]`. We choose this invariant because, as long as this relationship holds, `siz` and `elements` represent a stack. Note that the invariants are not necessarily true during the execution of a specific access program. For example, it is perfectly legitimate for the implementor to increase `siz` first and then store the value in `elements`. At the moment that `siz` is increased and the value yet not stored, the implementation state invariant does not hold, but this is acceptable. Also, note that the module state invariant is required to hold only after the implementation state is initialized.

A loop invariant is an assertion on selected variables that must be true before and after each execution of the loop body. Loop invariants play an important role in both loop verification and loop development. A loop usually consists of three parts:

initialization, loop body, and loop exit. We define a loop invariant before we code the loop. Then we establish the loop invariant in the loop initialization. When we code the loop body, we make sure it preserves the loop invariant. Finally, we make sure that the loop invariant still holds right after we exit the loop. Our experience has shown that it is very helpful to write loops this way.

The module correctness proofs should proceed in the following three steps.

1. Show that the implementation state invariant is established by the access program that initializes the implementation state, and is maintained by the other calls.

2. Argue the absence of certain specific faults, such as variable use before definition, array subscript range error, and pointer error.

3. Argue that the implementation meets its specification, one access program at a time, assuming that the implementation state invariant holds on entry to and exit from each call.

The work product to be produced for this step is the executable module implementation. Figure 4.4 shows the work product criteria for module implementations. The Code Format Rules codify traditional requirements for identifier scope and naming, code indenting and commenting. Programmers may have different personal preference on code format. However, it is very helpful that programmers in the same project team follow common guidelines. For example, we have a rule that requires identifier naming to be mnemonic and consistent. All the abbreviations used in the project are recorded in a file, `names`.

The Code Verification Rules contain a description of the abstraction function, implementation state invariant and proof scheme sketched above.

Audience
Module implementor, tester

Preconditions
Understanding of interface specification and implementation programming language

Purpose
Provide code whose behavior satisfies the specification

Additional Criteria

- Correctly implements the specification
- Satisfies the Code Format Rules
- Satisfies the Code Verification Rules
- Includes a set of default exception handlers, providing trivial exception reporting

Figure 4.4: work product criteria for module implementation

4.2.3 Module testing

Some people may ask: "If you have demonstrated that your module implementation meets its specification, why do you need testing?" It is true that testing is redundant provided that the design and implementation are ideal. Unfortunately, there is no such ideal implementation for complicated software. There may have been errors in our verification arguments. We use the redundancy of testing to improve reliability.

The test code should be part of production quality systems. Often tests are developed during or after the module implementation, and are thrown away after the module is considered to be stable, or acceptable. This will cause future maintenance problems. Software needs to be changed. Every time the module is changed, the tests need to be re-run. We cannot afford to develop test code again and again for the same module. It is true, of course, that the test code needs maintenance, just like

For the module under test, M:

Audience
Those considering running or modifying M's testing

Preconditions
A thorough understanding of M's specification and some understanding of M's implementation

Purpose
Implement the test plan as simply and inexpensively as possible

Additional Criteria
Implementation follows the test plan
Test runs to completion, with correct results and 100 percent statement coverage
Code obeys Code Format Rules

Figure 4.5: work product criteria for module test implementation

the module implementation.

It is often necessary to build some scaffolding to achieve adequate testing. We use available tools or build tools to automate the most tedious and repetitive tasks.

The work product to be produced is the module test implementation. Its work product criteria are shown in Figure 4.5.

Again, the work product is the executable test code. We emphasize that the implementation should follow the test plan. If the implementation cannot closely follow the test plan, this should be reflected in the test plan. In another words, the test plan should be maintained, because it serves not only a means of planning, but as documentation for the test implementation.

4.3 Incorporating the new module

As the result of the above phase, we have the selected module ready for use. Our task would have been finished if upgrading a module were like unplugging an integrated circuit chip, and plugging in a new one. Ideally, software should work that way as well. For a poorly designed system, unfortunately, it does not. We need to modify the existing system to incorporate the module into the system. It consists of two steps: module incorporation implementation and module incorporation testing.

4.3.1 Module incorporation implementation

The implementation step usually includes changing makefiles, if any, and the implementation code, such as `.c` or `.h` files. The makefiles must be changed to add the new module into the system and to do the necessary updating of the file dependencies. The users of the new module will normally depend on the new module.

Changing the implementation code should be carried out very carefully. First we must establish the work base by backing up the whole system. Also, we need to make sure that we know exactly how the system works in the upgrade environment before we change any code, because the upgrade environment may be different from the production environment. If we find something does not work or works in a strange way, we want to make sure it is caused by our changes, rather than spend days trying to figure out where we made a mistake and finally find that it is an unexpected "feature" of the existing system. Remember that there are often some features that are documented no where but in the software itself. Probably nobody knows all the "features" of the system.

If there are many files to be changed, changes to the system should be made

incrementally. We should complete and test changes to one or a few files, before we make changes to 20 or more files. It is possible that we will find that the new module interface needs slight changes after we have made changes to a few files. If this happens, incremental changing can save us some time.

The major work product criteria for module incorporation implementation is:

- Code meets the Code Format Rules and the Code Verification Rules.
- No unnecessary changes were made to the existing code.
- All changes made are recorded, and inspected.

We emphasize that all the changes made should be inspected. The author should be able to explain why the change made is necessary and correct. Typically, the way to explain is to establish the equivalence of previous code and the new code. Two pieces of code are said to be equivalent if both of them can be abstracted to the same design function.

4.3.2 Test module incorporation

Although we emphasize that design is more important than testing, we cannot ignore the importance of systematic testing. While it is a good idea to run a complete set of system tests after we made some, even small, changes to the system, it is often too expensive to do so in practice. For example, a complete set of system tests for a database management system may take several days to complete. Therefore, we often need to test the module incorporation selectively.

The work product to be produced for this step is: module incorporation test implementation. Its work product criteria are summarized below:

- Implement the module incorporation test plan as simply and inexpensively as possible.
- Implementation follows module incorporation test plan.
- Test runs to completion, with correct results and 100 percent statement coverage, if possible.
- Code obeys Code Format Rules.

Suppose M is the set of statements we added or modified. By "100 percent statement coverage," we mean that every statement in M is executed at least once. Commercial utilities for measuring statement coverage, such as TCOV on Unix, will not help in this situation. They can only measure the statement coverage of the whole program, but we are expecting the statement coverage of what we changed, not the whole program. However, we can divide M into a set of blocks, where either all or none of the statements in the same block will be executed. Then, we can insert some special observable action into every block we added or modified, such as printing out a special message, to indicate that the block was executed. We can tell if all blocks are executed by examining all the messages. In practice, it might be very difficult to achieve 100 percent statement coverage in one application. In that case, we should design several applications to exercise the system so that the 100 percent statement coverage is still achieved.

Chapter 5

The DCBL Project

In the previous chapter, we defined the module upgrade procedure (hereafter just *the procedure*). We wanted to know if it is practical, and whether the work product criteria are both precise enough to make faults identifiable and concise enough to be used in inspection meetings. Also, we wanted to know how to improve the procedure. Therefore, we started the TICS DCBL module upgrade project (hereafter just *the DCBL project*).

DCBL is a device control block list module that provides sequential and direct access to the list elements. The old DCBL used in-line code wherever list access was required. This caused maintenance difficulties.

Therefore, we started the DCBL project, with its purposes clearly stated. For our research, the purposes were

1. to evaluate and improve the procedure,
2. to present an example of applying the procedure, and
3. to determine the cost of each step, to support cost estimating.

For TICS users and programmers, the purpose is to reduce pointer errors and improve TICS maintainability.

5.1 Introduction to device handling in TICS

Each hardware device controlled by TICS has a Device Control Block (DCB) that records all data related to the device, such as device status (on/off) and device value (e.g., voltage). Each device is controlled through its DCB via a set of *general* and *device specific* functions. The DCBs are linked together, forming a single circularly linked list. The `next` field of the last DCB points back to the list head, i.e., the first DCB. The DCB list is accessed both sequentially and directly. The TICS CP operates on the DCB list.

5.2 The preparation phase

5.2.1 Why select DCBL

We selected the DCBL code for upgrading because

- The DCBL code is distributed across more than 30 files. As a result, it is hard to understand and maintain.
- DCBL does not heavily depend on other modules.
- The DCBL is critical to system safety.
- DCBL is of moderate complexity. We did not want to select one that is trivial or too complicated for the purpose of evaluating and improving the procedure.

- The author of TICS, Dr. Rolf Keitel, thought DCBL was a good choice.

5.2.2 Detailed planning

The first factor to consider in the planning step is people. We were confident about the qualifications of the upgrade team members: a senior software engineering expert and a well-trained software designer who knows the C programming language well. We have software inspection experience and a thorough understanding of software engineering principles. Also very important, we have a source of help: one TICS software maintainer, though he is far away from us.

