Canada

PROGRAMMING GUIDE

INFORMATION SYSTEMS MANAGEMENT

Regional Economic Expansion

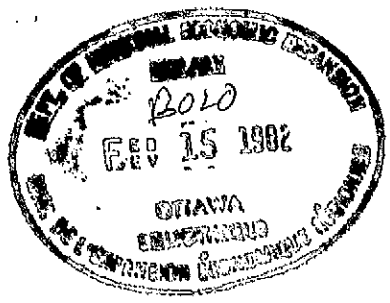
Expansion Économique Régionale

Canada.

DEPARTMENT OF REGIONAL AND ECONOMIC EXPANSION

SYSTEMS DEVELOPMENT LIFE CYCLE METHODOLOGY

PROGRAMMING GUIDE



OCTOBER, 1981

I N D E X

1. PROJECT MANAGEMENT HANDBOOK

2. DELIVERABLES REFERENCE MANUAL

3. USER'S GUIDE

4. ANALYSIS GUIDE

5. DESIGN GUIDE

6. PROGRAMMING GUIDE

NOTE: It is recognized that all roles referred to throughout this document will be filled by persons of either sex. However, to maintain readability, personel pronouns of the male gender are used.

He should be read as he/she.

His should be read as his/hers.

Him should be read as him/her.

SYSTEMS DEVELOPMENT LIFE CYCLE METHODOLOGY

PROGRAMMING GUIDE

TABLE OF CONTENTS

		Page
1.	INTRODUCTION	
	1.1 Purpose	1.1
	1.2 Objectives	1.2
	1.3 Scope	1.3
2.	SUMMARY OF SYSTEM DEVELOPMENT LIFE CYCLE	
	2.1 Overview Data Flow Diagram	2.1
	2.2 Phase Summaries	2.2
3.	ROLES IN SYSTEM DEVELOPMENT LIFE CYCLE	
	3.1 Major Responsibilities by Phase (Matrix)	3.1
	3.2 Summary of Roles	3.2
4.	USER ROLES	
	4.1 Introduction	4.1
	4.2 Verify Completeness of Module Specification	4.3
	4.3 Perform Detailed Design of Module	4.5
	4.4 Verify Detailed Design	4.6
	4.5 Code Module	4.8
	4.6 Verify Code	4.12
	4.7 Test Module	4.17
	4.8 Verify Completeness of Module Development	4.26
	APPENDIX	

SECTION 1

INTRODUCTION

1.1 Purpose
1.2 Objectives
1.3 Scope

1. INTRODUCTION

1.1 Purpose

This guide has been prepared for the purpose of providing programmers and their managers with a guide to performing programming tasks during the Development phase of an information systems development project.

Although the underlying philisophy contained herein is a belief that the approach to program development must be disciplined and structured to be successful, these guidelines are flexible to allow for programmer initiative to develop.

Each programming activity is described in terms of its objectives, inputs, methods, working documents and deliverables.

1.2 Objectives

This guide suggests techniques and methods for:

- developing programs that meet the Functional Specifications;
- developing programs that meet technical requirements, such as performance and security criteria;
- maximizing programmer efficiency;
- ensuring that programs are reliable through comprehensive testing procedures; and
- developing programs that are maintainable.

1.3 Scope

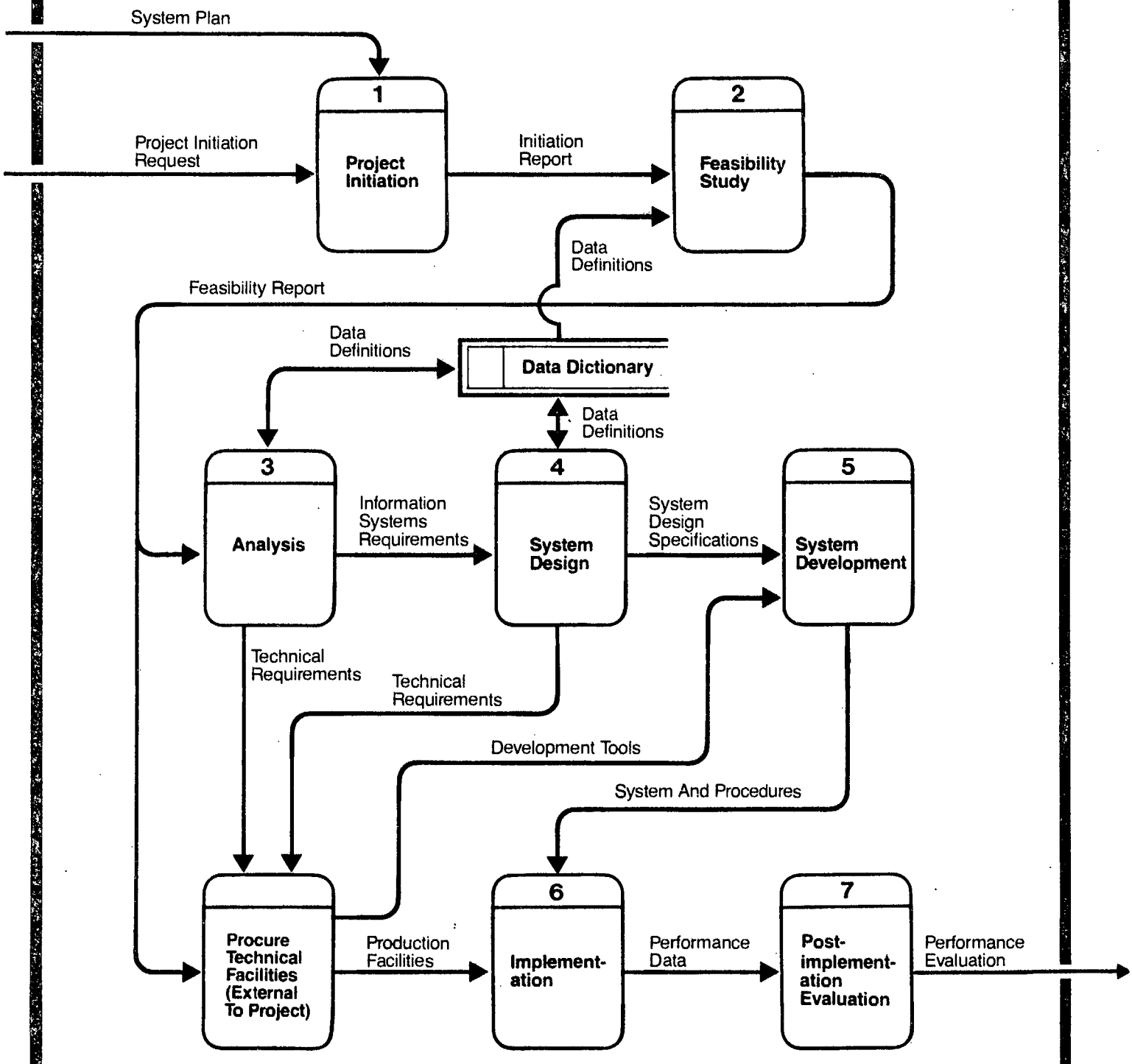
This guide only addresses the primary responsibilities of a programmer. They are to design, develop, document and test program modules. The role of a programmer ususally involves ancillary tasks such as support during implementation and participation in acceptance testing. These are outside the scope of this guide which addresses specifically methods relating to program development.

