An object-oriented testing approach based on a rigorous model of claimed functionality

A thesis by : Noel M O' Connor
Bachelor of Science in Computer Applications

Supervisors : Mr. R Verbruggen, MMgtSc, MMI(KULeuven)
Professor T Moynihan, MSc, PhD, FICS, FSS

Submitted to : Dublin City University
School of Computer Applications
for the degree of
Master of Science
August 1990

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other University or Institute of learning.

Abstract

An object-oriented testing approach based on a rigorous model of claimed functionality.

Author : Noel M O' Connor

Testing aims to enhance the quality of the software under test. This is achieved through finding and removing errors which, if they were present, would detract from the operational efficiency or accuracy of the product and therefore detract from the product's quality.
Black box testing is based on knowledge of the specified functionality of a product, whereas white box testing makes use of knowledge of the program code. Regardless of the testing technique employed, the main objective is to derive a set of test cases that will uncover defects in the code.
In this thesis my testing approach focuses on an object-oriented model of a software system. The model is constructed through "reverse engineering" a rigorous description of claimed functionality from the user documentation of the product. The model is represented in the form of objects, attributes and operations. The operations are perceived by the user as being the functionality of the system.
My approach uses an object-oriented black box testing technique to exercise the functionality of the model. This is achieved through deriving test cases from the model (which is represented at a different level of abstraction using graph theory notation).
The test cases consist of the valid sequences of operations that are allowed, along with the expected output for each such sequence. My objective is to completely exercise the functionality of the model using a minimum set of valid test cases.

To my parents.

Acknowledgements

Renaat, eternally grateful. Thanks for all the time you invested with me while working on this thesis. I couldn't have done it without you. Here's to the VERBRUGGEN development method!

Tony, thanks for all the help on the SCOPE project , your inspiration, and for your credit card when we most needed it (Paris 89).

Colophon : This thesis was typeset in Times font using Microsoft Word on a Macintosh IIx, and printed on a PSJet laser.

Contents

Chapter 5
Object-oriented black box testing

Chapter 1

Introduction

1.1 Introduction

Businesses and governments are increasingly investing money into software systems whose successful operation is critical to their bottom line. Producing reliable systems requires the use of several techniques - not just review, not just inspection, and not just one type of testing. It needs a well thought-out, comprehensive application of several techniques throughout the development life cycle. These techniques fall under the umbrella of "software verification and validation". My main interest focuses on the area of software testing.

1.2 Testing in perspective

Testing provides a complementary mechanism to explore a systems' operational behaviour. Few people would want to fly on a theoretically safe airplane that had not been flown before. Few would trust a program that had never been run. The sinking of the HMS Titanic gave us a graphic and tragic example of what can happen when an inaccurate model is used. Over the past forty years, as the use of digital computers has increased, there has been a corresponding increase in the number of software failures *[Gel 88]*. During this period, the idea behind making testing a cost-effective process has grown as well. Since testing is as old as coding, most people involved in the software industry have their own mental model of testing. As a result, there are many unrecognised differences in the various mental models, and the result has produced a lot of confusion amongst customers, managers, analysts, programmers and testers.

1.3 A History of Errors

Most computer malfunctions are caused by software failure of one kind or another. Aerospace, military and space travel provide such examples. One such example from military is when USAF pilots were testing the F-16 fighter aircraft, the first thing they did was to tell the onboard computer to raise the landing gear while the plane was still on the runway. Another disaster was when an F-16 plunged into the Gulf of Mexico because the onboard computer was not programmed to cope with very low flying. Medical computer glitches have cost a lot of lives. In 1980, a man undergoing microwave arthritis therapy was killed when the therapy reprogrammed his pace-maker, while faulty

software in an insulin pump, caused insulin to be delivered at the wrong rates and a medical system was recalled after it was discovered that it had mixed up patients' names and records.
In fact computer 'foul ups' have a long and less than distinguished history. In 1960 computers of the BMEWS(Ballistic Military Early Warning System) at Thule initiated a nuclear alert after the rise of the moon above the horizon was interpreted as a nuclear attack. In another case, a flock of geese was thought to be a group of inbound nuclear warheads *[Per 90]*.
While refrigerators, cars, washing machines and practically every other consumer goods are sold with guarantee of quality and workmanship, the same cannot be said of software.

1.4 Attaining Software Quality

The requirement for high quality of software is becoming increasingly acute, and recognition of this need is becoming more widespread. While techniques such as superior development methodologies can do a lot to enhance software quality, there still exists a consensus that software quality can neither be obtained nor convincingly demonstrated without significant testing activities *[Ric 89]*. Testing aims to determine and assure that software products are of "high quality". Testing researchers all agree that there is no absolute, fixed notion of what "software quality" means. Instead, there is an agreement that software should satisfy a variety of qualities such as robustness, functional correctness, efficiency and reliability. Different products emphasise different qualities and accordingly have different testing requirements.

1.5 Thesis overview

In this thesis I describe my method for object-oriented black box testing based on a rigorous model of claimed functionality. I worked on the European project SCOPE in which DCU are partners. The SCOPE project centres around the idea of software certification and the implications of providing a certificate to a software product.

Chapter two looks at the SCOPE project and software certification. It examines the different views of certification and the problems associated with such a venture in the context of software.

Chapter three describes the DCU case study on which I based my research. The case study involved taking a piece of 'live' software and documentation, deriving the quality attributes which needed to be assessed, in the form of a 'certification specification', and to make subsequent 'judgements' on these attributes. The attributes assessed were correctness, maintainability, reliability, security, performance, usability, quality of documentation and 'ease of housekeeping'. They were individually assessed using methods such as 'reverse engineering', checklists and testing. The 'judgements' reached on the attributes included in the 'certification specification' are documented in this chapter. Chapter three also looks at how a rigorous model of claimed functionality was "reverse engineered" from user documentation.

In Chapter four I put the testing area in perspective. There are various testing techniques varying from black box to white box approaches. They all share the same goal - to highlight the presence of errors in the software. The terminology associated with testing, such as unit testing, top-down testing, test cases etc., is also explained in this chapter.

In Chapter five I present my own testing method. The method is object-oriented based, using an object-oriented view of the system to drive a black box testing technique. My testing method uses ideas from the object-oriented rigorous model developed in Chapter three as well as incorporating some elements of graph theory. I was able to apply the method as a way towards generating test cases which were in the form of allowable sequence(s) of operations comparing actual output with expected output.

In the final chapter, I broadly assess my method showing how it can be integrated and expanded to incorporate existing testing techniques, as well as how it could be formalised.

Chapter 2

SCOPE project and Software Certification

2.1 Introduction

In this chapter I will examine the European project SCOPE and explain the terminology associated with software certification.

2.2 SCOPE - **S**oftware **C**ertificati**O**n **P**rogramme in **E**urope - is an European project partially funded by the ESPRIT II (**E**uropean **S**trategic **P**rogram for **R**esearch and Development in **I**nformation **T**echnology) initiative. There are eleven partners involved in the project including Dublin City University. The other partners are universities, standards/certification bodies and software developers from other European countries. The full list of partners involved are Verilog(VLG) - France, Etnoteam (ETN) - Italy, GMD,GRS - (Germany), ERIA (ERI) - Spain, ElectronikCentralen(ECT), - Denmark, City University(CTU), United Kingdom Atomic Energy Association(UKA), Strathclyde University(STR), Glasgow College(GCT) - UK and Dublin City University(DCU) - Ireland.The project is organised into two sequential work packages; the first one lasting 22 months (399 man months) and the second 42 months (912 man months) *[SCO[1] 89]*.

2.2.1 SCOPE in general

SCOPE has the job of defining the overall European certification scheme and identifying and developing software tools to support test laboratories. The EC wishes to see a European-wide system in place by the mid-90s. Under this scheme, each country would have one or more software test laboratories. Typically, a software developer would submit his product to a test lab and would specify the characteristics of the product to be certified e.g. conforming to specification, portability, usability, reliability. If the test lab was happy with the product, it would issue a 'quality seal' endorsing the particular product attributes inspected. The possession of that 'seal' would help the developer market his product across Europe.

2.2.2 SCOPE objectives

The three major objectives of the SCOPE project are :

(1) To clarify the customer/supplier relationships by defining procedures that would enable the granting of a 'quality seal' to the software when it complies with a certain set of pre-specified attributes.

(2) To develop new efficient and cost effective certification technologies for the granting of this 'seal'.

(3) To promote the use of modern software engineering technology to be used during the development of the software and contributing to the 'seal' delivery.

2.3 SCOPE breakdown

The first SCOPE work package is broken down into four tasks.

2.3.1 Technology

This task is concerned with :
identifying the relevant state of the art on software certification, agreeing on definitions and terminology, determining what software attributes can and should be certified;

defining a prototype certification programme that will be explored using the first set of six case studies, with the primary aim of deriving a set of pragmatic certification procedures and acceptance criteria;

establishing a relationship with standards organisations in order to prepare the implementation of the proposed certification technology as a basis for newer standards and to survey developments at an international level;

establishing and defining the legal issues and requirements which may affect software certification;

evaluating the data collected by the case studies and determining the applicability of the first set of certification procedures;

2.3.2 Database

The database task has four main aims :

(1) to specify the database service.

(2) to develop the database and analysis system.

(3) to collect data and to administer the database.

(4) to develop tools for use in data collection and to provide a tool based analysis service.