Setting up the appropriate environment for TICS to run took a long time. We began by visiting TRIUMF, since TICS was runing there. Then we bought a PC and copied the TICS source code. However, since TICS was maintained in a networked environment quite different from ours, it was not an easy job to get it compiled correctly on our machine. Makefiles had to be changed. Because we have no hardware simulators, we needed to emulate device status and values using main memory. For example, when the user issues a command to turn off a pump, intsead of turning a real pump off, a particular bit in memory is set to 0. When the user turns the pump on, the bit is set to 1.

A complete set of work products for this phase can be found in Appendix B. We will discuss only the module incorporation plan.

Figure 5.1 shows portions of the Module Incorporation Plan. Item b under step 3 was designed to help others to review the changes made. Flagging every change with `dl_s_` or `dl_g_` helps to locate a error, since we do not have to browse through the whole file. We chose `dl_s_` or `dl_g_` as the flag because the name of every access program of DCBL contains that string. Therefore, we need to do almost nothing to

1. Establish work base
   a. Ensure that the current version compiles and runs
   b. Make a list of compilation and linking warnings, if any

2. Backup the working version
   a. Backup the directory `c:/tics` to floppy disk, store off-site
   b. Backup `c:/tics/v3x/*.*` to `d:`

3. Establish criteria for changing the source code:
   a. Every file changed should include `dcbl.h`
   b. All changes made should be marked by `dl_s_` or `dl_g_`

4. Change makefiles

5. Replace in-line traversal code with calls to DCBL (see subtask plan 1)

6. Inspect the changes

Note:
   a. TICS CP `.obj` files listed in `tics.lis`.
   b. C files to be changed listed in `chg.lis`.

Subtask plan 1

1. Copy `dcbl.c, dcbl.h, dcbl_e.c` into `c:/tics/v3x`

2. Add `dl_s_init` into ticsmain.c

3. Replace the in-line code for traversing the DCB list in `tischedu.c` by function calls to DCBL module

4. Compile, test, and debug

5. Replace the in-line code in rest of files by function calls

Figure 5.1: Module Incorporation Plan

keep track of changes made, since calls to DCBL access programs of DCBL would appear in almost every line we would change.

5.3 Developing the DCBL module

The purpose of the DCBL module is to provide services for sequential and direct access to the elements in the DCB list. The direct access service is provided in two forms: by device name and by device number. The module must ensure that every pointer it returns is a pointer to a valid DCB. The secret of the DCBL module is the list data structure and the access method employed.

5.3.1 The DCBL module interface specification

Figures 5.2 and 5.3 show the syntax and semantics of the DCBL module interface specification. Every access program has the prefix `dl_`. We will discuss the service each access program provides and why the access program is needed.

If a module has a state, i.e., it remembers something between the execution of two access programs, it typically needs a specification state. This state needs initialization. `dl_s_init` is designed to initialize the specification state.

Sequential access to list elements is achieved by the following calls: `dl_s_start`, `dl_g_cur`, `dl_s_next`, and `dl_g_end`. We can view the sequential traversal as moving a cursor through the list. The element at the cursor position, *the current element*, is accessible to the module user. `dl_s_start` moves the cursor to the head of the list. `dl_g_cur` returns a pointer to the current list element, and `dl_s_next` moves the cursor to the next element in the list. `dl_g_end` returns a non-zero integer, if the cursor has been moved beyond the last element in the list.

```
/*****types*****/
typedef struct dl_item {
        char id;                /* control block id byte */
        int number;             /* device number          */
        int status;             /* 16 bit status flag word */
        struct dl_item *next;   /* link for element manager */
        char *firstcmd;         /* head of command list    */
        char name[11];          /* element name            */
        /* ... other data */
} dl_item;

/*****access programs*****/
void dl_s_init();
/*      void dl_s_init(list_header)
*       dl_item *list_header;
*/

void dl_s_start();

void dl_s_next();

dl_item *dl_g_cur();

/*boolean*/ int dl_g_end();

dl_item *dl_g_bynam();
/*      dl_item dl_g_bynam(dev_name)
*       char *dev_name;
*/

dl_item *dl_g_bynum();
/*      dl_item dl_g_bynam(dev_number)
*       int dev_number;
*/

/*exceptions*/
void dl_end();

void dl_legpos();
```

Figure 5.2: DCBL module interface specification – syntax

Assumptions

1. dl_s_init must be called before any other call to this module.
2. lp, the parameter to dl_s_init, must be either NULL or a pointer to a circular list of dl_item's.

Specification state

dcblst: sequence [0..] of DCBTYP; pos: integer;

Set call effects

dl_s_init(lp):
  dcblst := list associated with lp
  pos := 0
dl_s_start(): pos := 0
dl_s_next(): pos := pos+1

Get call return values

dl_g_cur() = dcblst[pos]
dl_g_end() = not pos in [0,length(dcblst)-1]
dl_g_bynam(nam) = an item with its name field being nam,
  or NULL if no one matches.
dl_g_bynum(num) = an item with its number field being num,
  or NULL if no one matches.

Exceptions

dl_s_next():
  (dl_end: pos = length(dcblst))
dl_g_cur():
  (dl_legpos: not pos in [0,length(dcblst)-1])

Local types

DCBTYP ::= tuple of {
  number: integer,
  name: string,
  data: DATA
}
DATA is a type that stores all the information related to
the DCB. We eliminate its details here.

Figure 5.3: DCBL module interface specification – semantics

The following code sequentially traverses a list, printing the device name of each element in the list.

```
dl_item *cursor;
dl_s_start();
while (!dl_g_end()) {
    cursor = dl_g_cur();
    printf("device name: %s\n", cursor->name);
    dl_s_next();
}
```

A particular element can be directly accessed by its device name, via `dl_g_bynam`, or by its device number, via `dl_g_bynum`. If a DCB with that name or number is found in the list, a pointer to the DCB is returned; otherwise, NULL is returned.

Two exceptions may be raised in DCBL. `dl_end` will be signaled if `dl_s_next` is called while the cursor is already beyond the last element of the list. Similarly, `dl_legpos` will be signaled if `dl_g_cur` is called while the current element is not valid, i.e., `dl_g_end` is true.

It is interesting that accessing the list elements is just like accessing the records in a file. In file access, we have an open operation that designates a specific file to be accessed. Our DCBL has a `dl_s_init` to allow user to specify which list to operate on. `dl_s_start` corresponds to the reset operation, and `dl_g_cur` plus `dl_s_next` corresponds to reading a record and moving to next record. Finally, `dl_g_end` corresponds to an end-of-file test. However, unlike file access operation, direct access to the list elements does not affect sequential traversal over the list. This is important because, in TICS, sequential and direct accesses are often interleaved.

5.3.2 Implementing the DCBL

The implementation of DCBL is relatively simple compared to other tasks in the project. We will focus our discussion on verifying that the code meets its specification.

The abstraction function is a mapping from the implementation state to the specification state. The specification state (see Figure 5.3) consists of two variables: `pos`, the position of the current DCB, and `dcblst`, the list of DCBs. The implementation state is as follows:

```
typedef struct dl_item {
    int number;
    char name[11];   /* null-ended */
    dl_item *next;
    DATA data;       /* other data */
} dl_item;
dl_item *cur;
dl_item *head;
```

where `cur` points to the current DCB, and `head` points to the first DCB in the list. The actual `dl_item` structure is more complicated, but it is irrelevant to our discussion, so we eliminate its details here. The abstraction function and implementation state invariant are shown in Figure 5.4. The implementation state invariant says that the list head is either NULL if the list is empty, or a pointer to a forward linked list of legal DCBs with the `next` field of the last DCB pointing back to the head. `cur` is the address of the current DCB, for sequential traversal purposes. It is NULL if the list is empty. Note that `cur` is a pointer variable, but `pos` is a integer variable.

An example can best show how the abstraction function works. Assume that the implementation state is

```
/*
*definition
*    define EL(h,n)
*        address of the nth element in the DCB list headed by h,
*        where h is the address of element 0.
*
*    a memory block is a "legal DCB", if
*        1. it is of size at least sizeof(dl_item)
*        2. let q be the address of the block, then
*           (dl_item *)q->name contains a '\0', and
*           (dl_item *)q->next points to another legal DCB.
*
*    a legal DCB list is a circularly linked list of legal DCBs.
*
*implementation state invariant
*    1. head is either NULL or the address of a DCB in a legal
*       DCB list
*    2. cur is either NULL or an address of a legal DCB
*
*abstraction function
*       pos = i, where EL(head,i) = cur
*       dcblst = <> iff head = NULL
*       (forall i)(if i in [0,len-1] then
*           dcblst[i].number = EL(head,i).number, and
*           dcblst[i].name = EL(head,i).name, and
*           dcblst[i].data = EL(head,i).data)
*       where len is the number of different DCBs in the DCB list
*       pointed to by head
*/
```

Figure 5.4: The abstraction function and implementation state invariant for DCBL

1. the list consisting of three DCBs with the `number`, `name`, `data`, and `next` fields

```
a1: <25, "pump", data1, a2>
a2: <15, "valve", data2, a3>
a3: <18, "psupply", data3, a1>
```

   and `a1`, `a2`, and `a3` the addresses of the three DCBs.