SECTION 2

SUMMARY OF SYSTEM DEVELOPMENT LIFE CYCLE

2.1 Overview Data Flow Diagram
2.2 Phase Summaries

Systems Development Life Cycle



2.2 Phase Summaries

The preceding diagram illustrates the phases comprising the System Development Life Cycle of most information systems projects. It depicts how the cycle commences with the receipt of a Project Initiation Request and ends with the preparation of a Post-Implementation Evaluation Report following system implementation. The end of each phase represents a major checkpoint where managment, external to the project, may review the continuing viability of the project and, as appropriate, commit only the resources needed to complete the following phase.

Summaries of each phase are as follows:

- Project Initiation

  Every project begins with the identification of an opportunity to be exploited, a problem to be solved, or a requirement to be satisfied. This phase starts when a request is received (on a Project Initiation Request form) from a user. The request is first screened to ensure that it is properly authorized, that the source of development funds is identified and that there is justification for proceeding further. Following this, details of the request are documented by a (Business) Systems Analyst. He/she prepares a brief Initiation Report which documents the issues to be addressed, objectives, scope, benefits, timeframe, policies, constraints and potential solution strategies.

  The objective of the report is to outline for management the initial perception of the issue, and to recommend an action plan to study the feasibility of various solutions.

  Normally, preparation of the report takes one half of a day or so.

- Feasibility Study

  This phase involves the Analyst working together with user management in the research and analysis of subject related data in order to identify various solutions. These solutions are both manual and automated, and are evaluated for their relevance and costs/benefits.

The overall objectives are to select a solution (or path), to develop a conceptual system design and to secure further resources in order to perform a detailed analysis of the information requirements.

The Feasibility Study is carried out at a general, or conceptual, level. It provides management with an early opportunity to evaluate the project's viability before any substantial amounts of money have been expended, and to re-evaluate it in relationship to the user's priorities and strategies.

. Analysis

The Feasibility Study examined the issue being addressed by the system at a general level. The Analysis phase examines it at a very detailed level. The precise business processes and the set of information forming the business system is clearly defined.

This definition establishes the basis for outlining the system from a user perspective. For EDP systems, this means that at the end of this phase the user will know what information is included in the system and what business processes will be machine assisted.

The Analysis phase involves substantial end-user participation since it is during that phase that the business content of the system is documented in preparation for the design and development phases.

. System Design

Whereas the Analysis phase defined that "what" of the system, the System Design phase defines "how".

The specifications delivered by the Analysis phase represents the bridge between the user community, who collectively define the business requirements for the project, and the project designers who design a system to address the requirements.

User participation in this phase involves reviewing and approving more detailed aspects of the system such as report and screen layouts, office procedures, forms, etc.

- System Development

  The objective of this phase is to develop the working procedures and if automated, the computer programs according to the system design specification. Testing of the procedures and programs is also done to ensure that all components of the system work properly.

- Implementation

  In this phase the working procedures and programs developed are made operational. Users are trained in preparation for the live running of the new system, data files are converted from old media to new media and the new system is installed. Parallel running, when applicable, takes place.

- Post-Implementation Evaluation

  This phase studies the operational performance of the system for a pre-determined period and performs to management its conclusions and recommendations. Optionally, according to management's preferences, it may also study the effectiveness and efficiency of the development process itself.

SECTION 3

ROLES IN SYSTEM DEVELOPMENT LIFE CYCLE

3.1 Major Responsibilities by Phase (Matrix)

3.2 Summary of Roles

3.1 MAJOR RESPONSIBILITIES BY PHASE

PHASE	DELIVERABLE/TASK	APPROVAL AUTHORITY	MANAGEMENT AUDIT	USER		PROJECT TEAM			
				MANAGEMENT	STAFF	PROJECT MANAGER	SYSTEMS ANALYST	SYSTEMS DESIGNER	PROGRAMMER
1. Initiation	Initiation Report	Approve	Approve	Approve	Participate	Prepare			
2. Feasibility Study	Feasibility Report -User Requirements -Conceptual Solution	Approve	Approve	Approve	Participate	Review	Prepare		
3. Analysis	Requirements		Review	Approve	Participate	Review	Prepare	Participate	
	Approval Authority Submission	Approve		Approve		Prepare			
4. System Design	EDP Design Specification		Review		Participate	Approve	Review	Prepare	
	Design of User Aids			Approve	Participate	Review	Prepare	Participate	
	Approval Authority Submission	Approve		Approve		Prepare			
5. System Development	Program Design							Participate	Prepare
	Program Code						Review		Prepare
	Program Test					Approve	Review		Prepare
	System Test		Participate			Approve	Prepare	Perform	Participate
	Operations Manual					Approve	Participate	Prepare	Participate
	User Manual			Approve	Participate		Prepare	Participate	
	Procedures Manual			Approve	Update		Participate		
	Training Manual			Approve	Participate	Review	Prepare		
	Approval Authority Submission	Approve				Prepare			
6. Implementation	Acceptance Test		Participate	Approve	Perform		Participate	Participate	Participate
	Conversion			Approve	Participate		Participate	Participate	Participate
	Production Operation				Perform				
	Approval Authority Submission	Approve				Prepare			
7. Post-Implementation Evaluation	Evaluation Report	Approve	Participate	Approve	Participate	Approve	Prepare		

3.2 Summary of Roles

Approval Authority

The Approval Authority for any information systems project may be a systems management committee, a project steering committee, the head of ISM (Information Systems Management) or a senior functional manager depending upon the nature of development project.

The Approval Authority acts on behalf of the user by approving each of the end-of-phase submissions, by allocating resources to each project phase, and by maintaining control over the project's progress. These responsibilities are exercised through periodic receipt of documents and submissions from both the Project Manager and the Systems Assurance Manager. Refer to the Departmental ISM policy manual for specific policies related to the approval process.

Business Systems Analyst

See: Systems Analyst

Data Analyst

A Data Analyst provides functional guidance and support to the project on matters related to the logical representation of data in project specifications. A Data Analyst is a specialist in data and data relationships. External to projects, he models the department in terms of its data for the purpose of developing efficient, cost effective data management facilities, e.g., data bases. In order to achieve this he must develop data models for each project application and synthesize them into the Departmental data model.

NOTE: The Data Analyst's role may not be a full-time staff position. The role may be filled by staff with other responsibilities.

Inspector

An Inspector reviews project specifications in order to assure their quality prior to release external to the project. In this regard he examines specifications for consistency in level of detail and style, and adherence to standards. He also looks for incompatabilities among related documents.