2.3.3 Case studies

There are many problems to be solved before a certification scheme for software products can be proposed to the EC. For example, what attributes of software are measurable in a reasonably objective, easy to produce way ? How should the measurements be made, and at what probable cost ? What form should a 'quality seal' take ? These problems can only be solved by trying out ideas on real software products, and seeing what works. Hence the need for case studies in the project. Their were six case studies in the first phase of SCOPE, each conducted by a single partner as given in **Figure 2.1** A further set are planned for the second phase. The case studies were selected in order to be representative of confidentiality conditions of third party assessment. One of the SCOPE partners ETNOTEAM *[ETN 89]*, proposed a three-tier product classification scheme covering risk class, domain and application area.

Partner	Case Study	Risk Class	Domain	Application
DCU	TANGO	limited economic	private	Service control
ETN systems	VICTOR	limited economic	private	Information
GRS	WHISKY	safety critical	private	Service control
GCT systems	X-RAYS	low economic	private	Information
STR	YANKEE	low economic	business	Service control
VLG	ZULU	low economic	business	Utility

Figure 2.1

The case studies were intended to demonstrate the feasibility of software certification on a limited scale by defining a prototype certification programme and they were also used to represent systems thought to warrant certification.

Each partner used a different approach in dealing with their case study, although the overall emphasis was predominantly on static analysis, data flow analysis and statistical validation analysis of metrics data. The DCU case study TANGO will be described in Chapter three.

2.3.4 Management

From a managerial viewpoint SCOPE has had to overcome several difficulties from a 'standard' ESPRIT project, such as the number of partners(eleven), and the number of people involved, along with the nature of the work which is mostly at the conceptual level as opposed to the tool development level. Management of the project was handled by Verilog and they had to ensure complete integration among all partners.

2.4 Certification Framework

During phase one of SCOPE, a certification framework was developed as shown in **Figure 2.2**. The framework was designed to be flexible, evolutionary and capable of rapid responses to change. In time the framework should be capable of dealing with any software product. The framework is based upon a **Certification model** and a **Certification method** which will allow a software product to be awarded a certificate, if after independent evaluation it is shown that the product complies with all necessary requirements, results and standards.

The certificate will be graded into a number of levels of increasing stringency. This will ensure the whole spectrum of software will be covered, ranging from simple non-critical application through to complex high safety critical applications.

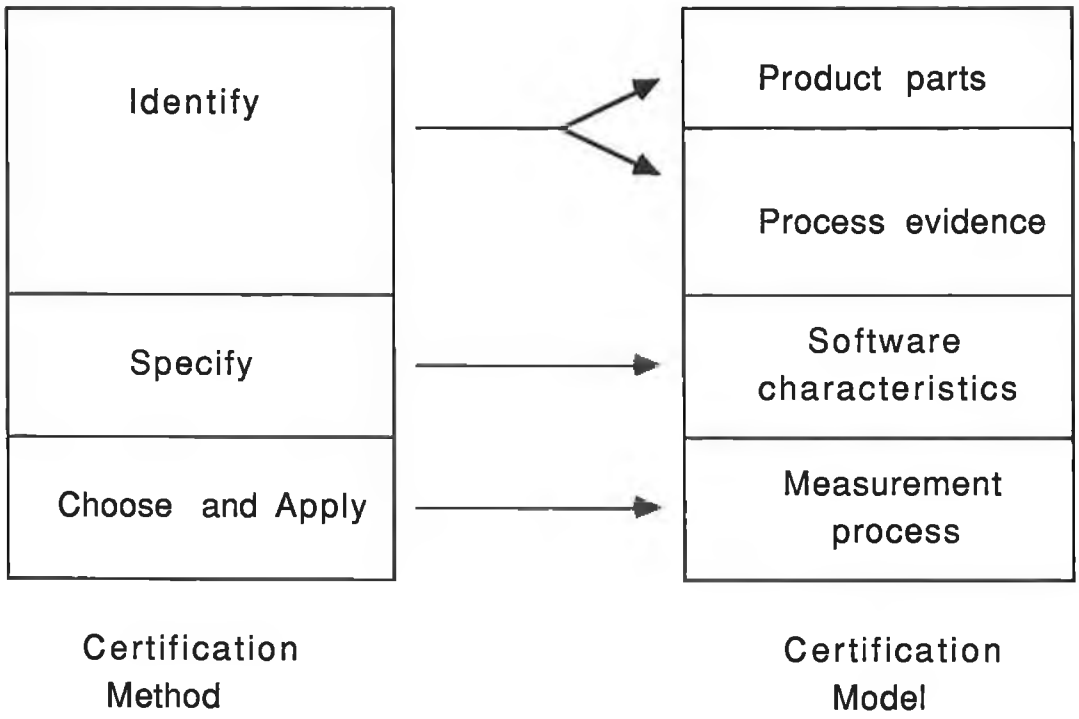


Figure 2.2 : SCOPE Certification Framework

2.4.1 The Certification Model and Certification Method

The certification model is comprised of four sub-models that help define the issues introduced by the certification method. It consists of the following

***(1)*Software product model**

***(2)* Software development process model**

***(3)* Software characteristics model**

***(4)* Measurement model**

These models will be further refined in the second phase of SCOPE.

(1) The **software product model** consists of a list of product parts which are precisely defined. The objective of this model is to only identify those elements in a product which are useful to the certification process.

(2) The **software development process model** identifies items of process evidence that may be useful to facilitate product measurements. In one sense this development effort attempts to reconcile the two ideas of "process" certification and "product" certification.

(3) The **software characteristics model** is the kernel of the certification model. It defines what characteristics of a software product can be assessed. Presently, this model consists of definitions of "correctness" and "reliability" and refinements to the notion of "workmanship". The major issue at stake here is to define the characteristics in an unambiguous way.

(4) The **measurement model** is more complex. The problem lies in dealing with the complexity of the state of the art measurement and assessment techniques. There are numerous metrics proposed and their relevant applications vary significantly. The measurement model uses a modular approach, deciding on a small set of metrics and assessment techniques, that can be specialised, and combined in to what is termed an "assessment brick" *[SCO[1] 90]*. This "brick" acts as a single assessment technique and gathers information pertaining to :

- software characteristics
- applicable tools
- the predicted cost of its application
- conditions for its application

2.5 Knowledge flow resulting from SCOPE

The resulting information flow from the project is shown in **Figure 2.3**. The main

streams of technology and information transfer are :

***(1)* Direct technology transfer from the SCOPE community to :**

-software engineering specialists developing software certification tools for use by the test labs

-software engineering specialists developing new tools for use by the software producers for enhanced quality and contributing to the delivery of the seal

***(2)* Secondary technology transfer from :**

-the developers of new technology to the test labs and software producers

-the test labs to the software producers for enhanced development quality

***(3)* feedback information transfer to the SCOPE community from :**

-the test laboratories

-the software producers

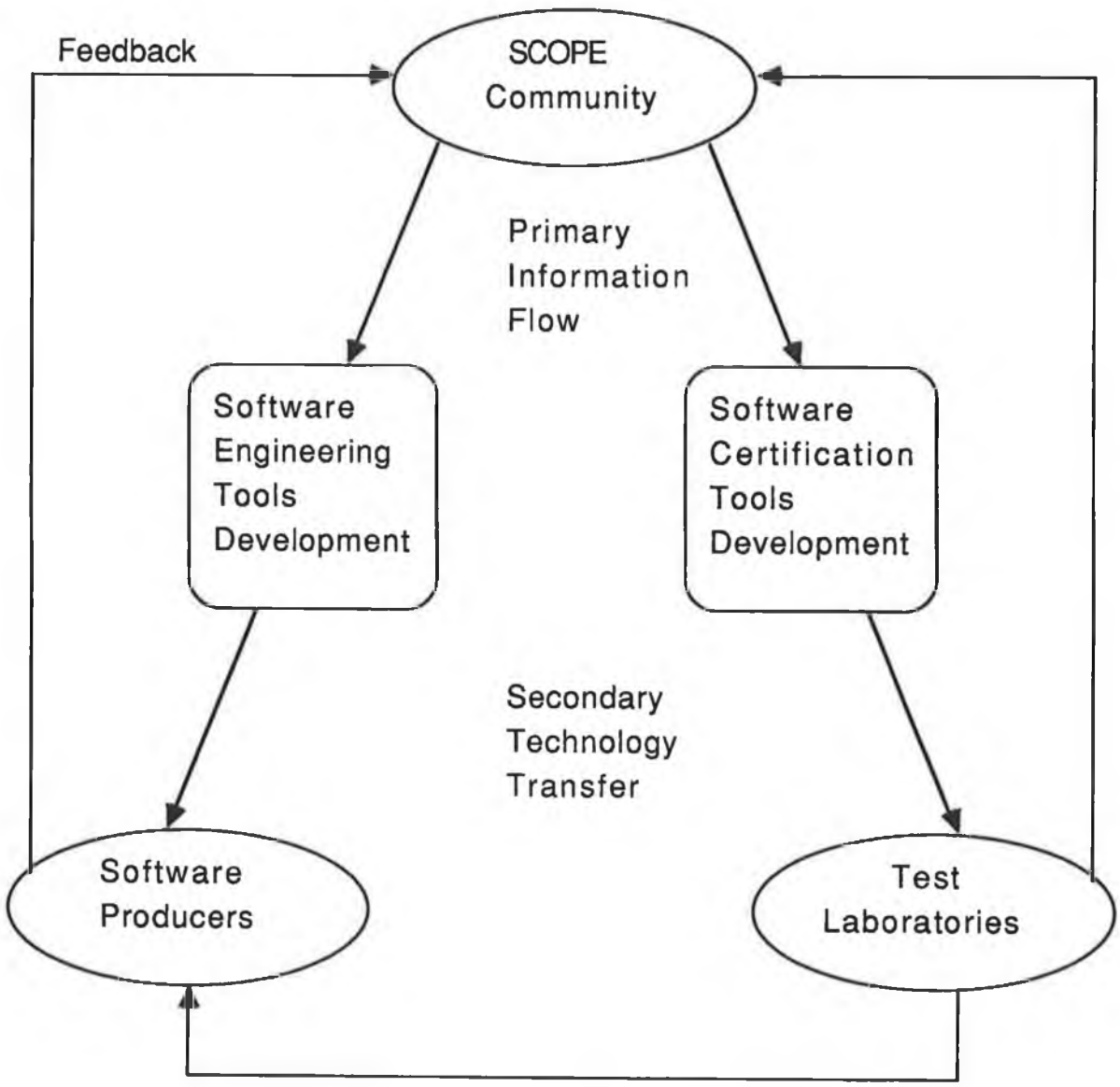


Figure 2.3 : Information and Technology Transfer

2.6 Definition of Certification

The process of confirming that a system, software subsystem, or computer program is capable of satisfying its specified requirements in an operational environment *[IEE 89]*.