2. `head = a1` and `cur = a2`

Now, what is the corresponding specification state? By definition, `EL(head,0) = a1`, `EL(head,1) = a2`, and `EL(head,2) = a3`. Since `EL(head,1) = cur`, according to the first sentence in the abstraction function, we know `pos = 1`. Since the list consists of 3 DCBs, we know `len = 3`. Therefore, `dcblst` is the sequence

```
<<1,"pump",data1>, <2,"valve",data2>, <3,"psupply",data3>>.
```

The following is an example of an exception check and an argument for meeting the specification. We will argue that the loop inside `dl_g_bynum` will terminate (exception check) and that it returns either NULL if no match, or the correct DCB pointer. The implementation is shown in Figure 5.5. The loop invariant is: `p` points to a legal DCB in a legal DCB list. As defined in Figure 5.4, a legal DCB list is a circularly linked legal DCBs. At line 6, `p` is initialized to `head`. By line 5, we know that `head` is not NULL. By note 2 in the semantics part of the interface specification, we know that `dl_s_init` will establish the implementation state invariant. According to the implementation state invariant, `head` points to a circularly linked list of legal DCBs. Thus, the loop invariant is established. Now suppose the first DCB does not match, then `p` is assigned `p->next`. Since `p` is a legal DCB in a *circularly* linked list, `p->next` keeps the loop invariant true. Again, because of the circular feature, `p` will traverse all the DCBs in the list and point back to `head`, if the loop does not exit from `loop exit1`. Therefore, `p != head` will be false. The loop terminates. Obviously, if the loop exits from `loop exit1`, the loop terminates. Note that this is an informal argument.

```
1  dl_item *dl_g_bynum(num)
2  int num;
3  {
4          dl_item *p;
5          if (head == NULL) return(NULL);
6          p = head;                          /*loop initialization*/
           /*loop invariant:
                  p points to a legal DCB in a legal DCB list*/
7          do {                               /*loop body*/
8                  if (p->number == num)
9                          return(p);         /*loop exit1*/
10                 p = (dl_item *)p->next;
11         } while (p != head);               /*loop exit2*/
12         return(NULL);
13 }
```

Figure 5.5: the implementaion of `dl_g_bynum`

A more formal argument can be made by means of a monotonic function with a bound.

Having proved that the loop invariant holds and that the loop terminates, it is easy to argue that `dl_g_bynum` returns either NULL, at line 5 or 12, or a legal DCB pointer, at line 9. If `dl_g_bynum` returns from line 9, we know `p->number` matches the given device number, thus meeting the specification. If it returns from line 5, i.e., `head` is NULL, we know by the implementation state invariant that the DCB list is empty, so a NULL return value meets specification. If it returns from line 12, we know no DCB in the list matches the given device number, since `p` has traversed all the DCBs. Therefore, `dl_g_bynum` meets its specification.

5.3.3 Module testing for DCBL

Our test implementation strategy is:

1. Develop the testing code on Unix, then port to PC-DOS.
2. Use the PGMGEN utility to automate the most tedious part of testing.
3. Use TCOV to measure statement coverage (Unix only).
4. Use C macros to improve readability.
5. Use the C function `gendl(n)` to create a DCB list of length n, with the number field of ith DCB is `NUM(i)` and the name field of the ith DCB is `NAM(i)`, where `NUM(i)` and `NAM(i)` are C macros.

The PGMGEN utility is an automated module testing tool. It reads a test script file, describing the test cases and expected outputs or exceptions, and generates an executable file. This executable file runs all the test cases, compares the actual output with the specified value, and reports any mismatches detected. TCOV is a utility designed for test coverage analysis. It records the number of times each statement in the source code has been executed. It is only available on Unix. Two C macros are defined: `NUM(i) = (i+1)*10` and `NAM(i) = (itos(NUM(i))`. `NAM(i)` is a string version of `NUM(i)`, just to improve readability. `gendl(n)` is needed because DCBL operates on an initialized DCB list, so we need to create a DCB list to test DCBL.

The test case selection strategy is shown in Figure 5.6. The exception testing is straightforward. We divide the normal case testing into two steps: (1) check that every access program returns correct values and (2) check that sequential traversal is unaffected by interleaved direct access. We created DCB lists of lengths varying from 0 to 10. For each list, we tested whether `dl_s_init` set default values correctly, and whether sequential traversal access and direct access behave correctly and independently. In the second step our goal was to check that access to the data fields, such as device name and device status, will not affect the correctness of DCBL.

In order to run the test code on DOS, we had to make slight changes to overcome the memory address differences between Unix and DOS. On Unix, all memory

```
critical implementation state values
   list length: [0,N]
   list elements: ith element has name and number fields
      generated by gendl.
   sequential traversal status:
      before s_start
      after s_start: number of s_next range in [0,length(list)]
exceptions
   empty list, issue g_cur, s_next
   non-empty list, move the cursor to the end of the list,
      issue g_cur, s_next
normal
   for each dlsiz in [0,N-1] /*list size*/
      call gendl(dlsiz)
      check g_cur if dlsiz > 0
      check g_bynam, g_bynum, g_end
      issue s_start
      for each ns in [0,dlsiz-1] /*number of s_next*/
         /*check direct access,including NULL return*/
         for each i in [0,dlsiz]
            issue g_bynum(NUM(i))
            issue g_bynam(NAM(i))
         check g_cur
            check current dcb's number field = NUM(ns)
            check current dcb's name field = NAM(ns)
         check g_end
         check s_next
   /*test if field changes will affect list traversal*/
   call gendl(N)
   for each i in [0,N-1]
      /*for the ith dcb in the list*/
      set its number field = NUM(i+1)
      set its name field = NAM(i+1)
      issue s_next
   for each i in [0,N]
      check g_bynum(NUM(i+1))
      check g_bynam(NAM(i+1))
```

Figure 5.6: test case seletection strategy for DCBL

addresses are 4 byte long. On DOS, memory addresses are represented in terms of *segment* and *offset*. Therefore different address values may correspond to the same memory location.

We executed 57 test cases (N = 10) on Unix and on PC, using less than 1 second CPU time. PGMGEN did report a error that we missed in the inspection meeting.

5.4 DCBL module incorporation

We followed the following four steps:

1. Identify the design function: the function a piece of code computes.
2. Re-implement the design function using DCBL.
3. Perform exception analysis for the calls on DCBL.
4. Show that the new and original implementation are equivalent.

In step 1 we located every piece of code that was part of the old DCBL, i.e., the code that implemented the sequential traversal and direct access services, and every piece of code that directly used these services. For example, the following piece of code is taken from `tischedu.c` in the original implementation.

```
ini_devices()
{
        extern SYSCB *tics;
        PPCB *dev;
        dev = tics->hd_device;
        do {
                ini_dev_functions(dev);
```

```
                (*(dev->set_status_function))(dev);
                if (dev_status(dev->id,dev->status) == ON)
                        dev->status |= EST_SHOULDBEON;
                else dev->status &= ~EST_SHOULDBEON;
                dev_ini(dev);
        } while ((dev = (PPCB *) dev->next) != tics->hd_device);
        return(0);
}
```

Its purpose is to sequentially traverse all the elements in the DCB list, set the expected element status flag according to the actual device status, and call device-specific initialization routines.

In step 2 we re-implemented `ini_devices` replacing the in-line list traversal code with calls to DCBL. The result is as follows:

```
ini_devices()
{
        extern SYSCB *tics;
        PPCB *dev;
        dl_s_init(tics->hd_device);        /*new*/
        while (!dl_g_end()) {              /*new*/
                dev = dl_g_cur();          /*new*/
                ini_dev_functions(dev);
                (*(dev->set_status_function))(dev);
                if (dev_status(dev->id,dev->status) == ON)
                        dev->status |= EST_SHOULDBEON;
                else dev->status &= ~EST_SHOULDBEON;
```

```
            dev_ini(dev);
            dl_s_next();                /*new*/
        }
        return(0);
}
```

We can see that no dependency on the DCB list structure appear in the new code. If, for example, the list structure were changed from a circularly linked list to an array, this piece of code will not be affected.

In step 3, we checked that the use of the DCBL access programs were legal: they do not violate the assumptions stated in the semantics part of DCBL interface.