Depending upon the size of the project team and the volume of project deliverables, the Inspector may be one individual appointed for the duration of the project, or he may be any member of the project team (for example, a Systems Analyst) appointed for the inspection of a single document.

An Inspector should not review specifications which he developed.

Programmer

A Programmer designs, develops and tests program modules using structured programming techniques. He may also be required to perform duties in system testing, acceptance testing, conversion and post-implementation support.

Project Manager

The Project Manager has overall responsibility for achieving the project goals through the day-to-day conduct of the project. In this respect, he develops operational plans and budgets, acquires the required resources, identifies and organizes the appropriate business and technical expertise, periodicaly submits plans, requests for approval and progress reports to the approval authority, coordinates with user management and the Systems Assurance Manager user participation in the project, conducts regular project management progress meetings and ensures effective quality control over project deliverables.

See Project Management Handbook for further details.

Steering Committee

See: Approval Authority

Systems Analyst

A Systems Analyst identifies, analyzes and specifies information systems requirements using structured analysis techniques. He may also carry out ancillary duties involving user interface such as development of user manuals, training, system conversion, and acceptance testing. Systems Analysts may be members of a user section or branch (Business Systems Analysts) or may be drawn from ISM staff.

See Analysis Guide for further details.

Systems Assurance Manager

The Systems Assurance Manager represents the departmental interest in a systems project and is responsible for ensuring that all user-related matters pertaining to quality control are addressed. Acting on behalf of the user, the Systems Assurance Manager:

- participates with the Project Manager in planning the commitment of user resources to the project;

- ensures that the appropriate level and quality of user resources are available to the project (i.e., that sufficently senior user personnel are assigned the key review and sign-off roles for all user-related deliverables produced by the project team);

- ensures that the user community's participation is comprehensive and active;

- verifies that the Project Manager has obtained user sign-off of all user-related deliverables (it is the responsibility of the Project Manager to obtain each sign-off);

- verifies that any changes to project plans which impact the user community have been agreed and approved by the user community;

- brings forward user concerns regarding the project to the Steering Committee for resolution if and when these concerns cannot be addressed through negotiations between the Project Manager and the user community;

- reports to the Steering Committee on user satisfaction with the project.

Ideally, the Project Manager and the Systems Assurance Manager should work cooperatively to support the successful execution of the project. Situations may arise, however, in which the Project Manager and the Systems Assurance Manager disagree (i.e., the Systems Assurance Manager may request, on behalf of the users, the expansion of the project scope, beyond the terms of reference understood by the Project Manager). The Project Manager and the Systems Assurance Manager are jointly responsible for making every effort to resolve any such disagreements to the mutual satisfaction of the project team and the user community. Disagreements should be brought forward to the Steering Committee only when resolution cannot be achieved through negotiation.

System Designer

A System Designer transforms information systems requirements, in the form of functional specifications, into system and sub-system design specifications using structured design techniques. Although a System Designer is normally the designer of the computer internals - system transactions, screens, files, input, output, etc. - this role may also encompass design of user aids such as training packages and user manuals.

See Design Guide for further details.

Technical Specialist

A Technical Specialist provides functional support and guidance to the project on matters of a technical nature. These would include hardware studies, telecommunications networking, technical feasibility of design alternatives, and acquisition and use of development tools.

He is considered "external" to any project and his abilities are shared on an organization-wide basis. This is to optimize the economic efficiency of using specialized technical staff.

User

The User's role in the Systems Development Life Cycle relates to those activities which have direct impact on him and his area of responsibility. These include:

- definition of systems subject matter;
- planning and provision of subject matter expertise;
- delegation of authority to staff assigned to participate in development activities;
- quality control over subject matter documented by the project team;
- training of staff;
- preparation of administrative environment for system installation;
- approval and acceptance of project deliverables.

In some sections or branches, user staff may also be engaged in carrying out development roles, such as systems analysis. These are not considered user roles.

See the User's Guide for further details.

SECTION 4

THE METHODOLOGY

		Page
4.1	Introduction	4.1
4.2	Verify Completeness of Module Specification	4.3
4.3	Perform Detailed Design of Module	4.5
4.4	Verify Detailed Design	4.6
4.5	Code Module	4.8
4.6	Verify Code	4.12
4.7	Test Module	4.17
4.8	Verify Completeness of Module Development	4.26

4. THE METHODOLOGY

4.1 Introduction

A systematic, disciplined approach to programming significantly increases the probability that programs will be developed efficiently and exhibit desirable characteristics. The approach described in this document consists of the following steps:

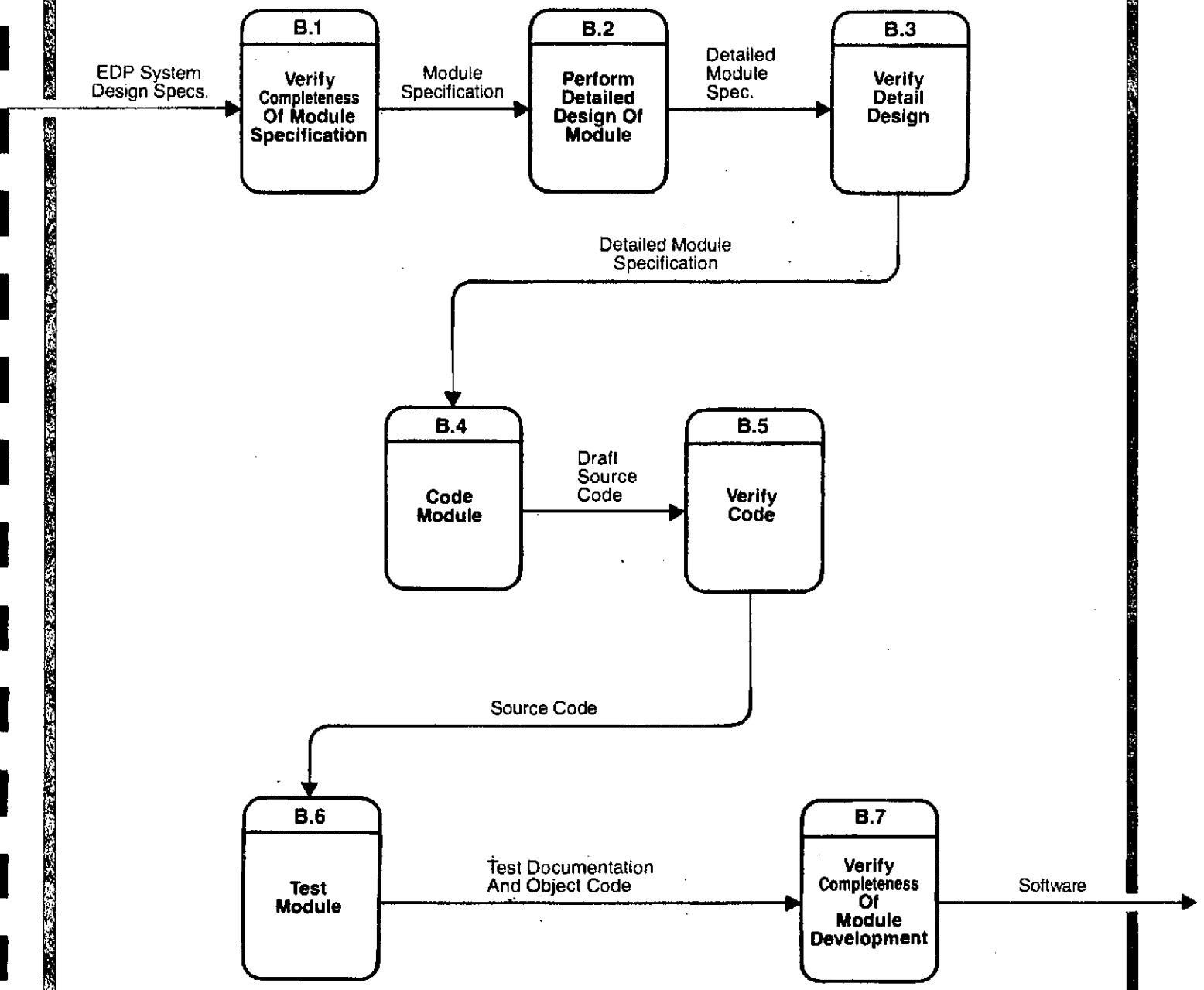
- Verify Completeness of Module Specification
- Perform Detailed Design of Module
- Verify Completeness and Accuracy of Detailed Design
- Code Module
- Verify Completeness and Accuracy of Code
- Test Module
- Verify Completeness of Module Development.