2.6.1 Why Certification ?

The process of certification, (i.e. awarding an independent certification formally attesting fulfilment of conditions and requirements) has been established for many years. Contemporary examples range from, "seals of approval" for domestic goods to "certificates of airworthiness" for aircrafts. A seal or certificate is seen as giving a product a cachet of superiority, as something distinguishing it from its competitors. In other cases, a certificate showing conformity with requirements and regulations is a statutory requirement before a product (such as an aircraft or ship) can be put into service.

Most examples of certified products that come to mind are hardware, physical objects that can be observed, analysed and measured. A software program poses unique problems when one questions how it may be certified in the same sense as hardware is certified. It is often impossible to measure all the characteristics, and behaviour modes of a software product. The fact that most software is only licensed for use, and not purchased by the user helps to further confuse the issue.

2.6.2 Purpose of Certification

The different views of the service given by a software product are :

- expected service
- specified service
- actual service

The user need is basically that the product (actual service) must agree with his/her expectations. i.e. there must be an absolute agreement between expected and actual service.

2.6.3 Stages of Certification

There are suggested ten distinct stages in the certification of a software product. These are:

(1) Submission of the product to the Certification agency
(2) Agreeing on the certification requirement
(3) Agreeing on an initial cost estimate
(4) Analysing the product
(5) Producing a certification specification
(6) Relating the specification to tasks, techniques and tools
(7) Producing a certification plan
(8) Costing the certification plan
(9) Implementing the certification plan
(10) Producing a certification report

As the user expectations are rather subjective and are often related to non-expressed characteristics, the certification process cannot base itself simply upon this agreement of actual service versus expected service *[SCO^2 89]*.The process should involve the correspondence between the specified and actual service as given in **Figure 2.4** below.

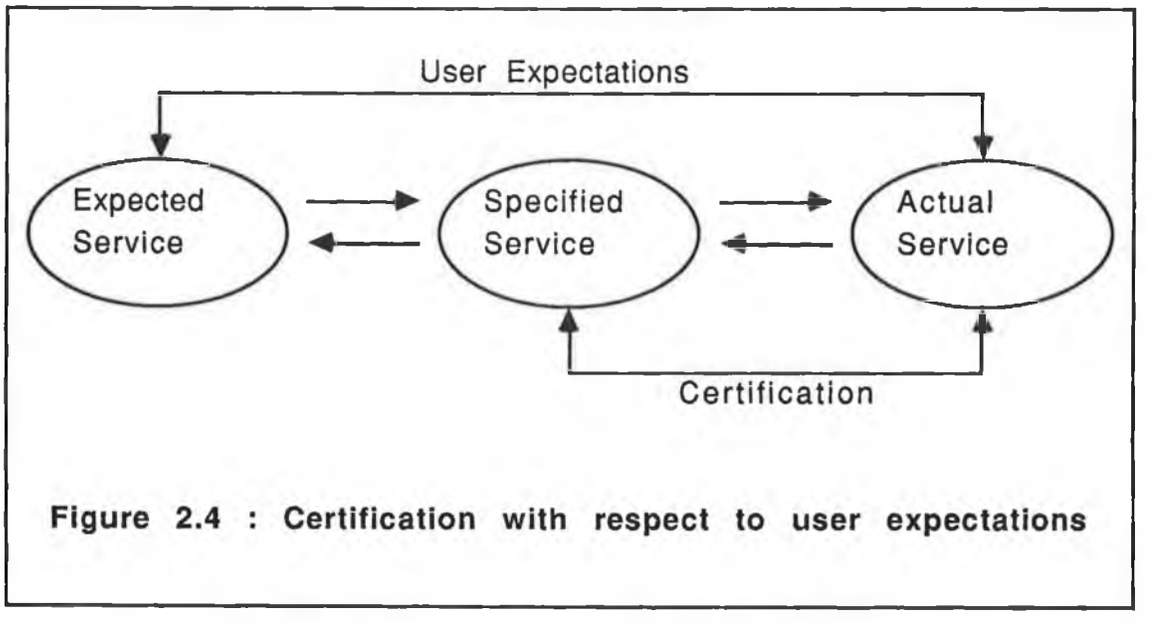


Figure 2.4 : Certification with respect to user expectations

2.7 Understanding Certification

The obvious perception of certification is that it gives some form of guarantee. At least it is assumed that the risks of failure are reduced where certification involves some form of independent assessment. Hardware can be relatively easily defined and measured, while software is not bounded by any physical consideration at all. It is when software is combined with hardware, in monitoring or controlling a physical process, that the inherent dangers of software failure start to become apparent.

2.8 Perceptions of Certification

Certification is seen by different people or organisations in different ways. Certification will always be given on the basis of closely defined parameters, derived from particular needs and for particular applications. These limitations need to be fully understood by everyone concerned such as users, customers, manufacturers, assessors, legal representatives, and the general public. Without this understanding, the role of certification will be misinterpreted, and this will be swiftly followed by the process becoming discredited.

2.8.1 The SCOPE view

Considerable progress can be made towards a framework of software certification which will, at least, provide economic levels of assurance commensurate with all the risks involved in the use of the software in any particular situation.The most appropriate state-of-the-art tools and methods to attest conformity of the software product with specific design requirements and attributes can be used.
Where safety is not an issue, the certification process may provide no more than a "seal of quality". This would give assurance to the user that a well defined development methodology had been used; the product had been adequately tested; the product would do what it was supposed to do under defined conditions of use, and would be maintained if things went wrong.

2.8.2 The user/customer view

The user/customer view of certification will be that of a guarantee - first, that the software will perform as described and second, that it will not fail in normal use. A customer who buys a certified product will not be surprised if it costs more than a competitor's product which is not certified. The extra cost will be justified by the expectation that the certified product is, in some way superior. Above all, the customer will expect the manufacturer to support a certified product in the event of any problems. It may be expected that the manufacturer will indemnify the user

against any consequential loss, attributable to the use of the product, and that the certificate will provide protection against third-party claims for damages.

2.8.3 The manufacturer's view

The software manufacturer's view is likely to be the most diverse of all those concerned with certification. The view will be influenced by the manufacturers in depth knowledge of the problems in producing software economically, and from the need to gain a competitive 'edge' in the marketplace.
On the positive side, a certificate could signify a superior product which would sell better. Independent assessment would provide a further layer of test and checking for errors that may have remained undiscovered during development.

2.8.4 The Independent evaluator's view

Clearly, the independent evaluator (e.g. a 'test lab') will see certification as a valuable source of work and will build an organisation suited to the needs of the task. This will involve suitably qualified staff who are familiar with the problems associated with a range of applications, the availability of suitable standards, tools and techniques, and the economic costs associated with the process. One particular cause for concern will be the possibility of being deemed to be responsible when certified software fails.

2.8.5 The legal community's view

The legal community's view will be constrained by the legislation which relates to software. Examples of such legislation are Copyright, Product Liability and Consumer Protection laws, reinforced by the legal judgements made in relevant cases.
There is, as yet, little experience from litigation concerning software other than on alleged infringement of copyright. The legal problems that are related to certification are much more likely to arise from failure of the software in some degree. Claims for economic or financial loss, damage, injury or death which are allegedly caused by the failure of a certified software product could be expected.
Questions concerning whether software is a product or service, or dealing with the failure of a certified product (e.g. Who should be liable ? - Should it be the software developer, the independent assessor, or both developer and assessor) All these questions need to addressed.

2.8.6 The public's view

It is extremely likely that the general public will have little knowledge of the

limitations of a certificate awarded to software. Like the user, the public's perception will be that of a guarantee against failure, a certificate that the product is reliable and safe.

2.9 Factors which may have an influence on Certification

2.9.1 Software in use

Application

The application to which software is directed will determine the level of certification necessary. The same software product (e.g. a spreadsheet program) may be used in different ways with quite different levels of risk in use and consequence of failure within different sectors.

User environment

The environment in which a software product will be used is extremely difficult to control or define. The principal problem will be to decide a representative environment for certification.

Risks and consequences of failure

The consequences of software failure vary widely, from minor irritation to human or environmental disaster. The need to assure a low probability of failure for a software product will affect the cost of certification.