In step 4, we argued that the new code was compatible with the old. This was done by means of the DCBL interface specification, without reference to the implementation of DCBL. We found that the new code could handle the special case of an empty DCB list, while the old code could not.

The module incorporation testing was done through a tailored application program called "dtrans". We used `printf` to check that each line of the modified code was executed.

Chapter 6

Experiences and discussion

6.1 Time cost analysis

The DCBL project was started in the Summer of 1990 and was completed in the Spring of 1991. Recording the cost of each step was one of the goals of the DCBL project. By recording the cost of one project, we hope to provide better support for the estimation of other similar projects. Also, we can analyze which components of the cost can be saved by scheduling upgrades together with enhancements that have to be done anyway.

6.1.1 The time cost of DCBL project

Figure 6.1 shows the detailed time cost for each phase and step of the DCBL project. The cost of some steps are divided into sub-units. For example, in the preparation phase, the module selection step was further divided into studying the TICS system, and defining the problem to be solved. We did so to distinguish the time directly

	Time (in person-hours)
Phase I: Preparation	460
• Module selection	212
1. Study TICS system	176
2. Define the problem to be solved	36
• Detailed plan development	248
1. Project overview	36
2. Module interface development plan	34
3. Module implementation plan	36
4. Module test plan	58
5. Module incorporation implementation plan	40
6. Module incorporation test implementation plan	44
Phase II: Module development	116
• DCBL module interface specification	44
• DCBL module implementation	28
• DCBL module testing	44
Phase III: Module incorporation	160
• Module incorporation implementation	112
1. TICS configuration changes	12
2. TICS CP change experiment	28
3. TICS CP changes	72
• Module incorporation test implementation	48
Total	736

Figure 6.1: Time cost for the DCBL project

related to the specific DCBL module and the time needed for a similar enhancement so that we can analyze the time we can save by scheduling upgrades together with enhancements.

The preparation phase took us 460 hours, a lot of time. This was caused by factors that were not fully under our control, such as difficulty in communication with TRIUMF. Studying the TICS code took us a long time. If the TICS system was decomposed into well defined modules, it would have been much easier to grasp the system structure. Many global variables were used making it very difficult to find out how they are maintained.

Developing the detailed task plans was the major task of the planning phase. It was the most time consuming step, not because it was the most difficult one, but because it covers the whole project life cycle. It included six work products to be produced. Also, the detailed planning step required further studying of the TICS code.

The module development phase was completed smoothly. This is probably because we have developed many modules similar to DCBL. We found that the development environment affects the speed of development a lot. Easy access to the standards, and similar examples for implementation state invariants and abstraction function can save lot of time. Utilities like TCOV and PGMGEN played an important role in module testing.

The module incorporation phase would have taken us more time, if the TICS software maintainer had not helped us to develop the system testing applications.

From the figure we can see that the most time we spent was setting up the environment, learning the TICS code, and planning how to do the module upgrade.

6.1.2 Scheduling upgrades with enhancements

Now let us use an example to analyze the savings achievable by scheduling upgrades with enhancements. Assume that the author of TICS has decided to enhance the performance of TICS CP. The DCB list structure will be changed from a linked list to a sorted array so that binary search can be used to achieve fast access to DCBs in the list. Remember that, before the DCBL project, TICS used in-line code for list access. The *enhancement project* includes the following steps (hereafter called *enhancement steps*).

1. Carefully study the TICS code,
2. produce an overview of the enhancement project,
3. find all the pieces of code that access the list,
4. produce a plan for testing the result of the enhancement,
5. replace all of the list access code with new code that employs a sorted array and binary search, and
6. build applications to test the enhancement result.

Now let us go through the recorded time costs of DCBL project shown in Figure 6.1 to see which costs can be saved by doing the upgrade along with the enhancement.

For the module selection step, both the enhancement project and upgrade project require careful study of the TICS code. Scheduling the upgrade with the enhancement will incur no extra cost. Therefore, part 1 of the module selection step (with actual cost of 176 person-hours) can be saved. Part 2 of the module selection step, defining the problem to be solved, is irrelevant to the enhancement project. Therefore this cost, 36 person-hours cannot be saved.

For the detailed planning step, producing the project overview (36 person-hours) is almost the same as the enhancement step 2 above. The enhancement steps 3 and

4 do almost exactly what is needed to produce the module incorporation implementation plan (40 person-hours) and module incorporation test plan (44 person-hours). Upgrades requires no extra cost. Therefore, we save 120 person-hours (36+40+44). The cost of producing the rest of the work products cannot be saved.

For the steps in the module development phase, no time can be saved, because the enhancement project does not produce a new, separate module.

For the module incorporation implementation step, enhancement step 5 does almost the same work except that no configuration changes are needed (since no new module is added). Therefore we can save 100 (72+28) person-hours.

For the module incorporation test implementation step, enhancement step 6 does almost the same work. Therefore we can save 48 person-hours.

In summary, we can save 444 (176+120+100+48) person-hours by scheduling upgrades with enhancements for this example. In another words, almost 60 percent (444/736) of the work needed for the DCBL upgrade is needed for a similar enhancement project. We can save 60 percent of effort by doing the upgrade at the same time of an enhancement.

6.2 Maintaining the work products

In the module upgrade procedure, we emphasize systematic planning. Each step begins only after the previous steps have been completed. The work products produced in each step will contain enough information for the following steps and are not changed once the next step begins. This is an ideal way of upgrading. In practice, the project is often carried out in an iterative fashion. We may find work products from previous steps need to be modified. Nonetheless, we maintain the work products as

if they were correctly produced the first time.

For example, in the first version of DCBL interface and implementation, direct access was achieved by means of sequential traversal and key-comparison. Therefore, any direct access will make the sequential traversal state void. However, when we replaced the in-line traversal code with function calls to the DCBL module, we found that sequential traversal and direct access were often interleaved. Therefore, we had to go back to the module development phase and modify the interface and implementation so that direct access and sequential traversal are independent.

Chapter 7

Summary and Future Work

7.1 Summary of our work

Although substantial research work has been done in the area of software re-engineering, little help is available in improving complex industrial software systems. Commercial tools are available to extract program control or data flows, measure complexity, or perform program restructuring. However, they can only provide services for syntactic analysis. Human understanding and decision making are still necessary elements of the re-engineering process. Human resources are the major part of cost of a re-engineering project. Therefore, it is important to pursue methods to reduce the man-power cost.

We proposed the following approach: divide re-engineering projects into categories, and, for each category, establish a standard procedure to control the re-engineering process. This procedure can be applied to other projects in the same category, thus reducing errors and improving productivity. By applying this procedure repeatedly, we can improve the system quality incrementally.

We have studied one specific category of projects, software module upgrades. The systems in this category have poor module structure. The implementation of a module may be scattered in many files. We established a procedure for incrementally improving the quality of this kind of software system. The procedure specifies a process of collecting a set of related functions into a module with a well-defined interface. Incorporated into this procedure are software engineering principles such as information hiding, verification, and testing. The procedure was designed to make the upgrade process reviewable at each step. Our goal has been to define the procedure precisely and to allow the system to be upgraded incrementally. Applying this procedure to a portion of a software system takes time and effort, but we may significantly reduce the future maintenance cost for that portion.

We defined a set of work products to be produced at each module upgrade step. The guidelines for producing these work products and the constraints on them are discussed. The module upgrade procedure consists of three phases: preparation, module development, and module incorporation. The preparation phase identifies which portion of the system is to be upgraded, checks that the upgrade will be cost-effective, and produces a detailed plan for doing the upgrade. The module development phase defines the new module interface, and implements and tests the new module. Typically, services provided by the new module existed in the original system, but were scattered in many places. The last phase is to replace the scattered code with calls to the new module.

Our TICS DCBL module upgrade project presents an example of following the module upgrade procedure. We upgraded the device control block list management of the TICS system.

Through the DCBL project, we learned that software module upgrade can be done at reasonable cost. The maintainability will be steadily improved after more

and more portions of system are upgraded. The software module upgrade procedure helped us to carry out the DCBL upgrade project smoothly and successfully. This indicated that the module upgrade procedure is beneficial in practice.

The detailed time costs of the DCBL upgrade project will help us to make accurate cost estimation for other module upgrade projects.

7.2 Future work

So far, we have no precise measurement on the improvement of the maintainability. Our argument that the maintainability of TICS software system has been improved is based on human judgement. We would like software metrics that measure the maintainability of a software system. While it is difficult to develop a metric that accurately reflects software maintainability, it would be better to have some metric than nothing.

Applying the module upgrade procedure to more upgrade projects will improve the procedure. Also, more time records can help more accurate cost estimate and schedule control. We would also like to extend the module upgrade procedure to a broader domain.