These steps, and their related deliverables, are shown in the data flow diagram on the preceding page.

In addition to having a detailed knowledge of this guide, it is desirable for programmers to be familiar with other documents, including:

- Project Management Handbook
- Analysis Guide
- Design Guide
- User's Guide
- Deliverables Reference Manual
- Functional Specifications of the System
- Phase and Project Plans
- Other Material (see Appendix A)
- Coding Guidelines (See Appendix C).

Activity 5B – Develop Software



4.2 Verify Completeness of Module Specification (5.B.1)

Objectives

- to verify that the General Module Specification is complete
- to familiarize the programmer with the module to be developed
- to understand the general design logic
- to request revision of the module documents, if necessary.

Inputs

- General Module Specification
- Designer's Notes

Methods

- Summary

  - Verify Specification Documents for Completeness
  - Module Familiarization
  - Understand Module Logic
  - Request Revision of Specification Documents

Note that some or all of these steps may not be necessary if the programmer was involved with the design of the module (e.g., it is often desirable for the programmer to take part in design walkthroughs).

- Verify Specification Documents

  It is important for the programmer to first verify that the specification documents are complete. This is done by carefully reviewing the specification, ensuring that it makes sense at a general level, and is consistent with documentation format given in the Deliverables Reference Manual.

  Any obvious or potential problems should be resolved immediately by consulting with the designer and, where appropriate, following the formal change control procedures that apply for the project.

- Module Familiarization

  The programmer normally receives a General Module Specification and Designer's Notes from the designer. The programmer is responsible for developing a detailed understanding of these documents, which may involve some brief discussions with the designer. Lengthy discussions at this point may indicate that the specification is too general, or may indicate that the programmer is not sufficiently familiar with the background material listed in Section 4.1.

  The Designer's Notes are used by the designer to record informal remarks that will help the programmer, but which will not form part of the formal module documentation. Such remarks could include clarification, suggested methods, references, warnings, and so on.

- Understand Module Logic

  While the module logic will have been verified from a design point of view during the design phase, it is important for the programmer to spend some time understanding the logic and satisfying himself that it is correct.

- Request Revision of Specification Documents

  When the verification process is complete, any required changes to the specification must be authorized using the change control procedures defined in the Project Management Handbook.

Working Documents

- References (if any given in Implementation Notes)

Deliverables

- (Verified) General Module Specification

4.3 Perform Detailed Design of Module (5.B.2)

Objectives

- To expand the detail of the General Module Specification and the Designer's Notes in order to prepare for and facilitate the coding step.

Input

- General Module Specification
- Designer's Notes

Methods

It is usually not advisable to code directly from the General Module Specification. This specification is normally written at a relatively general level of detail and is not intended to form the structure of the actual code. In fact, programming language or other constraints may force the programmer to write code that is structurally quite different from the general specification. For this reason, a detailed module specification is developed at a relatively high level of detail to initiate a preliminary code structure.

Implementation Notes should be produced to take into account the refinement of the specification. The Detailed Module Specification and Implementation Notes should follow the same standards and conventions that are used for the inputs from the Design phase.

Working Documents

None

Deliverables

- Detailed Module Specification

4.4 Verify Detailed Design (5.B.3)

Objectives

- To verify that the Detailed Module Specification is complete.
- To ensure that the detailed design is logically correct.
- To revise the module documents, if necessary.

Inputs

- Detailed Module Specification
- Implementation Notes

Methods

- Summary of Steps

  - Verify Specification Completeness
  - Verify Module Logic
  - Revise Documents

- Verify Specification Completeness

  A careful verification that the detailed specification addresses all requirements greatly reduces problems in the coding and testing phases. Detailed verification procedures will be specified for each project.

  As a minimum, the programmer will be asked to carefully desk-check the specification. Where appropriate, a walkthrough will be performed with one or more additional members of the project team. A detailed description of the procedures to use for walkthroughs can be found in Appendix B.

- Verify Module Logic

  It is important to also verify that the module's logic is correct. This step is again done by desk-check, or walkthrough, depending on the situation.

It will usually be more efficient to perform the completeness and logic verification steps at one sitting or meeting, although they should still be treated as distinct steps.

Walkthrough decisions are formally recorded.

. Revise Documents

When the verification process is complete, detailed design specifications are modified as required.

Working Documents

. Walkthrough Records

Deliverables

. (Verified) Detailed Module Specification
. (Verified) Implementation Notes

4.5 Code Module (5.B.4)

Objectives

- To produce fully-documented source code for the module

Inputs

- Detailed Module Specification
- Implementation Notes

Methods

- Summary of General Principles
  - Write Clear and Simple Code
  - Follow Coding Standards, Guidelines, and Conventions
  - Anticipate Testing and Debugging

- Write Clear and Simple Code

  In the past, it has often been a software development priority to minimize hardware resource usage (processor cycles, memory, disk, etc.) through the use of complex coding structures. This frequently resulted in code that was difficult to understand and, therefore, difficult to maintain and low in reliability. However, with the cost of hardware rapidly decreasing relative to the cost of software, it is becoming increasingly important that programs be reliable and maintainable. As a result, characteristics such as clarity and simplicity are now generally considered to be more desirable than hardware usage considerations for most applications.