Host computer

Programs can run under both different computers and different operating systems. This may involve multiple certifications of the same product.

Maturity

Maturity will be important in considering both applications and products. Can experience of the same or similar applications help to guide the certifier ? Is the product already in use, with end-user experience ? Is the product similar to another product that has already been certified or is it just a new version ?

Expertise of user

Software products are used by people ranging from naive to expert users, so the user experience level will differ significantly for the same product.

Support of producer

The level of in-service support available from the producer may have an influence on the integrity of a product. Support will vary from minimal for a low-cost commercial package, to continuous on-site support for complex 'one-off' commissioned applications.

2.9.2 Software development

Experience of developer

The experience of a software developer should be considered. The maturity of the organisation, skill of the staff, development methods employed, familiarity with the application, are all factors which have been shown to influence product quality.

Interaction with the customer or developer

Besides the support for a product provided by a software developer, the level of interaction with the customer (who may not be the end user) in formulating the requirements is an important factor.

Commercial constraints

The quality of the final product will be influenced by a number of commercial constraints. Budgets, profits and time scales will all influence the design, development methodology, and level of testing respectively.

Methodology used

A wide range of development methods are in use. Some of these are new and others are mature with recognised standards. One method may be preferred by a particular software developer, e.g. in the UK, SSADM is used for governmental administrative applications. The maturity of the methodology (in terms of experience and results achieved) may also influence the scheme of certification required.

Programming language

Programs are implemented in a wide variety of languages and a certification process must be able to accommodate the use of any of them in a product. Some applications may require the use of more than one language - e.g. real time applications are often a mixture of high level languages and assembler.

2.9.3 Software in systems

Minimum systems

A program must be associated with a processor before it can be used, and this will form a minimum system. Usually, the program will be loaded on to a computer, but it is not uncommon for a program to be an integral part of a processor e.g. a computer stored on a chip.

Typical systems

Usually the computer and program will be managed by an operating system. This combination will interact through peripheral equipment with an operator and other equipment such as mass storage devices, vdus, printers, plotters. In some applications it will be necessary to ensure that peripheral equipment is included as an integral part of the certification process.

Complex systems

Complex based computer systems will pose the most difficult problems for a certification process. These systems involve computer control of electronics, devices which may be mechanical, hydraulic or chemical processes, or plants. The functionality of this type of system may be inseparable from the functionality of the software.

2.10 Experiences of Software Certification

Although the use of certification is not widespread in the software industry as in other industrial areas, there are some domains which have experience in certifying software. Two such domains, the nuclear industry and the civil aviation authority,.concern very high criticality software for which certification is required.

2.11 Summary

I have looked at the SCOPE project along with the notion of software certification.When human lives are at risk the notion of certification is approached very cautiously. In the next chapter I examine the case study TANGO, in which DCU played the role as 'certifiers'.

Chapter 3

The Tango Case Study

3.1 Introduction

The SCOPE case study deliverable focused on six different case studies. They were intended to represent future independent third party certification of software. They included different types of software, with varying criticality levels, which hadn't undergone any certification process.
The DCU case study focused on a product code named as Tango. There follows a description of Tango, and the objectives of the case study in relation to the SCOPE project. The rigorous model developed during the course of the case study forms the basis for my object-oriented testing approach described in Chapter five.

3.2 Tango description

Tango is a computerised medical records system for use in health screening clinics. It provides complete clinical management facilities for health screening clinics and large companies that provide a screening service. Its main objective is to computerise the entire process of patient administration in such an environment.
The main functions of Tango include :

Patient Appointment Management
Medical Report Generation
Patient Recall Management
Medical Test Results Storage
Patient Billing

The system also provides a number of administrative functions which include the back-up of the database to floppy disks and access to a separate word processing package which allows the user to change reports. Tango was developed over a five man month period. It was developed in a strictly informal manner with no functional specification. As a result there is an incomplete development trail. The system was written in a high-level fourth generation language - INFORMIX 4GL.The system incorporates the use of UNIPLEX to cater for the word-processing requirements mentioned. Therefore, Tango depends on other products in the run-time environment. Tango runs on a 386 PC based machine under the UNIX operating system. It consists of approximately 7-8K of source code. The memory requirements are 2MB of RAM for one user, with an extra 1MB for each additional user. One patient record requires 4KB of disk memory. The documentation provided with Tango is informal and consists of a user/technical manual. The developer is prepared

to make modifications to the software to suit specific customer needs. It is possible to have multiple versions of the product existing with minor differences. The typical Tango user is a non-computing expert who is hoping that Tango will contribute to improved efficiency and effectiveness in running his/her business.

3.3 Quality attributes of Tango to be assessed

As case study partners, DCU took on the role of 'certifiers'. The first step was to decide on the product attributes to be assessed. These are included in the 'Certification specification'. The attributes are correctness, reliability, integrity/security, usability, 'ease of housekeeping', performance, quality of documentation, and maintainability. A brief description of each product attribute is given.

Correctness : Tango must provide the functionality claimed for it in user-documentation. The product described is the basis of the contract between the developer and customer.

Reliability : Tango plays an essential role in the operations of a health screening clinic. System crashes etc.. would pose very serious problems for the user. Hence, reliability is of great importance.

Integrity/Security : Tango is used in health screening clinics so it deals with very sensitive personal data. Security/Integrity of data is therefore very important. Access to sensitive data should be restricted and properly controlled. As an important integrity aspect, due to Tango's application domain, loss or corruption of data must be minimised at all times.

Usability : Tango is used by an ever changing population of non-expert users, so Tango must be easy to learn and use. It should cope well with users' mistakes and provide good error detection and correction. The user interface is of great importance.

'Ease of housekeeping' : Users will not be experts. Routine operations such as doing back-ups, restoring from a crash etc., should be easy to do.

Performance : Tango is used in real time during consultation with a patient so processing times and response times must be stated. Limiting factors such as file sizes should also be clearly stated and accurate.

Quality of documentation : Tango documentation should be readable, complete, consistent and accurate. The documentation is a description of Tango's functionality, so clarity and completeness of documentation are of vital importance.

Maintainability : Tango software and documentation must be easy to maintain as the product is tailored for individual sites and is subject to ad hoc field maintenance. for bug correction, functional enhancement and adaption to changes in the run time platform.

3.4 Objectives of the Tango case study

While working on the case study I invested most of my effort in assessing 'correctness' and 'quality of documentation' as I felt that the most important question to be answered in the certification of a product like Tango is :

"Does Tango do what its supportive documentation claims it does ?"

as this is the basis for the contract between the developer and the supplier. To answer this question involves answering two subsidiary questions :

(1) : **"What is it claimed Tango does?"**

and

(2) : **"Does Tango meet the claims?"**

The answer to the first subsidiary question lies in the supportive documentation (brochures, technical and user manuals).

The answer to the second subsidiary question is obtained by black box testing to reconcile actual functionality with claimed functionality.

The notion of "reverse engineering" a rigorous description of claimed functionality from the user documentation was a major aim of the case study, particularly seen as how I could apply this rigorous model. The process of reverse engineering would also expose ambiguities and inconsistencies in Tango documentation. A major use for the model would also be in generating black box test cases. This would exercise the functionality of the system better than test cases based simply upon page 26 of the user manual which says that "if I choose option three from the patient menu then a patient data entry screen will be shown". The rigorous model would therefore help to assess the quality(e.g. accuracy, completeness) of the user documentation in respect of its description of the functionality of Tango. The experience gained with Tango would provide valuable insights into the problems and potential of certification of low cost products, produced through an informal small scale development process.

3.5 Assessing quality of documentation

The documentation on which the 'purchaser' will make his/her decision may suffer from a number of flaws. These flaws can at a syntactic or semantic level. Here the process of building the rigorous model of Tango is described by applying it to an early draft of a piece of real user-documentation. This modelling approach is object-oriented, based upon the ideas of *[Boo 86]*, *[Coa 90]* ,*[Ver 88]* and *[Mey 88]*.
Also some ideas from the field of formal specification are referenced (*[Jon 90]*,*[Woo 88]*).

3.5.1 Developing the rigorous model

This section describes a method for critically evaluating Tango's user-oriented functional specification by 'reverse engineering' a rigorous model of the system's functionality from the documentation *[Moy 90]*. The objective of reverse engineering is to increase the overall comprehensibility of a system for both maintenance and new development. This is achieved through analysing a subject system to identify the system components and their interrelationships, and to create a representation of that system in another form or at a higher level of abstraction *[Chi 90]*. The process of building the model helps to expose the structure of the system and gives clues about possible defects in the documentation.
The model that was constructed forms the basis for my object-oriented testing approach in Chapter five. The method is applied to a real instance of documentation and highlights a number of significant flaws which would probably have been missed by a traditional document review.

Here is the list of sequential steps taken towards developing the rigorous model.

Step 1 : Identify candidate objects
Step 2 : Identify object-class and assembly-structure hierarchies
Step 3 : Identify attributes and user-operations for each object
Step 4 : Identify instance connections between candidate objects
Step 5 : First document review session with developer
Step 6 : Extract the pre- and post-conditions for each operation
Step 7 : Confirm the existence of operations to add, modify, and remove instances of each object type
Step 8 : Final document review session with the developer

Step 1 : Identify candidate objects

The first step was to identify the application-domain objects and object-classes which are referenced in the documentation. Some objects were excluded, even those clearly playing an important part in the application environment, which were not explicitly recognised by the software.