Control of the concurrency problem in an upgrading process is an issue that cannot be neglected when the upgrade team consists of many people. The concurrency problem occurs when the upgrade needs a multi-person team, and, as a result, many files may be modified concurrently. Two team members may want to upgrade the same portion of the system in conflicting ways. One member may finally announce that his/her portion of upgrade cannot be done within a reasonable cost. Also, one member may leave in the middle of a project. Our proposal to address the above

problems is to borrow the two-phase-commitment technology from the distributed processing area, and incorporate it into our module upgrade procedure. Thus, the module upgrade procedure can be extended to handle these problems.

Bibliography

[Arn86] Robert S. Arnold. An introduction to software restructuring. In *Tutorial on software restructuring*, pages 1–10. IEEE Computer Society, 1986.

[Bak77] B. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[Bas80] V. Basili. *Tutorial on Models and Metrics for Software Management and Engineering.* IEEE computer society, 1980.

[Big89] T.J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, pages 36–49, July 1989.

[Bus85] E. Bush. The automatic restructuring of COBOL. In *Proceedings of the Conference on Software Maintenance–1985*, pages 35–41. IEEE Computer Society, 1985.

[Can84] R. Canning. Rejuvenate your old systems. *EDP analyzer*, 22(3):1–16, March 1984.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–18, January 1990.

[Fag76] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, July 1976.

[Fre83] Peter Freeman. Reusable software engineering: concepts and research directions. In *Proceedings of the workshop on reusability in programming*, 1983.

[HH91] Marinos Hadjistylianou and Daniel Hoffman. Upgrades for engineering software. Submited for publishing, January 1991.

[HMKD82] W. Harrison, K. Magel, R. Kluczny, and A. Dekock. Applying software complexity metrics to software maintenance. *IEEE computer*, 15(9):65–79, September 1982.

[Hof89] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance*, pages 100–105. IEEE Computer Society, October 1989.

[Hof90a] Daniel Hoffman. *Work Product Criteria.* Computer Science Dept., University of Victoria, Summer 1990. Course note for CSC 365.

[Hof90b] D.M. Hoffman. On criteria for module interfaces. *IEEE Trans. Soft. Eng.*, 16(5):537–542, May 1990.

[Hou83] C. Houtz. Software improvement program (sip): A treatment for software senility. In *Proceedings of the 19th computer performance evaluation users group*, pages 92–107, 1983.

[KP76] B.W. Kernighan and P.J. Plauger. *Software tools.* Addison-Wesley, 1976.

[Laf90] Gilles M. E. Lafue. Panel on software re-engineering. In *12th International Conference on Software Engineering*, page 118, March 1990.

[LG84] R. G. Lanergan and C.A. Grasso. Software engineering with reusable designs and code. *IEEE transaction on software engineering*, SE-10(5):498–501, November 1984.

[Lyo81] M. J. Lyons. Salvaging your software asset (tools based maintenance). In *Proceedings of the National Computer Conference 1981*, pages 337–341. AFIPS Press, 1981.

[M.84] Sneed H. M. Software renewal: a case study. *IEEE software*, 1(3):56–63, July 1984.

[Mor84] H. W. Morgan. Evolution of a software maintenance tool. In *Proceedings of the 2nd National Conference on EDP Software Maintenance*, pages 268–278. U.S. Professional Development Institute, 1984.

[Par72a] D.L. Parnas. Information distribution aspects of design methodology. In *Proc. IFIP Congress 1971*, pages 339–344. North-Holland Publishing Company, 1972.

[Par72b] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[Par84] G. Parikh. Logic retrofit may save millions of dollars in software maintenance. In *Proceedings of 2nd national conference on EDP software maintenance*, pages 427–429, 1984.

[PC86] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Trans. Soft. Eng.*, SE-12(2):251–257, February 1986.

[PS83] H. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[War78] J.D. Warnier. *Program Modification.* Martinus Nijhoff, 1978.

Appendix A

The Work Product Criteria

This appendix consists of the work product criteria for the module upgrade procedure. Our goals have been to make these work product criteria both precise enough to make faults identifiable and concise enough to be practically usable in an inspection meeting. Meeting these two goals is challeging.

For each work product P, we have developed its criteria, a specification of the constraints on P. In all cases, the constraints of simplicity and clarity are assumed.

Each work product criteria has five sections. The *Audience* and *Preconditions* sections state the intended readers and the backgound knowledge assumed. The *Purpose* and *Additional criteria* sections describe the fundamental purpose and specific constraints, if any, on the work product. The *References* section (optional) names the references needed to understand the work product.

Reference for all work products: [PC86].

A.1 Summary of steps and work products

Summary of Steps and Corresponding Work Products

A. Preparation phase
   1. Module Selection
      Module Selection (MS)
   2. Detailed Plan Development:

Project Overview (PO)
Module Interface Development Plan (MIDP)
Module Implementation Plan (MIP)
Module Test Plan (MTP)
Module Incorporation Implementation Plan (MIIP)
Module Incorporation Test Plan (MITP)

B. Module Development phase
1. Module Interface Definition
Module Interface Spec (MIS)
2. Module Implementation
Module Implementation (MI)
3. Module Testing
Module Test Implementation (MTI)

C. New Module Incorporation phase
1. Module Incorporation Implementation
Module Incorporation Implementation (MII)
2. Module Incorporation Test Implementation
Module Incorporation Test Implementation (MITI)

A.2 The preparation phase

Work Product Criteria for Module Selection

Audience

Software designers, maintainers

Preconditions

- An understanding of system requirements, software module upgrade concept, and information hiding principle.
- Some knowledge of system structure

Purpose

- Describe which module is selected
- Describe what factors have been considered or why the above module is selected

Additional Criteria

- The module selection must follow above module selection criteria.
- The secret of the module is clearly documented.
- The services offered by this module is briefly described.
- The modules used, if any, are listed

Work Product Criteria for Project Overview

Audience

Managers, upgrade team members, software designers and maintainers

Preconditions

An understanding of system requirements, software module upgrade concept, and some knowledge of system structure

Purpose

- Present an overview of the project
- Name the tasks in the project, and, for each task, give: a unique task id, a brief description of the task, and the estimated amount of time needed in person-hours

---

Work Product Criteria for Module Test Plan

For the module under test, M:

Audience

Those considering running or modifying M's testing

Preconditions

A thorough understanding of M's specification and some understanding of M's implementation

Purpose

- Serve as a planning tool for development and evaluation of the test input strategy
- Document the test implementation

Additional Criteria

- The test plan must be sufficient to provide an estimate of both the adequacy of the test inputs and the feasibility of the test implementation.

- The following sections are required:

  assumptions
  State any assumptions, not contained in M's interface specification, on which the testing depends.

  test implementation strategy
  Describe the key aspects of the test implementation

  test case selection strategy
  Critical implementation states and parameter values are specified.
  Tests are planned to invoke every access routine in both normal and exceptional situations
  100 percent statement coverage is expected.

- For each service module M, a simple interactive tester is provided.

A.3 The module development phase

Work Product Criteria for Module Interface Specification

Audience
Module designer, implementor, tester and user

Preconditions
An understanding of the interface specification language

Purpose
Describe the assumptions users are permitted to make about module behavior, independent of the underlying implementation

Additional Criteria

- Describes the interface precisely and completely.
- Follows the interface specification language rules.
- Satisfies the interface design criteria, where practical.
- Is testable: no unjustifiable controllability or observability problems.

Work Product Criteria for Module Implementation

Audience
Module implementor, tester

Preconditions
Understanding of interface specification and implementation programming language

Purpose
Provide code whose behavior satisfies the specification

Additional Criteria

- Correctly implements the specification
- Satisfies the Code Format Rules
- Satisfies the Code Verification Rules
- Is testable: No unjustifiable controllability or observability problems
- Includes a set of default exception handlers, providing trivial exception reporting

---

Work Product Criteria for Module Test Implementation

For the module under test, M:

Audience
Those considering running or modifying M's testing

Preconditions

A thorough understanding of M's specification and some understanding of M's implementation

Purpose

Implement the test plan as simply and inexpensively as possible

Additional Criteria

Implementation follows the test plan

Test runs to completion, with correct results and 100 percent statement coverage

Code obeys Code Format Rules

A.4 The module incorporation phase

Work Product Criteria for Module Incorporation Implementation

Audience

Software designers, maintainers

Preconditions

An understanding of software module upgrades, interface specification (or the requirements specification for application modules), and implementation programming language.

Be familiar with the module incorporation plan.

Purpose

Provides code whose behavior meets the system requirements, with the new module incorporated.

Additional Criteria

No unnecessary changes to the existing code.

All changes made are recorded, and inspected.

Work Product Criteria for Module Incorporation Test Implementation

Audience

Those who consedering running or modifying the module incorporation testing.

Preconditions

A thorough understanding of the system requirements and some understanding of module implementation and module incorporation implementation.