  A number of techniques and rules have been identified to help make code clear and simple, including:

  - Use structured coding techniques;
  - Don't write programs that modify their own code;
  - Avoid complicated arithmetic expressions, particularly where implicit type conversion is involved;
  - Where possible and practical, avoid negative Boolean logic;

- Avoid the use of constructs that "rename" the same area of memory (e.g., REDEFINES in COBOL, EQUIVALENCE in FORTRAN);

- Avoid jumping in and out of loops; and

- Format code for readability using spacing, indentation, and other such techniques, especially for complex statements and loops.

The inclusion of appropriate comments in the source code is another important aspect of good coding. The primary purpose of such comments is to facilitate the maintenance of programs. Therefore, the comments should be oriented towards describing what the program does and, where necessary, why the program does it that way. The comments should not simply restate the code.

Comments are an integral part of the program, and they must be included as the code is written.

Comments added during coding are more effective than comments added after coding has been completed.

- Follow Coding Standards, Guidelines, and Conventions.

There are two reasons for using coding standards, guidelines, and conventions:

1. For technical reasons, the use of various constructs and techniques may be encouraged, discouraged, restricted, or even prohibited. For example, constructs that often cause reliability problems (i.e. REDEFINES in COBOL) are often prohibited or severely restricted.

2. For consistency reasons, the use of various constructs and techniques may be standardized. For example, a FORTRAN coding standard might include a specific method for coding WHILE loops.

Standards, guidelines, and conventions are developed to help programmers produce code that is reliable, easy to test and debug, and easy to understand. The ISM guidelines are given in Appendix C.

AN EXAMPLE OF STRUCTURED CODE

```
PERFORM INITIALIZATION.
PERFORM UPDATE-MASTER
  UNTIL NO-MORE-TRANSACTIONS
    OR NO-MORE-MASTER-RECORDS,
IF     NO-MORE-TRANSACTIONS
  THEN PERFORM COPY-REMAINING-MASTER
         UNTIL NO-MORE-MASTER-RECORDS
ELSE   (NO MORE MASTER RECORDS)
  SO   PERFORM ADD-REST-OF-TRANSACTIONS
         UNTIL NO-MORE-TRANSACTIONS.
PERFORM TERMINATION.
STOP RUN.
.
.
UPDATE MASTER.
  IF   TRANS-ACCTNO IS GREATER THAN MAST-ACCT-NO
 THEN PERFORM WRITE-OUT-MASTER
      PERFORM GET-NEXT-MASTER-RECORD
  ELSE (TRANS EQUAL OR LESS THAN MAST)
    IF   TRANS-ACCT-NO IS EQUAL TO MAST-ACCT-NO
      THEN PERFORM SET-UP-NEW-MASTER
           PERFORM WRITE-OUT-MASTER
           PERFORM GET-NEXT-TRANSACTION
           PERFORM GET-NEXT-MASTER-RECORD
    ELSE   (TRANS LESS THAN MASTER)
.
.
.
```

. Anticipate Testing and Debugging

All programs must be thoroughly tested, and debugged. Therefore, it is important to design, specify, and code programs so that testing and debugging are made easier. Normally, a number of techniques for doing this are specified as part of the overall project test plan. The test plan will probably include some or all of the following techniques:

- parameter validation on subroutine entry;
- control and/or data tracing;
- error detection and processing for different severity levels;
- techniques to allow convenient usage of debugging utilities;
- I/O counts by type of access; or
- transaction counts by type.

A detailed knowledge of the testing methods outlined later in this document will help programmers become proficient in "defensive programming" techniques such as those listed above.

Working Documents

ISM Standard Practices Guides

Deliverables

. Detailed Module Documentation (includes Detailed Module Specification plus other module documentation generated during coding).

. Module Source Code

4.6 Verify Code (5.B.5)

Objectives

- to verify that the coding and detailed module documentation are complete
- to ensure that the code is logically correct
- to ensure that the code obeys all applicable standards, guidelines, and conventions
- to revise the code and module documents, if necessary

Inputs

- Detailed Module Documentation
- Module Source Code

Methods

- Summary of Steps

  - verify code and module documentation completeness
  - verify module logic
  - verify adherence to standards

- Verify Code and Module Documentation Completeness

  The completeness of the code and module documentation should be verified using the desk-check or walkthrough methods.

  Records of walkthroughs are to be maintained as a means for follow-up action.

- Verify Module Logic

  The module logic should be verified in an orderly manner. During a desk-check or walkthrough of actual code, there are many potential sources of error that should be checked. These can be broken down into the following groups:

  - Data Reference
  - Data Declaration
  - Computation
  - Comparison
  - Control Flow
  - Interfaces
  - Input/Output
  - Miscellaneous

The following lists provide questions that should be asked about each group. These questions should first be asked by the programmer, and then by the inspection team.

Data Reference

- Are variables initialized?
- Do subscripts exceed minimum/maximum ranges?
- Are integers used for subscripts?
- Do the record and structure attributes agree?
- Do structure definitions match across procedures?
- Are there "off-by-one" errors in indexing/subscripting?

Data Declaration

- Are all variables declared?
- Are the default attributes understood/correct?
- Are arrays/strings properly initialized?
- Are correct lengths, types and storage classes assigned?
- Is initialization correct with storage classes?
- Are there variables with similar names?

Computation

- Are the computations on numeric (arithmetic) variables?
- Are there mixed mode computations?
- Are there computations on different length variables?
- Is the target size less than the size of assigned value?

- Is there intermediate result overflow/underflow?

- Is there a division by zero (or variable set/computed to zero)?

- Is a variable's value outside a meaningful range?

- Are integer divisions correct?

- Is operator precedence understood (order in which computations take place, e.g. multiply/division before addition/subtraction)?

Comparison

- Are there inconsistent comparisons between variables?

- Are there mixed-mode comparisons?

- Are Boolean expressions correct?

- Are comparison relationships correct?

- Is operator precedence understood (order in which comparisons take place - AND before OR)?

- Is the computer evaluation of Boolean expressions understood?

Control Flow

- Are multiway branches exceeded (i.e. are there more values than branches for a GO TO DEPENDING ON)?

- Does each loop always terminate?

- Does the program terminate?

- Are any loops by-passed because of entry conditions?

- Are possible loop fallthroughs correct?

- Are there any "off-by-one" iteration errors?

- Do the "DO/END" statements match?

- Are there any non-exhaustive decisions?

Interfaces

- Are the number and attributes transmitted to this module received correctly?
- Are the number and attributes of parameters passed to called modules sent correctly?
- Are there any references to parameters that are not associated with the current point of entry?
- Are any input only arguments changed?
- Are global variable definitions consistent across modules?
- Are constants passed as arguments?

Input/Output

- Are file attributes correct?
- Are open statements correct?
- Do the format specifications match I/O statements?
- Does the buffer size match the record size?
- Are any files opened before they are used?
- Are end-of-file conditions handled?
- Are I/O errors handled?
- Is there more than one record area per file?
- Are files closed when no longer needed?

Miscellaneous

- Are there any unreferenced variables in the cross reference listing?
- Is the attribute list as expected?
- Are there any warning/informational messages?
- Is the input checked for validity?
- Are there any missing functions?