Each object was given a unique name, generally the name by which it was most frequently referred to in the documentation. Also noted were any apparent synonyms for the same object. Nouns were judged to be possible synonyms if they appeared to be used interchangably in the documentation or if they 'sounded' the same at a common sense level. Further confirmation that two or more nouns were really synonyms for the one object was obtained if the same user-operations seemed to be applicable to each and if they appeared to share the same attributes.The high frequency of apparent synonyms made the task quite difficult and represented a serious flaw in the document.

Object	Synonym(s)
ADMISSION	
ADMISSION NOTIFICATION	
ADMISSIONS LIST	RECORD BOOK
APPOINTMENT SCHEDULE LIST	DAY-SHEET
BLOOD DATA RESULTS	
BLOOD DATA TEST	
BIOCHEMISTRY RESULT	
BIOCHEMISTRY TEST	
BLOOD PRESSURE RESULTS	
BLOOD PRESSURE TEST	
CHEST X-RAY RESULT	
CHEST X-RAY TEST	
CONFIGURATION PARAMETERS	
CONSTANT	
CROWN CRISP TEST	
CROWN CRISP TEST RESULT	
CYTOLOGY RESULT	
CYTOLOGY TEST	
CYTOLOGY TEXT	
DOCTOR EXAMINATION CODE	
DOCTOR EXAMINATION	PHYSICAL EXAMINATION
DOCTOR EXAMINATION RESULTS	
DOCTOR EXAMINATION TEXT	
DOCTOR TEST RESULTS	
DOCTORS TEST	
ECG TEST	
ECG TEST RESULT	
ECG TEXT	
FAMILY HISTORY TEST	
FAMILY HISTORY TEST RESULT	
GP LETTER	DOCTORS LETTER
HAEMOCULT TEST	

Figure 3.1

Object	Synonym(s)
HAEMOCULT TEST RESULTS	
HAEMOTOLOGY RESULT	
HAEMOTOLOGY TEST	
HAEMOTOLOGY TEXT	
HEARING TEST	AUDIOMETRY TEST
HEARING TEST RESULTS	
HEIGHT AND WEIGHT RESULTS	
HEIGHT AND WEIGHT TEST	
INVOICE	
LABEL	
LABORATORY TEST	
LABORATORY TEST RESULTS	
LETTER	
LUNG FUNCTION RESULTS	
LUNG FUNCTION TEST	
MAMMOGRAM RESULT	
MAMMOGRAM TEST	
MAMMOGRAM TEXT	
MEDICAL FORMULA	
NEW PATIENT	
NEW PATIENT APPOINTMENT	
NURSE TEST RESULTS	
PATIENT	
NURSES TEST	
OUTPUT	
PAST HISTORY TEST	
PAST HISTORY TEST RESULT	
PATHOLOGY RESULT	
PATHOLOGY TEST	ADDITIONAL PATHOLOGY
PATHOLOGY TEXT	
PATIENT ADDRESS LABEL	

Figure 3.1 (continued)

Object	Synonym(s)
PATIENT APPOINTMENT	VISIT
PATIENT CONFIRMATION LETTER	CONFIRMATION LETTER
PATIENT FOLLOW-UP REPORT	
PATIENT LETTER	
PATIENT MEDICAL REPORT	MEDICAL REPORT
PATIENT RECALL LABEL	
PATIENT RECALL LETTER	RECALL LETTER
PATIENT TEST RESULT	
PERSONAL DETAILS RESULTS	
PERSONAL DETAILS TEST	
PHYSICAL EXAMINATION DEFAULT TEXTS	STANDARD TEXTS
PREVIOUS VISIT SUMMARY MEDICAL REPORT	PATIENT SUMMARY MEDICAL REPORT
PRINTER PARAMETER	
REPORT	
REPORT PARAMETER	
RETURNING PATIENT	
RETURNING PATIENT APPOINTMENT	
SAMPLE IDENTIFICATION LABEL	
SCREENING DOCTOR	DOCTOR
SCREENING NURSE	NURSE
SCREENING TYPE	
SOCIAL HABITS TEST	
SOCIAL HABITS TEST RESULTS	
TEST	
TEST RESULT CODE	
TEST RESULT IDEAL RANGE	
URINALAYSIS RESULT	
URINALAYSIS TEST	
VISION RESULTS	
VISION TEST	
X-RAY TEXT	

Figure 3.1 (continued)

Step 2 : Identify object-class and assembly-structure hierarchies

The object-types shown in **Figure 3.1** are highly interrelated. Some object-types denote a super-class or sub-class of other object-types. For example, TEST is a super-class which includes DOCTORS TEST,NURSES TEST and LABORATORY TEST as sub-classes. Some object-types are assembly structures built from two or more other object-types. An example is where, PATIENT TEST RESULT is the aggregate of DOCTORS TEST RESULT, NURSES TEST RESULT and LABORATORY TEST RESULT. **Figure 3.1** also includes 'objects' which are actually individual instances of an object-class. For example CROWN CRISP TEST is an instance of DOCTORS TEST.
Such object structures were looked for in the documentation. There was considerable difficulty in doing so. Only one hierarchy was explicitly identified. Other hierarchies were implicit and were pieced-together only with painstaking study of the document. **Figure 3.2** shows the provisional object structures that were found. The notation used to represent the structures is drawn from *[Coa 90]*. The 'consists of' relationship in an assembly-structure is represented with a triangle pointing to the aggregate object.
The exercise was well worth while. It greatly added to the understanding of the scope and static structure of Tango. A nice example of abstraction in action.