Purpose

Implement the module incorporation test plan as simply and inexpensively as possible

Additional Criteria

Implementation follows test plan.
Test runs to completion, with correct results and 100statement coverage.
Code obeys Code Format Rules.

Appendix B

DCBL Work Products

This appendix consists of a set of work products produced in the DCBL module upgrade project. They are shown in the order in which they were developed.

The equipments used are Sun 3/60 running Unix operating system release 4.1 and IBM-PC running MS-DOS version 3.3. The programming languages used are C on Unix, and Microsoft Optimized C version 5.1 on PC.

The project was started in the Summer of 1990 and completed in the Spring of 1991. The main purpose of the DCBL project was to verify and improve the software module upgrade procedure we defined, and present an example of applying the procedure. The time cost of each step was recorded.

B.1 The preparation phase

Module Selection

Brief description of the module

name
DCBL (Device Control Block List)

secret
TICS device control block list structure and its access method.

service

provide sequential traversal through the list and random access to elements in the list by device name or device number.

For the sequential traversal, restart must be always allowed. The module should provide an access program that will enable the user of the module to access the current element in the list (called cursor) and allow the user to tell if the end of the list has been reached.

The random access should not interfere the sequential access.

Why DCBL

critical

The device control block list is the heart of the TICS control system, since TICS is a device control system. The safety of the control block list directly affects the system reliability.

independent

The device control block list management is relatively independent, in that it does not heavily depend on other modules.

cost effective

The upgrading of DCBL will isolate the list structure from its usage. The list may be accessed in hundreds of places, but the structure of the list is used only in the DCBL module. Hence, the maintainability is improved. Pointer errors caused by the list accessing could also be eliminated through disciplined inspection and testing. There should be no major difficulties in upgrading this module.

Project Overview

The project consists of following steps:

module interface specification — 40 hours

This task is to study the current TICS code, decide what is the interface specification of the dcbl module. The interface specification (dcbl.h and dcbl.intspec) should be inspected.

module implementation — 40 hours

This task includes the development of dcbl.c and the inspection on the implementation. This includes stating the assumptions of how the dcbl will be used, and then prove that all pointers returned by sequential or random access service routines will be pointers to legal device control blocks.

module testing — 40 hours

This task includes dcbl test plan refinement, test code development, and execution of the test code.

module incorporation implementation — 130 hours

This task includes changing the makefiles to add the dcbl module into the system, and changing the files related to device control block list access. That is to find all files accessing the device control block list, and replace the in-line code for traversing the list by procedure calls to dcbl module. Changes are only made to the TICS control program.

module incorporation testing — 40 hours

This is to best ensure that the system is upgraded correctly. Every change made to the system should be exercised at least once. It may be necessary to develop an application using TICS language that contains more than two devices.

DCBL Module Interface Specification - Task Plan

Purpose

study the TICS code related to device control block list, develop the dcbl module interface specification.

Implementer

prerequisition

understanding of software engineering principles (e.g. information hiding)
work product criteria of Interface Specification, Module Implementation, Test Plan, and Test Implementation
knowledge of Microsoft Optimized C version 5.1
knowledge of PC-Dos/Ms-Doc,
knowledge of Unix
some knowledge of TICS structure

names
Dan Hoffman , Rolf Keitel, Marvin Li

Resources needed

utilities
GREP, TCOV, DIFF, PMAKE, PGMGEN

work directory
Unix: csr: mli/tic/lm
PC: c:/tics/v3x
PC: c:/tics/lm
PC: d:/tics_sav

Streategy
Study the current TICS code, understand how TICS controls the DCB list.

Find out, exactly, what services the DCBL module should provide Should the DCBL module be responsible for creating the list? How about deleting devices from the DCB list?

Decide field access strategy: Should fields of DCB be accessed by access routines provided by this module only or allow user of the module to access fields in-line? Are users of the module responsible for maintaining the next field?

DCBL Module Implementation - Task plan

Purpose
develop dcbl.c dcbl_e.c on Unix and PC

Implementer

prerequisition
understanding of software engineering principles (e.g. information hiding)
work product criteria of Interface Specification, Module Implementation, Test Plan, and Test Implementation
knowledge of Microsoft Optimized C version 5.1

knowledge of PC-Dos/Ms-Dos,
knowledge of Unix
some knowledge of TICS structure

names
Marvin Li

Resources needed

utilities
GREP, TCOV, DIFF, PMAKE, PGMGEN

work directory
Unix: csr: mli/tic/lm
PC: c:/tics/v3x
PC: c:/tics/lm
PC: d:/tics_sav

Streategy

- Decide under which environment the module is to be developed and tested Since this is a stand alone DCB list maintenance module, it can be developed under either Unix Workstation environment or under the production environment, which is PC-Dos. The Unix environment is preferrable because it has much better support facilities such as PGMGEN, tcov, and nice laser printers. Also it is accessible from J-hut and from remote site through modem.

  Of couse, it is important to be able to transfer files between PC and Unix hosts easily.
- Design major data structures: PPCB is defined in tistruct.h. It should be extracted from tistruct.h and put into DCBL.h.

DCBL Module Testing - Task Plan

Purpose
to test if the newly developed module meets its specification

Implementer

prerequisition
knowledge of module testing
work product criteria of Interface Specification, Module Implementation, Test Plan, and Test Implementation
knowledge of Microsoft Optimized C version 5.1
knowledge of PC-Dos/Ms-Doc, Unix
knowledge of PGMGEN and TCOV
some knowledge of DCBL module implementation

names
Marvin Li

Resources needed

utilities
GREP, TCOV, DIFF, PMAKE, PGMGEN

working directory
Unix: csr: mli/tic/lm
PC: c:/tics/v3x
PC: c:/tics/lm
PC: d:/tics_sav

Streategy

Develop the test plan

Develop the test script
to build an batch tester according to the test plan

Implement the interactive tester
The interactive tester is aimed at debugging purpose.

Run the tests under Unix
The Unix environment is nicer and easier to access than the Dos environment. Therefore, the test suite is developed and run under Unix environment.

Run the tests under Dos

After the test suite and module is stable, run the test under Dos environment, since it is a different environment after all.

Files affected

dcbl.c, dcbl.tplan, dcbl_b.script
dcbl_i.c, dcbl.h, tstruct.h
dcbl.intspec, dcbl_e.c

Note

For simplicity reasons, the PPCB structure that describes the list elements of dcbl is extracted from tistruct.h and put in tstruct.h.

DCBL Module Test Plan

pgmgen driver
no stubs
all tests run under Unix and Dos
C micro:
  NUM(i) = (i + 1) * 10
  NAM(i) = (itos(NUM(i)) /*string version of NUM(i)*/
C function:
  gendl(n)
    create a dcb list of length n, with
    the number field of ith dcb = NUM(i)
    the name field of ith dcb = NAM(i)
statement coverage measured using the Unix utility tcov
  (Unix tests only)
critical implementation state values
  list length: [0,N]
  list elements: ith element has name and number fields
    generated by gendl.
  sequential traversal status:
    before s_start
    after s_start: number of s_next range in [0,length(list)]
exceptions
  empty list, issue g_cur, s_next
  non-empty list, move the cursor to the end of the list,

```
        issue g_cur, s_next
    normal
        for each dlsiz in [0,N-1] /*list size*/
            call gendl(dlsiz)
            check g_cur if dlsiz > 0
            check g_bynam, g_bynum, g_end
            issue s_start
            for each ns in [0,dlsiz-1] /*number of s_next*/
                /*check direct access,including NULL return*/
                for each i in [0,dlsiz]
                    issue g_bynum(NUM(i))
                    issue g_bynam(NAM(i))
                check g_cur
                    check current dcb's number field = NUM(ns)
                    check current dcb's name field = NAM(ns)
                check g_end
                check s_next
        /*test if field changes will affect list traversal*/
        call gendl(N)
        for each i in [0,N-1]
            /*for the ith dcb in the list*/
            set its number field = NUM(i+1)
            set its name field = NAM(i+1)
            issue s_next
        for each i in [0,N]
            check g_bynum(NUM(i+1))
            check g_bynam(NAM(i+1))
```

Module incorporation implementation plan

Purpose

Add the new module DCBL into system and replace in-line list traversal code with calls to DCBL