. Verify Adherence to Standards

  A team member appointed as "inspector" ensures that the programs adhere to standards. Any deviation detected should be amended by the program's author.

Working Documents

. Walkthrough Notes

Deliverables

. (Verified) Detailed Module Documentation

. (Verified) Module Source Code

4.7 Test Module (5.B.6)

Objectives

- To test the program to ensure that the module operates in accordance with all applicable specifications and requirements.

- To debug the program by finding and correcting errors that are detected in the test step.

Inputs

- Detailed Module Documentation

- Module Source Code

Methods

- Summary of Steps

  - Formulate Test Plan
  - Create Test Data
  - Determine Expected Results
  - Run Tests
  - Debug Module

- Formulate Test Plan

  Before testing begins for a single module or a group of modules, a detailed test plan must be formulated. This plan will be simple for a single module, but may be quite complex if a group of modules is involved. Usually, programmers prepare test plans for their own modules. However, on some projects, particularly large ones, it may be desirable to have someone other than the programmer plan and perform the testing. In any case, the test plan should be subjected to a walkthrough.

  One key issue that must be considered when formulating a test plan for a group of modules is whether to use a top-down, bottom-up, or hybrid approach. Individuals preparing test plans should be aware of the tradeoffs involved. Readers unfamiliar with these tradeoffs should read the sections related to software testing in References 1, 2, and 3 of Appendix A.

. Create Test Data

A set of test data must be created for each module. The first action in this process is to create data to ensure that all possible paths through the module are tested. This document describes a method for doing this, and consists of the following steps:

1. prepare a "logic path diagram".

2. use the logic path diagram to prepare a "case diagram".

3. use the case diagram to prepare the actual test data.

This method is a straightforward way (in a structured environment) to ensure that all paths in a module are tested. While additional test cases are needed to fully test the module, this method is a convenient way to handle a significant amount of test case creation.

A sample problem will be used to show how the test data creation method works. Assume that the following program is to be tested:

> TRIANGLE PROBLEM. The program reads three integer values that represent the lengths of the sides of a triangle. The program then prints a message that states whether the triangle is scalene, isosceles, or equilateral. The program should also detect the error condition that occurs when a triangle cannot exist with the input side lengths.

The (simplified) pseudocode for this program might be:

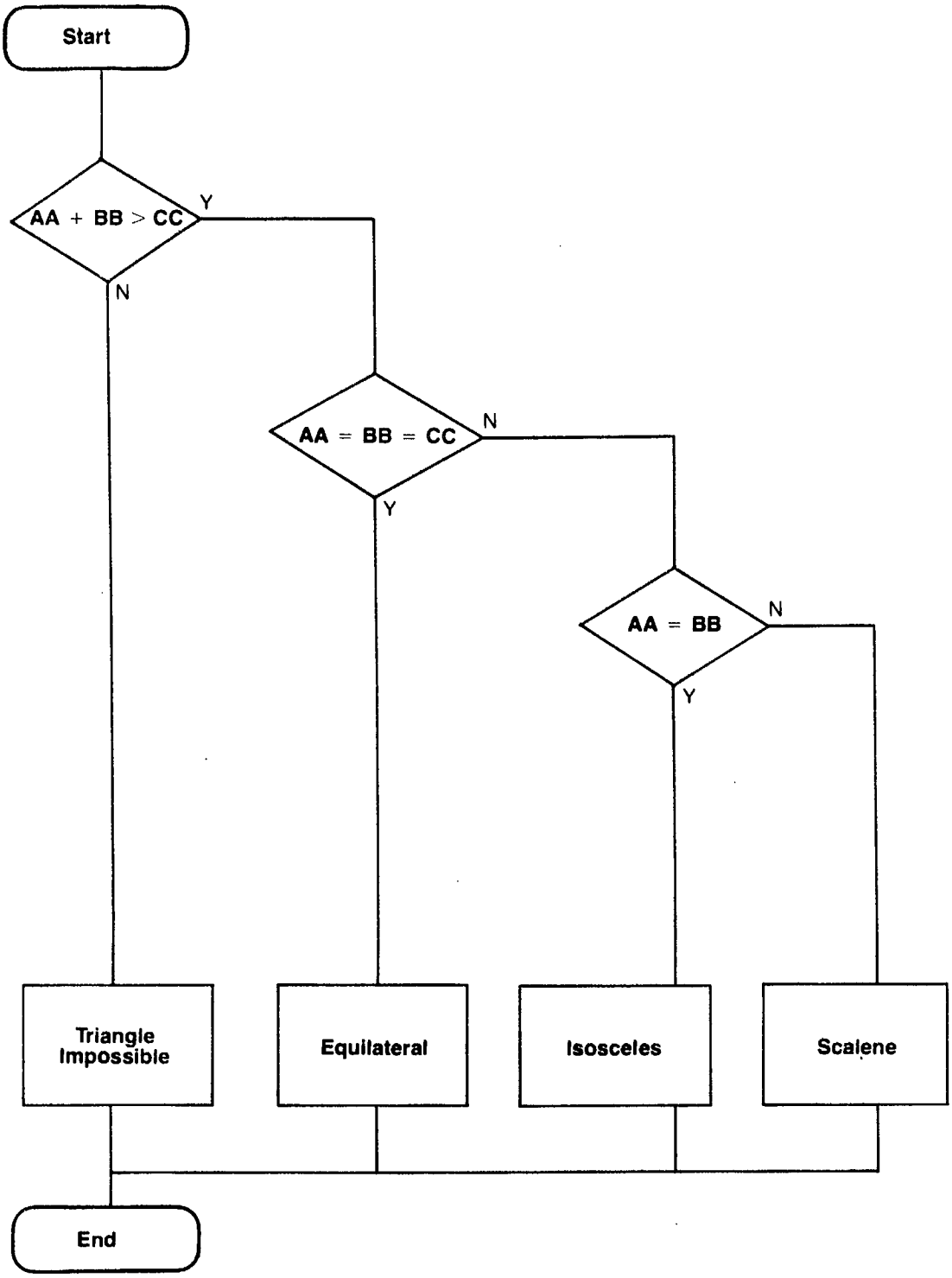
```
TRIANGLE-PROBLEM:
READ VALUES A, B, AND C
SORT VALUES A, B, AND C (ASCENDING)
  TO GIVE AA, BB, AND CC
IF AA + BB  CC
:  IF AA = BB = CC
:  :  PRINT "EQUILATERAL TRIANGLE"
:  ELSE
:  :  IF AA = BB
:  :  :  PRINT "ISOSCELES TRIANGLE"
:  :  ELSE
:  :  :  PRINT "SCALENE TRIANGLE"
:  :  ENDIF
:  ENDIF
ELSE
:  PRINT "TRIANGLE IMPOSSIBLE"
ENDIF
```

(Note that this algorithm has a bug in it; it will not properly detect certain kinds of isosceles triangles. This bug emphasizes the fact that the test data created using this method must be supplemented with test cases designed to detect logic errors).