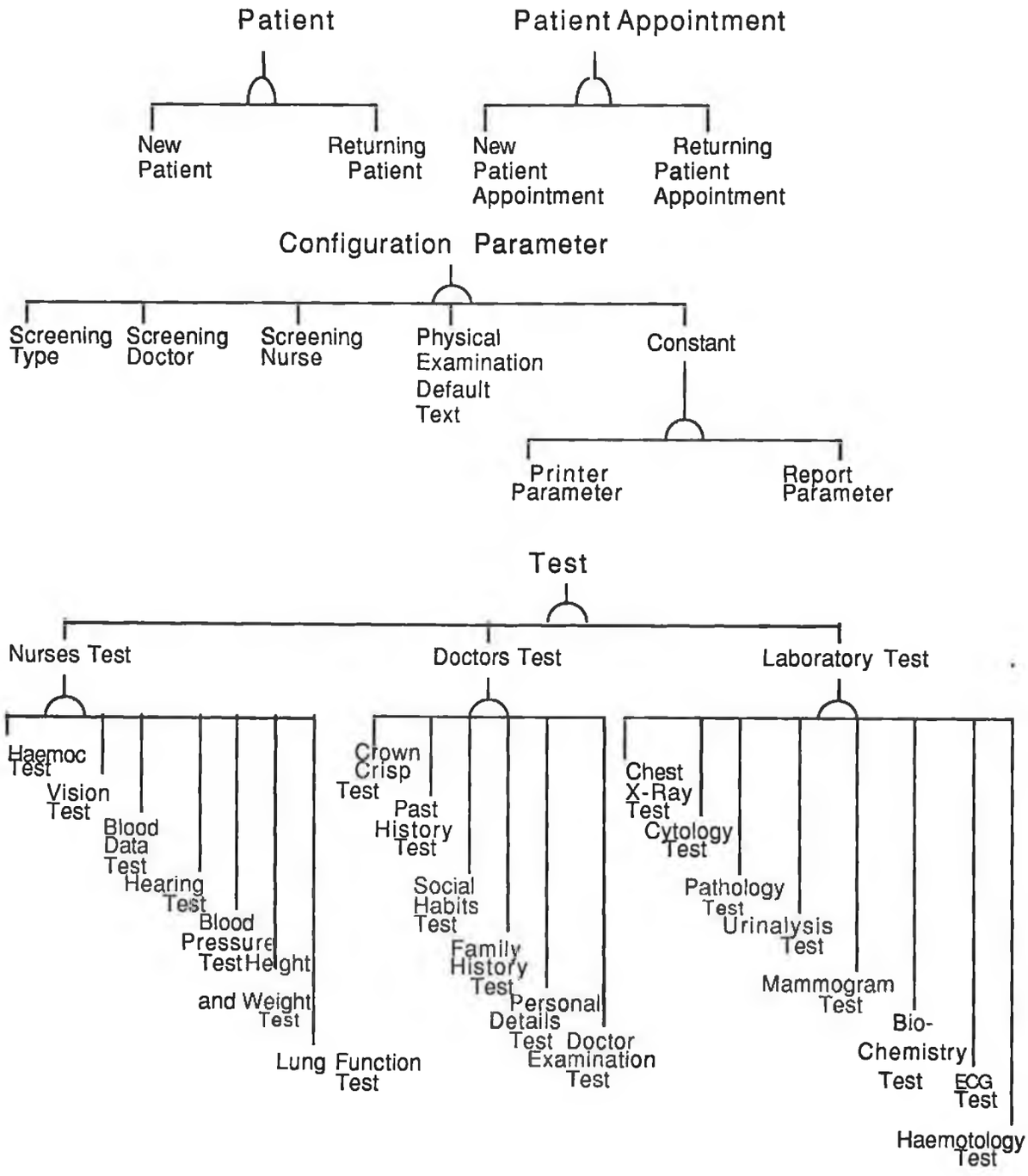


Figure 3.2 : The Object Class Hierarchies

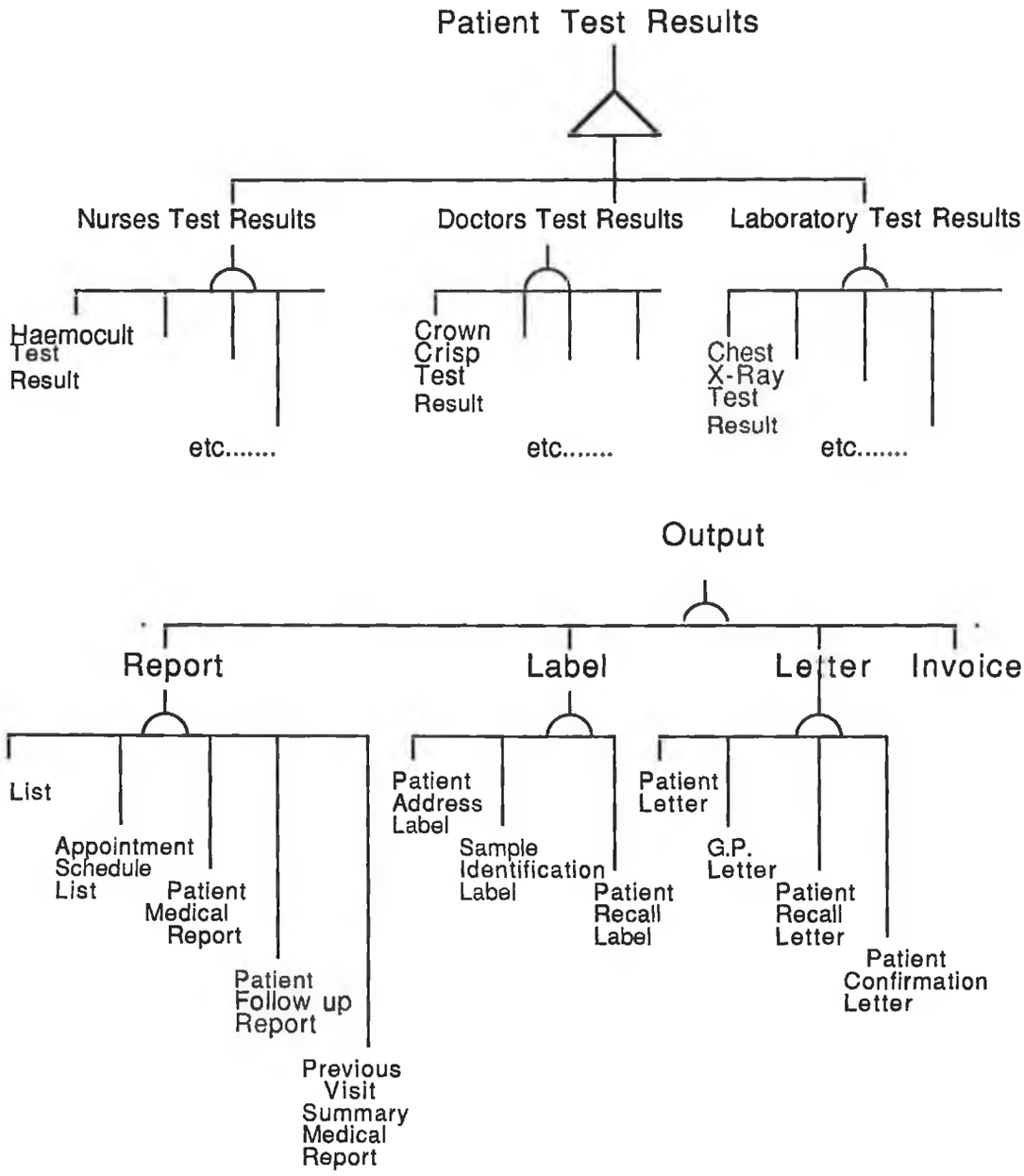


Figure 3.2 : The Object Class Hierarchies (continued)

Step 3 : Identify attributes and user-operations for each object

The next step was to identify the attributes of each candidate object and the user-operations which could be applied to the object. It was found that a number of user-operations automatically triggered-off a series of 'side-effect' operations on other objects. For example, the user-documentation states that when the user attempts to add an appointment for a 'new' patient, the personal details of the new patient are automatically requested, and a record for that patient automatically added to the patient details file, before the appointment is processed. Some of these automatically invoked operations may also be invoked directly by the user whilst others may not. A simple notation was developed to capture this behaviour.

The analysis was restricted to operations explicitly described in the user-documentation. There was no surmise about 'invisible' operations internal to the software.

Some difficulty was experienced in gathering the attributes and operations for many of the objects because these were not explicitly recorded in an 'easy to get at way' in the document. The presence of apparent synonyms for some attributes and operations added to the difficulty of the task.

The process threw up a number of apparent anomalies in the documentation. For many 'objects' no attributes and no operations could be found. Some obvious questions posed were : Could it be that these 'empty' objects were not really objects in their own right ? Could it be that they were really attributes of other objects, unrecognised synonyms, or were intended to play some other role in the system ? These problematic objects are listed in **Figure 3.3.**

ADMISSION	**MEDICAL FORMULA**
ADMISSION NOTIFICATION	**DOCTOR EXAMINATION CODE**
MAMMOGRAM TEXT	**ECG TEXT**
TEST RESULT IDEAL RANGE	**HAEMOTOLOGY TEXT**
TEST RESULT CODE	**PATHOLOGY TEXT**
DOCTOR EXAMINATION RESULT	**X-RAY TEXT**
DOCTOR EXAMINATION	**CYTOLOGY TEXT**
PHYSICAL EXAMINATION DEFAULT TEXT	**DOCTOR EXAMINATION TEXT**

Figure 3.3 - Candidate Objects for which no operations and no attributes could be found

Figure 3.4 shows the attributes and user-operations that were identified for each 'non-empty' object. For economy and clarity of presentation, the full use of 'inheritance' was made in that each attribute and user operation is located in the most general object-class to which it appears to apply. Members of sub-classes inherit these attributes and operations from their super-classes. **Figure 3.4** gives a very good overview of the structure and functionality of Tango. This is a good example of the descriptive power of abstraction and inheritance when used in combination.