Implementer

prerequisition
understanding of software engineering principles (e.g. information hiding)
work product criteria of Interface Specification, Module Implementation, Test Plan, and Test Implementation
knowledge of Microsoft Optimized C version 5.1
knowledge of PC-Dos/Ms-Dos,
some knowledge of TICS structure

names
Marvin Li

Resources needed

utilities
GREP, TCOV, DIFF, PMAKE, PGMGEN

work directory
PC: c:/tics/v3x
PC: c:/tics/lm
PC: d:/tics_sav

Streategy:
1. Establish work base
   a. ensure that the current version compiles and runs
   b. make a list of compilation and linking errors, if any
2. Backup the working version
   a. backup the whole directory: c:/tics to floppy disk, store off site
   b. backup c:/tics/v3x/*.* to d:
3. Establish criteria for changing the source code:
   a. every file changed should include dcbl.h
   b. all changes made should be located by "grep dl_[sg]_ *.c"
4. Change makefiles
5. Replace in-line traversal code with calls to DCBL, see subtask plan 1
6. Proceed for rest of files in the same pattern
7. Inspect the changes

Subtask plan 1
1. Copy the following files into c:/tics/v3x
   dcbl.c dcbl.h dcbl_e.c
2. Add dl_s_init into ticsmain.c

3. Replace the in-line code for traversing the DCB list in tischedu.c
   by function calls to DCBL module
4. Compile and run
5. Fix problems until it runs

Note:

1. TICS CP obj files sorted in alphabetic order (the postfix .obj is omited):
   cursor dspreloc getput gplot gtype keys logplot lsetcomp mdrv numlock pardisp plotdrv plotprep prepare reloc retrieve splint stbcamac stbmodic stboptom ticsmain tisalarm tisask tisbypas tiscale tiscbin tiscbout tiscedit tischedu tiscmd tiscmdds tiscond tisconf tisconn tiscons tisconv tiscpp1 tiscpp2 tiscpp3 tisdcb tisdev tisdev2 tisdev3 tisdocmd tisdos tisdsp tisdsp1 tisdsp2 tisdspdv tisdspfi tisdsphl tisdspmo tisdsppp tisdspps tisdspsd tisdtran tisedi2 tisedit tiselect tisfatal tisfile tisfind tisfkey tisfunc tishelp tishist tishrc tiskeyp tislog tismemfi tismot tismouse tismsg tisnoega tisnonet tisopti tispawn tisplot tisps tisreloc tistatus tistatxt tistbl tisterm tisutil tiswarn tiswatch tiswin
2. C files to be changed:
   tisbypas tiscale tiscedit tischedu tiscmd tiscmdds tiscpp2 tiscpp3 tisdcb tisdev tisdev2 tisfind tisfunc tislog tismot tismouse tisopti tisreloc tiswarn

---

DCBL Module Incorporation Test Plan

Purpose

Test to see if the new module was added into system and the changes made to the other files are correct.

Implementer

prerequisition

understanding of software engineering principles (e.g. information hiding)
work product criteria of Interface Specification, Module Implementation, Test Plan, and Test Implementation
knowledge of Microsoft Optimized C version 5.1
knowledge of PC-Dos/Ms-Dos,
some knowledge of TICS structure

names
Marvin Li

Resources needed

utilities
GREP, TCOV, DIFF, PMAKE, PGMGEN

work directory
PC: c:/tics/v3x
PC: c:/tics/lm
PC: d:/tics_sav

Implementation strategy

1. Use memory buffers to simulate device status and value
2. Figure out how to print debugging messages on the screen
3. Record debugging message by redirecting standard i/o

Test case selection strategy

1. build application that contains at least two devices
2. check if the DCBL is linked into the system
3. check if every piece of new or modified code is executed at least once.

related .c files (the postfix .c omited)
tisbypas tiscale tiscedit tischedu tiscmd tiscmdds tiscpp2 tiscpp3 tisdcb tisdev tisdev2 tisfind tisfunc tislog tismot tismouse tisopti tisreloc tiswarn

B.2 The module development phase

DCBL module interface specification – syntax

```
/*****types*****/
typedef struct dl_item {
        char id;                    /* control block id byte */
        int number;                 /* device number          */
        int status;                 /* 16 bit status flag word */
        struct dl_item *next;       /* link for element manager */
        char *firstcmd;             /* head of command list   */
        char name[11];              /* element name           */
        /* ... other data */
} dl_item;

/*****access programs*****/
void dl_s_init();
/*      void dl_s_init(list_header)
*       dl_item *list_header;
*/

void dl_s_start();

void dl_s_next();

dl_item *dl_g_cur();

/*boolean*/ int dl_g_end();

dl_item *dl_g_bynam();
/*      dl_item dl_g_bynam(dev_name)
*       char *dev_name;
*/

dl_item *dl_g_bynum();
/*      dl_item dl_g_bynam(dev_number)
*       int dev_number;
*/

/*exceptions*/
void dl_end();

void dl_legpos();
```

DCBL module interface specification – semantics

Assumptions

1. dl_s_init must be called before any other call to this module.
2. lp, the parameter to dl_s_init, must be either NULL or a pointer to a circular list of dl_item's.

Specification state

dcblst: sequence [0..] of DCBTYP; pos: integer;

Set call effects

dl_s_init(lp):
  dcblst := list associated with lp
  pos := 0
dl_s_start(): pos := 0
dl_s_next(): pos := pos+1

Get call return values

dl_g_cur() = dcblst[pos]
dl_g_end() = not pos in [0,length(dcblst)-1]
dl_g_bynam(nam) = an item with its name field being nam,
  or NULL if no one matches.
dl_g_bynum(num) = an item with its number field being num,
  or NULL if no one matches.

Exceptions

dl_s_next():
  (dl_end: pos = length(dcblst))
dl_g_cur():
  (dl_legpos: not pos in [0,length(dcblst)-1])

Local types

DCBTYP ::= tuple of {
  number: integer,
  name: string,
  data: DATA
}
DATA is a type that stores all the information related to
the DCB. We eliminate its details here.

DCBL module implementation – Makefile

```
INC = .

CFLAGS = -I$(INC)

export: dcbl.o dcbl_e.o

dcbl_b: dcbl_b.o dcbl.o
        cc -o dcbl_b dcbl_b.o dcbl.o

tcov: dcbl_b.o
        cc -a -c -I$(INC) dcbl.c
        cc -a -o dcbl_b dcbl_b.o dcbl.o
        dcbl_b
        tcov dcbl.c
        rm -f dcbl.o # tcov version is
                dangerous to leave around

dcbl_i: dcbl_i.o dcbl.o dcbl_e.o
        cc -o dcbl_i dcbl_i.o dcbl.o dcbl_e.o

dcbl_b.o: $(INC)/dcblu.h $(INC)/tstruct.h

dcbl_i.o: $(INC)/dcblu.h $(INC)/tstruct.h

dcbl.o: $(INC)/dcblu.h $(INC)/tstruct.h

dcbl_e.o: $(INC)/dcblu.h $(INC)/tstruct.h

dcbl_b.c: dcbl.script
        pgmgen dcbl.script
        mv test.c dcbl_b.c

lint:
```

```
        lint -I$(INC) dcbl_i.c dcbl_e.c dcbl.c

clean:
        rm -f dcbl_i dcbl_b *.o dcbl_b.c dcbl.tcov dcbl.d

doc:
        pr -188 Makefile dcbl.intspec *.h *.c dcbl.tplan \
                dcbl.script
```

DCBL module implementation – dcbl.c

```
#include "dcbl.h"
#include <stdio.h>

/*****module state*****/

dl_item *head;
dl_item *cur;

/*
*definition
*    define EL(h,n)
*        address of the nth element in the DCB list headed by h,
*        where h is the address of element 0.
*
*    a memory block is a "legal DCB", if
*        1. it is of size at least sizeof(dl_item)
*        2. let q be the address of the block, then
*           (dl_item *)q->name contains a '\0', and
*           (dl_item *)q->next points to another legal DCB.
*
*    a legal DCB list is a circularly linked list of legal DCBs.
*
*implementation state invariant
*    1. head is either NULL or the address of a DCB in a legal
```

```
*       DCB list
*    2. cur is either NULL or an address of a legal DCB
*
*abstraction function
*       pos = i, where EL(head,i) = cur
*       dcblst = <> iff head = NULL
*       (forall i)(if i in [0,len-1] then
*           dcblst[i].number = EL(head,i).number, and
*           dcblst[i].name = EL(head,i).name, and
*           dcblst[i].data = EL(head,i).data)
*       where len is the number of different DCBs in the DCB list
*       pointed to by head
*/


/*****access programs*****/

/*
*dl_s_init(lp) has to be called again if
*lp is relocated.
*/
void dl_s_init(lp)
dl_item *lp;
{
        cur = head = lp;
}

void dl_s_start()
{
        cur = head;
}

void dl_s_next()
{
        if (cur == NULL)
                dl_end();
        else if ((dl_item *)cur->next == head)
                cur = NULL;
        else
```

```
                cur = (dl_item *) cur->next;
}

dl_item *dl_g_bynam(nam)
char *nam;
{
        dl_item *p;

        if (head == NULL) return(NULL);
        p = head;
        do {
                if (!strcmp(p->name,nam))
                        return(p);
                p = (dl_item *)p->next;
        } while (p != head);
        return(NULL);
}

dl_item *dl_g_bynum(num)
int num;
{
        dl_item *p;

        if (head == NULL) return(NULL);

        p = head;
        do {
                /*loop invariant: p points to a
                legal DCB in a circular link*/
                if (p->number == num)
                        return(p);
                p = (dl_item *)p->next;
        } while (p != head);
        return(NULL);
}

dl_item *dl_g_cur()
{
        if (cur == NULL)
```

```
                dl_legpos();
        return(cur);
}

/*boolean*/ int dl_g_end()
{
        return(cur == NULL);
}

void dl_g_dump()
{
        dl_item *p;

        if (head == NULL) {
                printf("List is empty\n");
                return;
        }
        p = head;
        do {
                printf("name: |%s|, number: %d\n",
                        p->name, p->number);
        } while ((p = (dl_item *)p->next) != head);
}
```