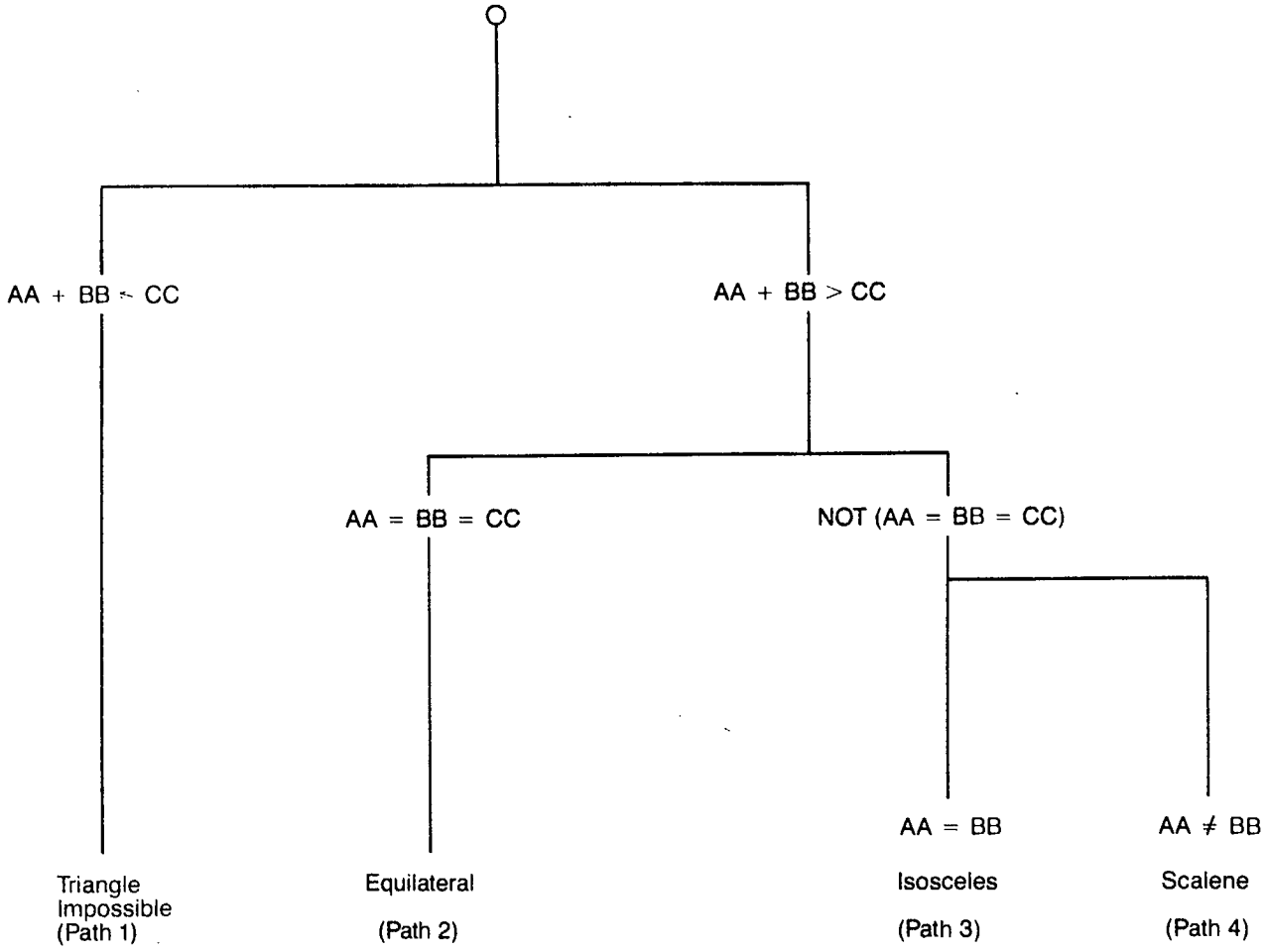
The first step is to prepare a logic path diagram. These diagrams are similar to flowcharts, except that only branches are represented. Code that does not result in a branch is ignored or grouped together. If the logic path diagram is very complex and difficult to understand, it is likely that the module itself is too complex and should be split into two or more new modules. The logic path diagram for the Triangle Problems is shown in Figure 4.7.A.

The second step is to prepare a case diagram, which is used to determine all the paths that can be taken within the module. The case diagram is prepared by examining the logic path diagram and producing a decision tree, where the decision nodes correspond to the decision points in the logic path diagram. Each terminal node in this tree represents a path within the module. The case diagram for the Triangle Problem is shown in figure 4.7.B. The result of taking a particular path must be a distinguishable action, such as writing a record or printing a line. If such a distinguishable action does not exist for a path, then temporary code must be inserted to supply an action (e.g. print a message).

The third and final step is to prepare the actual test data. Each terminal node in the case diagram tree results in a test case. The actual test data for the test case is constructed by following the path from the terminal node back to the rest of the tree; the test data must be consistent with all of the conditions on the branch of this path.



Logic Path Diagram For Triangle Problem
Figure 4.7.A



Case Diagram For Triangle Problem
Figure 4.7.B

The following table shows sample test cases developed from the tree shown in Figure 4.7.B.

PATHS	TEST CASE VALUES		
	AA	BB	CC
PATH 1 (Triangle Impossible) AA + BB < CC	1	2	4
PATH 2 (Equilateral) AA = BB = CC AA + BB > CC	2	2	2
PATH 3 (Isosceles) AA = BB NOT (AA = BB = CC) AA + BB > CC	2	2	3
PATH 4 (Scalene) AA = BB NOT (AA = BB = CC) AA + BB > CC	2	3	4

In addition to the test data derived from the case diagram, it is important to introduce test data related to such things as "extreme conditions". Some examples of common extreme conditions are:

- zero or negative values;
- zero or one transactions;
- empty files;
- missing files (file name not resolved);
- multiple updates of one Master;
- full, empty or missing tables;
- window headings;
- table entries missing;
- subscripts out of bounds;
- sequence error; or
- no remaining file space left.

Program reliability problems associated with extreme conditions are very common and often costly to detect and correct.

. Determine Expected Results

For each test case, the expected result(s) of that case should be recorded.

. Run Tests

Once the test data is prepared, the actual running of the tests should be a relatively straightforward procedure. For some projects, a standard test driver will be available for running the tests. This may be a generalized driver that provides sophisticated and very convenient test facilities, or it may simply be a "skeleton" program that must be edited to produce a customized driver program. In some cases, a special driver program must be developed, while sometimes the data must be entered manually (e.g. on-line). In any case, full test documentation should be maintained for each module. This documentation will consist of the driver program (or equivalent), the test data, and the test results. The exact nature of this documentation will depend on the project environment and will be defined in detail early in the project.