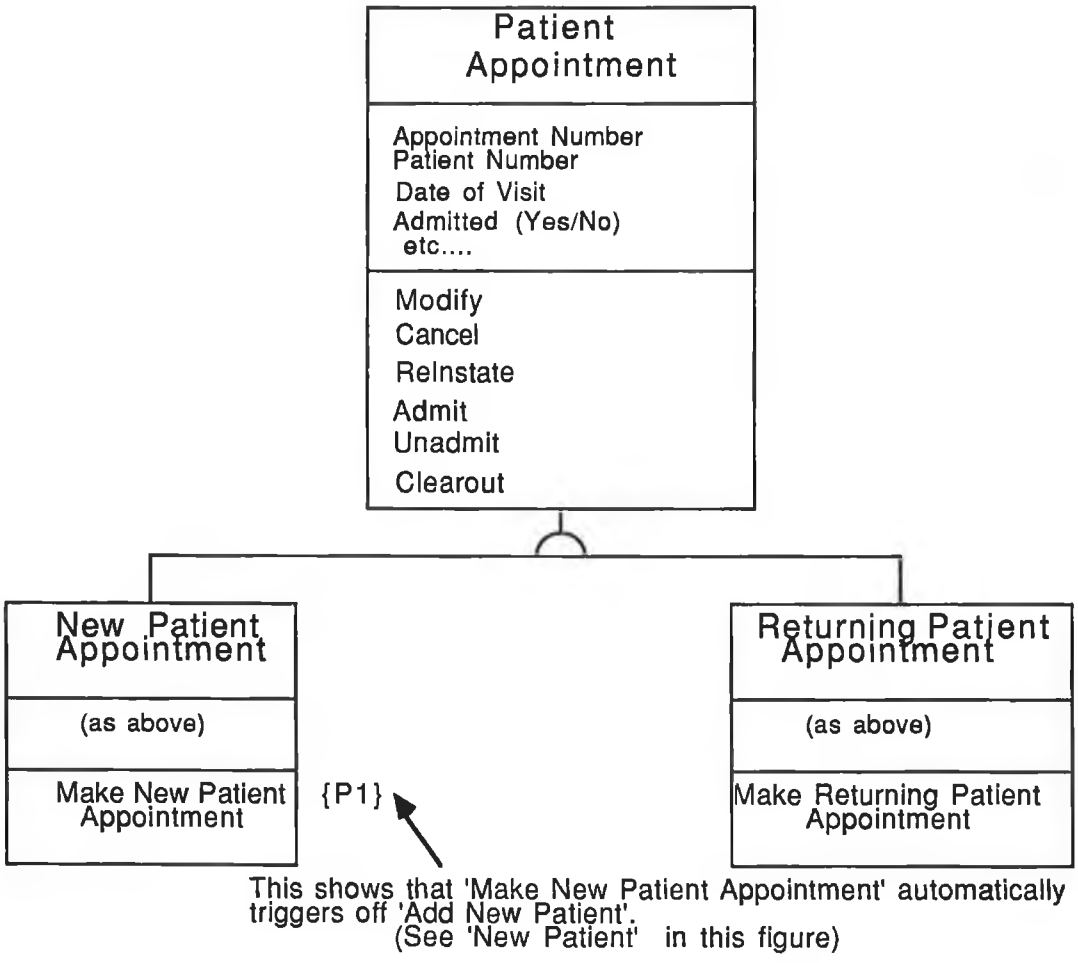


Figure 3.4 : Attributes and operations

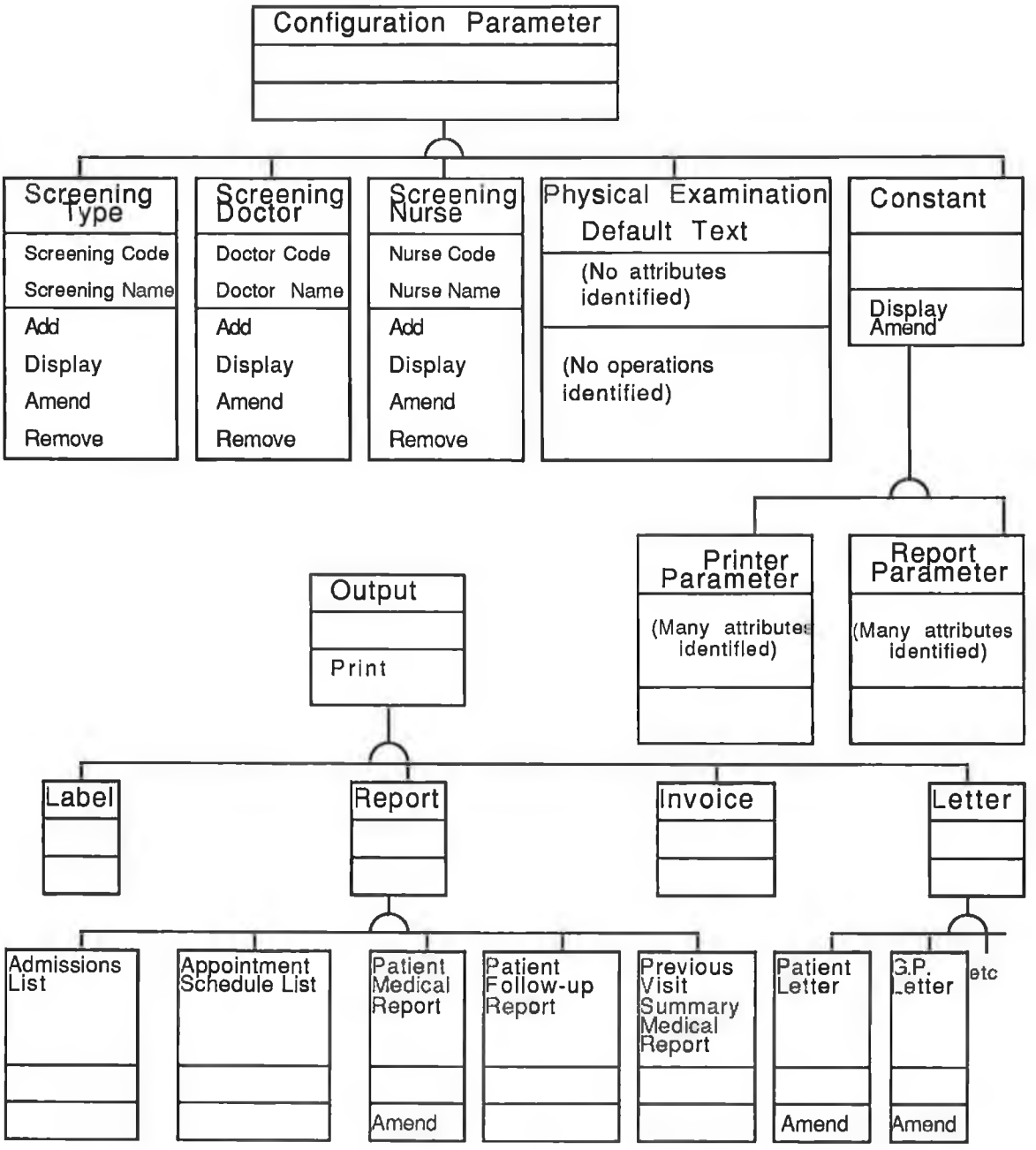
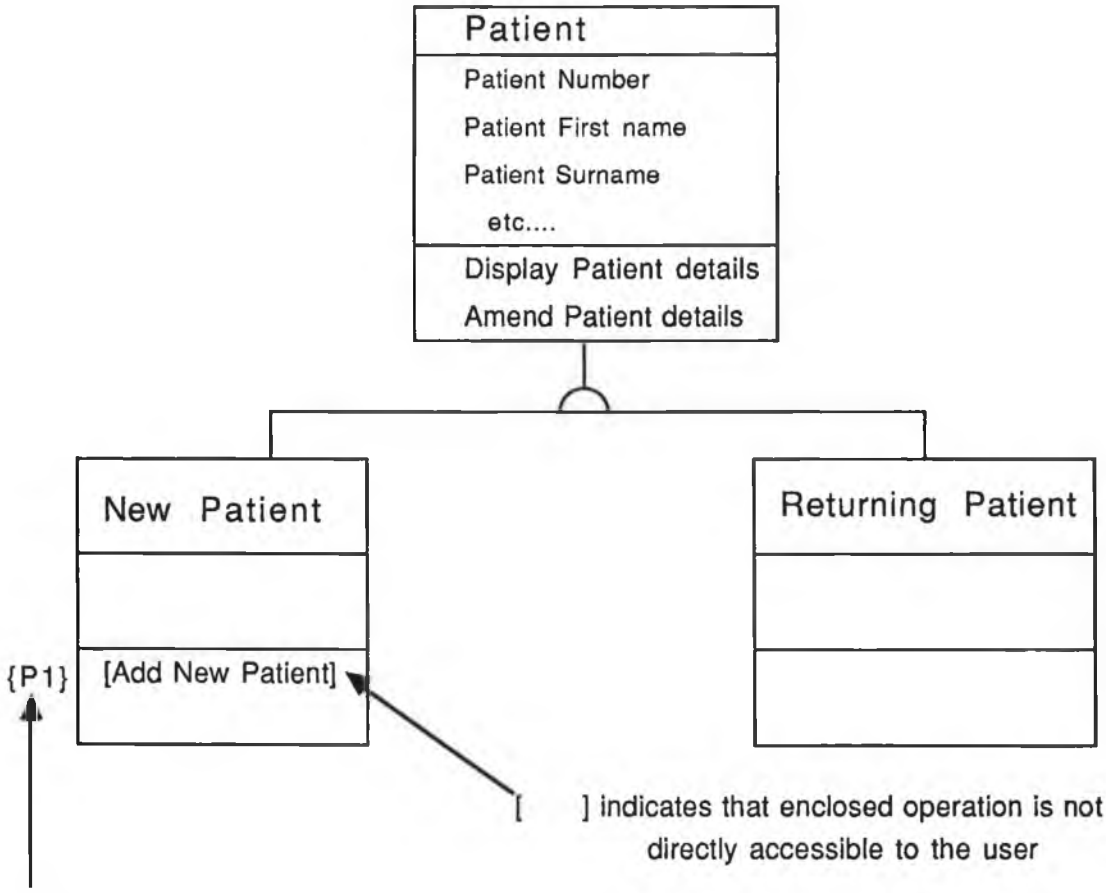
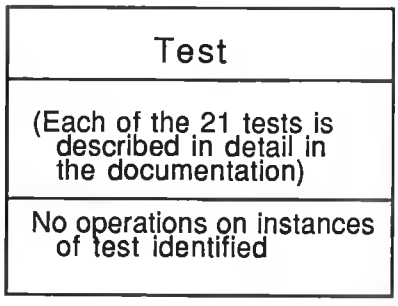
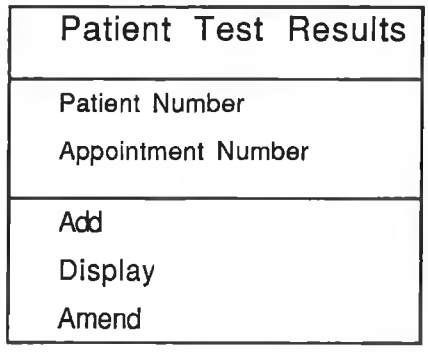


Figure 3.4 : Attributes and operations (continued)


Patient
Patient Number
Patient First name
Patient Surname
etc....
Display Patient details
Amend Patient details
New Patient
[Add New Patient]
Returning Patient
{P1}
[ ] indicates that enclosed operation is not directly accessible to the user
This shows that 'Add New Patient' is automatically triggered by 'Make New Appointment'.
(See 'Patient Appointment' in this figure)


Test
(Each of the 21 tests is described in detail in the documentation)
No operations on instances of test identified


Patient Test Results
Patient Number
Appointment Number
Add
Display
Amend

Step 4 : Identify instance connections between candidate objects

An instance connection is a constraint on the existence of an instance of an object-class. For example, a set of test results for a patient (i.e. PATIENT TEST RESULTS) can exist only if there is a corresponding instance of a PATIENT APPOINTMENT. Instance connections are an important component in the description of any system.

Figure 3.5 shows the instance connections that could be identified from Tango documentation. The instance connections are drawn at the highest level of generalisation at which they appear to apply. Thus lower level object-classes are shown in **Figure 3.5** only if they participate in instance connections in which the more general or aggregate object does not participate. The cardinality of each connection is also shown. Where cardinality could not be determined from the documentation, a "?" appears. The notation is used to express the cardinality of an instance connection. Here is an example:

APPLE<1:2--------1:1>ORANGE shows that each instance of APPLE must be associated with exactly one instance of ORANGE and that each instance of ORANGE must be associated with one or two instances of APPLE. The task of constructing **Figure 3.5** from user-documentation proved to be rather difficult. Few instance connections were explicitly identified.