DCBL module implementation – dcbl_e.c

```
void dl_end()
{
        printf("dl_end occured\n");
}

void dl_legpos()
{
        printf("dl_legpos occured\n");
}
```

DCBL module test implementation – dcbl_i.c

```
#include "dcbl.h"
#include <stdio.h>
#include <assert.h>

#define QUIT 0
#define S_INIT 1
#define S_START 2
#define S_NEXT 3
#define G_END 4
#define G_CUR 5
#define G_BYNAM 6
#define G_BYNUM 7
#define G_DUMP 8

#define BUFLEN 81
#define DLSIZ 10

dl_item cb[DLSIZ];

int nextcall()
{
        int reply;
        char s[81];

        do {
                printf("\nEnter command:\n");
                printf("\t0:quit\n");
                printf("\t1:s_init\n");
                printf("\t2:s_start\n");
                printf("\t3:s_next\n");
                printf("\t4:g_end\n");
                printf("\t5:g_cur\n");
                printf("\t6:g_bynam\n");
                printf("\t7:g_bynum\n");
                printf("\t8:g_dump:");
                gets(s);
```

```
                if (sscanf(s,"%d",&reply) != 1)
                        /*user error - stay in loop*/;
                        reply = -1;
        } while (reply < 0 || reply > G_DUMP);
        return(reply);
}

int readint(msg)
char *msg;
{
        int reply,found;
        char s[BUFLEN];

        found = 0;
        while (!found) {
                printf(msg);
                gets(s);
                if (sscanf(s,"%d",&reply) == 1)
                        found = 1;
        }
        return (reply);
}

void load_list(num)
int num;    /* num of nodes in list */
{
    int i, j;
    dl_item *dev;

    for (i = 0; i < num; i++) {
        cb[i].number = i;
        cb[i].next = &cb[i+1];
        sprintf(cb[i].name, "%d", i);
    }
    cb[num-1].next = cb;
}

main()
{
```

```
int reply;
int i;
char nam[80];
int num;
int siz;

printf("Input the list length: [0-10] ");
scanf("%d", &siz);
assert(siz <= DLSIZ);
load_list(siz);
while ((reply=nextcall()) != QUIT) {
        switch(reply) {
        case S_INIT:
                dl_s_init(siz ? cb : NULL);
                break;
        case S_START:
                dl_s_start();
                break;
        case S_NEXT:
                dl_s_next();
                break;
        case G_END:
                printf("g_end return: %d\n",
                        dl_g_end());
                break;
        case G_CUR:
                printf("g_cur return: %lx\n",
                        dl_g_cur());
                break;
        case G_BYNAM:
                printf("Input the name: ");
                scanf("%s", nam);
                printf("g_bynam return: %lx\n",
                        dl_g_bynam(nam));
                break;
        case G_BYNUM:
                printf("Input the number: ");
                scanf("%d", &num);
                printf("g_bynum return: %lx\n",
```

```
                                dl_g_bynum(num));
                        break;
                case G_DUMP:
                        dl_g_dump();
                        break;
                }
        }
}
```

DCBL module test implementation – dcbl.script

```
module
        dl_

accprogs
        <s_init,s_start,s_next,g_cur,g_end,g_bynam,g_bynum>

exceptions
        <end,legpos>

globcod
{%
#include <stdio.h>
#include "dcbl.h"
#include <assert.h>

/*set DOS = 1 if test is to be run under DOS,
* else set DOS = 1
*/
#define DOS 0

#define DLSIZ 100
#define NUM(x) (((x)+1)*10)
#define NAM(x) (itos(NUM(x)))
#define END (dlsiz ? 0 : 1)
#define ADDR(x) ((x) < dlsiz ? &dls[x] : NULL)
#define F_NAM (dl_g_cur()->name)
#define F_NUM (dl_g_cur()->number)
```

```
static dl_item dls[DLSIZ];
static dl_item *head;
static int dlsiz,ns,i;

char *itos(i)
int i;
{
        static char buf[80];

        sprintf(buf,"%d",i);
        return(buf);
}

static void gendl(n)
int n;
{
        int i,j;

        if (n == 0) {
                head = NULL;
                dl_s_init(NULL);
                return;
        }
        assert(n <= DLSIZ && n > 0);
        for (i = 0; i < n; i++) {
                dls[i].number = NUM(i);
                strcpy(dls[i].name,NAM(i));
                dls[i].next = &dls[i+1];
        }
        head = dls[n-1].next = dls;
        dl_s_init(dls);
}

int cmp_addr(x1, x2)
char *x1, *x2;
{
        extern char *to_far();
#if DOS == 1
```

```
        return(to_far(x1) == to_far(x2));
#else
        return(x1 == x2);
#endif
}

int prt_addr(aval,eval)
int aval,eval;
{
        printf("\tExpected value:%lx.  Actual value:%lx\n",eval,aval);
}

#if DOS == 1
char *to_far(x)
char *x;
{
        char *ptr;
        unsigned l,h;

        ptr = (char *)&x;
        l = *(unsigned *)ptr;
        h = *(unsigned *)(ptr+2);
        h += (l / 16);
        l = l % 16;
        *(unsigned *)ptr = l;
        *(unsigned *)(ptr+2) = h;
        return((char *)x);
}
#endif

%}

cases

/*****exceptions*****/

<gendl(0).s_next, end, dc, dc, dc>
<g_cur, legpos, dc, dc, dc>
<gendl(3).s_next.s_next.s_next.g_cur, legpos, dc, dc, dc>
```

```
<s_next, end, dc, dc, dc>

/*****normal case*****/

{%
/*list size*/
for (dlsiz = 0; dlsiz < DLSIZ; dlsiz++) {
        gendl(dlsiz);
        if (dlsiz)
%}
        <, noexc, g_cur(), head, addr>
        <, noexc, g_end(), END, bool>
        <, noexc, g_bynam(NAM(0)), head, addr>
        <, noexc, g_bynum(NUM(0)), head, addr>

        <s_start, noexc, dc, dc, dc>
{%
        /*number of s_next*/
        for (ns = 0; ns < dlsiz; ns++) {
                /*check direct access,including NULL return*/
                for (i = 0; i <= dlsiz; i++) {
%}
                        <, noexc, g_bynam(NAM(i)), ADDR(i), addr>
                        <, noexc, g_bynum(NUM(i)), ADDR(i), addr>
{%
                }
%}
                <, noexc, g_cur(), ADDR(ns), addr>
                <, noexc, F_NUM, NUM(ns), int>
                <, noexc, F_NAM, NAM(ns), string>
                <, noexc, g_end(), 0, bool>
                <s_next, noexc, dc, dc, dc>
 {%
         }
 }
 %}

 /*test if field changes will affect list traversal*/
 {%
```

```
gendl(DLSIZ);
for (ns = 0; ns < DLSIZ; ns++) {
        F_NUM = NUM(ns+1);
        strcpy(F_NAM,NAM(ns+1));
        dl_s_next();
}
%}
{%
for (ns = 0; ns <= DLSIZ; ns++) {
%}
        <, noexc, g_bynam(NAM(ns+1)), ADDR(ns), addr>
        <, noexc, g_bynum(NUM(ns+1)), ADDR(ns), addr>
{%
}
%}
```

VITA

Surname: **Li** Given Names: **Marvin Xuewen**
Place of Birth: **ZheJiang, China** Date of Birth: **January 4, 1963**

Educational Institutions Attended:

Qinghua University	1979 to 1984
Peking University	1984 to 1986
University of Victoria	1989 to 1991

Degrees Awarded:

B.Sc.	1987	Qinghua University, Beijing, China

Honors and Awards:

University of Victoria Graduate Supplement, 1989/90
University of Victoria Fellowship, 1990/91

Partial Copyright License

I hereby grant the right to lend my thesis (the title of which is shown below) 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 this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

A Procedure for Software Module Upgrades

Author: [redacted]

Marvin Xuewen Li

April 15, 1991