- Debug Module

If errors are detected during the test runs, then the errors must be investigated and corrected. There are many approaches and techniques that can be used to attempt to optimize debugging. Some of these may be standardized for a project (e.g. dynamic subroutine entry/exit trace) and therefore will be mandatory for that project. Others will be selected by the tester depending on circumstances.

Readers should be familiar with the debugging concepts and techniques found in References 2 and 3 of Appendix A. These techniques can be summarized as follows:

1. Write small modules. It has been found that the number and subtelty of bugs increases exponentially with the number of instructions in the module. By keeping modules small, it is easier to find the bugs.

2. Try to determine the nature of the error. It may be possible to isolate the general cause of the error, (e.g., hardware, compiler, operating system, application program) before narrowing the search.

3. See if the bug is repeatable, otherwise the bug may not be under the control of the programmer and attention should be focused elsewhere.

4. Keep records of past mistakes. When a new bug is found, a search through these records may show a similar bug that occured before and the method used to correct it.

5. Be thorough, methodical and logical in the search.

6. Don't jump to conclusions. Try to explicitly identify the possible causes of the error and concentrate first on the cause that seems most probable. Continually update the list of possible errors and their probabilities as more information is obtained.

7. Don't take anything for granted. The cause of the error may not be in the module under investigation, it could be in other modules that link to this, in the software, and even occasionally in the hardware.

8. Use test cases to confirm or reject suspicions. The time and effort used to run a small test will give far better quantifiable results than just speculation (see points 6 and 7 above).

9. Seek help from colleagues. After working on the problem for a reasonable time without any success, the programmer should obtain a second opinion. Because the programmer is so deeply involved with the details, he might be missing an obvious fact.

Working Documents

None

Deliverables

. Detailed Module Documentation

. (Tested) Module Source Code

. (Tested) Module Object Code

. Test Documentation
  - Module Test Plan, including inspection notes
  - Test Data Creation (logic path diagram, case diagram, test data, expected results)
  - Test Results (actual results, debugging notes)

4.8 Verify Completeness of Module Development (5.B.7)

Objectives

- To ensure that module development is complete.
- To initiate the "release" of the module into more widespread circulation (e.g. system testing).

Inputs

- Detailed Module Documentation
- Module Source Code
- Module Object Code
- Test Documentation
- Working Documents (from previous steps)
- Historical Documents (from previous steps)

Methods

- Verify Completeness of Module Development

  For each project, the module completion procedures will be defined as part of project policy, and a "completion checklist" will be prepared. The programmer should review the checklist to confirm that all required documents exist and are filled properly. The documents to be filled normally include working documents and historical documents as well as deliverables.

- Initiate Module Release

  All projects will have a formal procedure for releasing a module. It may for example, involve integrating the module into a higher-level program or placing the module in a library. In any case, this step must be initiated by the programmer.

Working Documents

- Completion Checklist

Deliverables

- (Verified) Detailed Module Documentation
- (Verified) Module Source Code
- (Verified) Module Object Code
- (Verified) Test Documentation
- (Verified) Working Documents
- (Verified) Historical Documents

APPENDIX A

References

REFERENCES

1. Software Reliability by Glenford J. Myers
360 pages
This book was written to describe solutions to the problems of unreliable software. Every aspect of software production is examined. Highly recommended for all programmers and designers.

2. Techniques of Program Structure and Design by Edward Yourdon
An easy-to-read book on program design philosophies and methods. Coverage includes chapters on the characteristics of a "good" computer program, top-down program design, modular programming, structured programming, programming style -- simplicity and clarity, anti-bugging, program testing methods, debugging concepts and techniques, and four major example problems.

3. The Art of Software Testing by Glenford J. Myers
177 pages
The book provides practical, rather than theoretical, discussion of the purpose and nature of software testing. It is highly recommended.

4. Structured Systems Analysis, Tools and Techniques by C. Gane and T. Sarson
373 pages
The book describes techniques for the analysis and definition of a new system using such tools as logical data flow diagrams, data dictionaries and other structured design techniques.

5. Composite/Structured Design by Glenford J. Myers
This book presents a program design methodology with the goal of producing programs of higher reliability and extensibility.

6. A Simplified Guide to Structured COBOL Programming by Daniel D. McCracken and Brian Randall
389 pages

7. High Level COBOL Programming by Gerald M. Weinberg
Stephen E. Wright, Richard Kauffman and Martin A. Goetz
252 pages

REFERENCES

8. Structured Programming in COBOL edited by H.P. Stevenson
Papers from an ACM symposium that cover the implementation of strucuted programming in COBOL.

9. The Structured Programming Cookbook by Paul Noll
220 pages
This book is written to improve the productivity of the experienced COBOL programmer.

APPENDIX B

Walkthrough

WALKTHROUGH

The walkthrough/inspection concept came from IBM's programming teams. It uses the theory that the Programmer is a part of the complete team and that the team (not just the Programmer) is responsible for each program. This is commonly known as egoless programming.

The walkthrough concept is basically an extension of the desk check process. During desk checking, the programmer examines his code to discover errors. During a walkthrough members of the team inspect the code in a systematic manner to find any errors.

Although the walkthrough concept was developed for inspecting programming output, and this description is in that context, it is equally applicable to the products of analysis, design and testing.

The objective of this process is to find errors in logic, in specifications, etc. An inspection also looks for errors in style such as readibility, efficiency, unreasonable specifications, etc. the purpose of the inspection is not to find fault with the originator of the product being inspected but to improve upon that product.

Also Refer to Datamation Oct. 1977.

Inspecting Software Design and Code
By M.E. Fagan.

Below is an outline of an inspection technique used on one project.

Inspection team consists of:

- Chairman, who coordinates and schedules the meetings, chairs the inspection, notes all errors, circulates the inspection report and follows up on the rework.
- Document creator, the person who has created the document, whether it be the program specification, design or code. It is his responsibility to have all documents circulated to the other members at least 24 hours before the inspection.

- Implementors, those who will be taking over responsibility for the document (e.g., designers who will receive the specifications, programmers who will receive the program design, etc.). There will normally be one or two people in this category.

Inspection process consists of:

- Preparation and distribution of the document to all members of the inspection team prior to the meeting by the document creator.

- Preparation by all of the inspection team members which involves going over the document in some depth before the meeting.

- Inspection of the document by the whole team in the meeting. As the objective is to find errors, discussion continues only until the point where an error is recognized. The aim of the inspection is only to find errors, so often the chairman must be firm in limiting discussion. The error is then noted by the chairman. At the end, the team decides if the document passes the inspection and if not, a date is set for further inspection.

- Circulation of the inspection report by the chairman within 24 hours of the conclusion of the inspection meeting.

- Rework by the document creator to correct the errors.

- Follow-up. If the number of errors is small, than the Chairman is responsible for verifying that all errors are redressed. If there are a large number of errors, the inspection cycle is repeated.

During a walkthrough of actual code, there are major areas where problems occur. These are:

DATA REFERENCE
DATA DECLARATION
COMPUTATION
COMPARISON
CONTROL FLOW
INTERFACES
INPUT/OUT