A number of apparent anomalies were identified. These fall into two categories :

(1) The existence of seemingly 'disconnected' objects

Despite a lot of effort no instance connections for sixteen candidate objects were identified. (see **Figure 3.6**). Maybe connections do exist but they could not be deduced from the documentation. These sixteen 'disconnected' objects happen to be the same sixteen 'objects' for which no attributes and operations were found (see **Figure 3.3**).

(2) The existence of instance connections of unspecified cardinality. The cardinality of four instance connections could not be established from the user-documentation:

PATIENT <1:1-------0:?> PATIENT-APPOINTMENT
NURSE'S TEST RESULTS <?:1-----?:7> NURSES' TEST
DOCTOR'S TEST RESULTS <?:1-----?:6> DOCTORS'TEST
LABORATORY TEST RESULTS <?:1-----?:8> LABORATORY TEST

These unspecified cardinalities raise a number of questions which could be of great importance to potential users of Tango. For example, can only one future scheduled appointment exist at any given time for a patient? Or is it possible to stack-up a number of future appointments for a patient? If some of the nurses',doctors' or laboratory tests are not administered to a patient , do the test results for the tests which ARE administered constitute a valid set of patient test results?

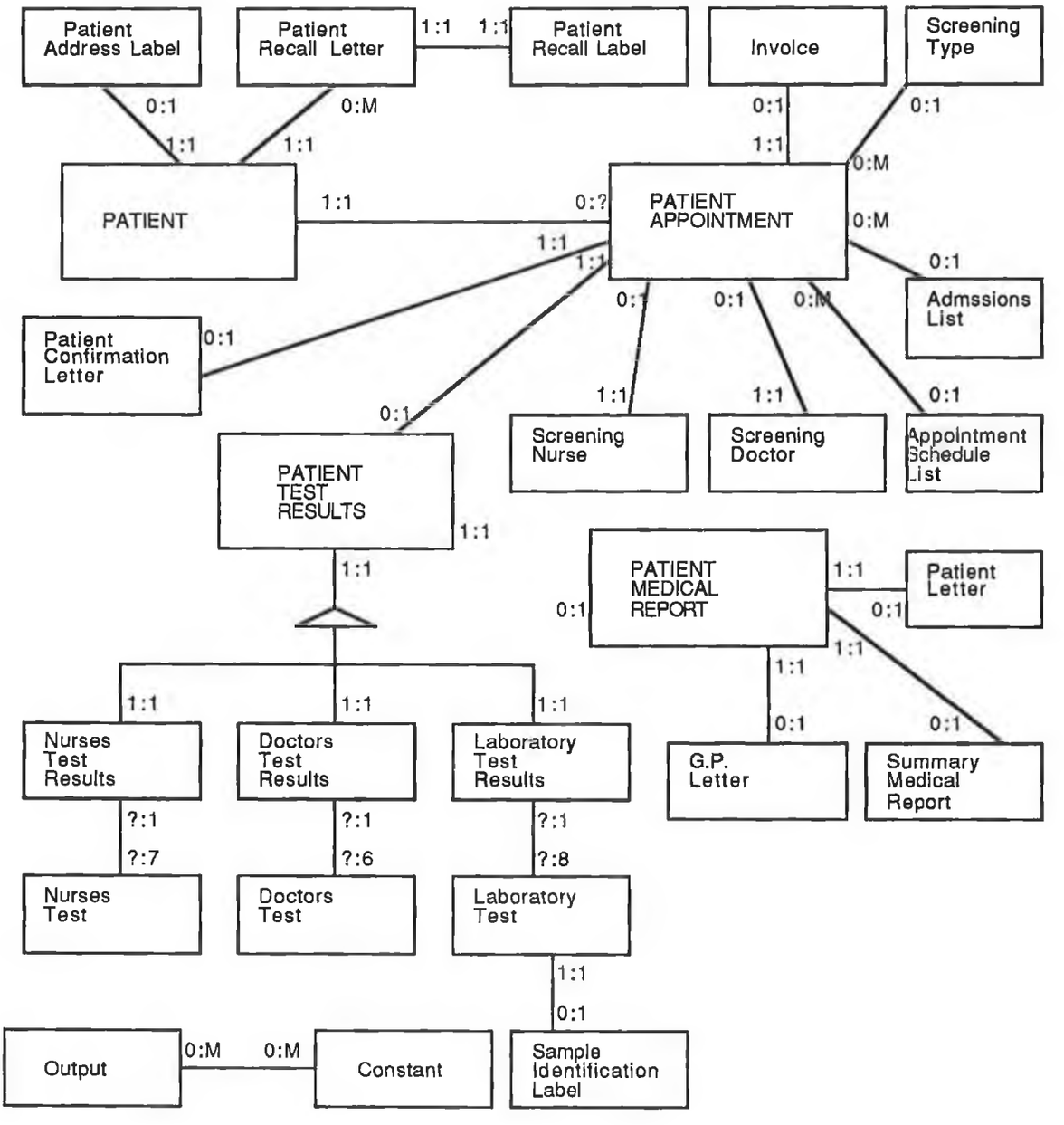


Figure 3.5 : Instance Connections

ADMISSION	**MEDICAL FORMULA**
ADMISSION NOTIFICATION	**DOCTOR EXAMINATION CODE**
MAMMOGRAM TEXT	**ECG TEXT**
TEST RESULT IDEAL RANGE	**HAEMOTOLOGY TEXT**
TEST RESULT CODE	**PATHOLOGY TEXT**
DOCTOR EXAMINATION RESULT	**X-RAY TEXT**
DOCTOR EXAMINATION	**CYTOLOGY TEXT**
PHYSICAL EXAMINATION DEFAULT TEXT	**DOCTOR EXAMINATION TEXT**

Figure 3.6

Step 5 : First document review session with developer

Steps 1 through 4 identified a number of 'puzzles' with Tango's user-documentation. Before proceeding any further with the analysis of the document, a document review session was held with the developer to confirm the validity of the analysis and to sort out the 'puzzles'.

First **Figure 3.1** was reviewed. It was confirmed that all but one of the candidate synonyms really are synonyms. VISIT is not a synonym for PATIENT APPOINTMENT as thought, but is a synonym for ADMISSION.

Next **Figures 3.3** and **3.6** were reviewed which list, respectively, objects that are apparently 'empty' (i.e. have no attributes and no operations) and objects that do not seem to participate in any instance connections. It was learnt that DOCTOR EXAMINATION actually consists of nineteen different tests, called PHYSICAL EXAMINATION TESTs, which involve examining some physical attribute of the patient (for example his/her eyes). Each of these tests is identified with a unique code (DOCTOR EXAMINATION CODE synonym for which is TEST RESULT CODE) which is an attribute of the test rather than an object in its own right. The result for a test is DOCTOR EXAMINATION RESULT. Associated with each test is a paragraph of standard text (PHYSICAL EXAMINATION DEFAULT TEXT). The doctor may include this paragraph of text in the PATIENT MEDICAL REPORT if the result of the test is 'normal'. If the result is 'abnormal', the doctor may generate his/her own text (DOCTOR EXAMINATION TEXT).

It was also learnt that MEDICAL FORMULAs and TEST RESULT IDEAL RANGEs are algorithms and constants respectively used by the software when computing PATIENT TEST RESULTS. These object types appear to be 'hard-coded' into the system. The term ADMISSION NOTIFICATION was used in error in the documentation. It should have read APPOINTMENT NOTIFICATION and is a further synonym for PATIENT CONFIRMATION LETTER. The term ADMISSION refers to a PATIENT APPOINTMENT for which the PATIENT has attended and undergone tests. MAMMOGRAM TEXT, ECG TEXT, HAEMOTOLOGY TEXT, PATHOLOGY TEXT, X-RAY TEXT and CYTOLOGY TEXT refer to free text paragraphs which may be generated by clinic staff for inclusion in the PATIENT MEDICAL REPORT if, for example, the results of these tests are abnormal.

Figure 3.5 (Instance Connections) was then reviewed. It was confirmed that the analysis was correct. The four instance connections for which no cardinalities could be found were looked at in particular The review established that the system will allow only one future, scheduled PATIENT APPOINTMENT to exist for any PATIENT (i.e. future appointments can't be stacked up for a patient). From the review it was learnt that a partial set of test results does constitute a valid set PATIENT TEST RESULTS (i.e. not all tests must be administered to a patient).

Following on the review the various diagrams were amended as necessary. Having resolved all the puzzles, the next step in the analysis took place.

Step 6 : Extract the pre- and post-conditions for each operation

This step concerned constructing the pre- and post-conditions for each user-operation. This proved to be moderately difficult. Very few of the conditions were made explicit in the documentation. This meant relying heavily on a limited application domain knowledge and on intuition. As a result, there was no reason to suggest that the analysis was accurate and complete. **Figure 3.7** shows a 'best shot' at making the pre- and post-conditions explicit. As in earlier figures, each operation is attached to the most general object-class to which it applies. It is made explicit in the diagram where the pre- or post-conditions differ for a sub-class. This use of abstraction greatly simplifies the presentation.

The notation used for recording the conditions is fairly informal, but was adequate for the purpose in hand. Had the complexity of the application been greater, it would have been necessary to adopt a more formal and elaborate notation such as that used in VDM *[Jon 90]* or Z *[Woo 88]*. In some of the conditions, the notion of a 'state-indicator' is introduced *[Ski 90]*. The use of the state-indicator helped to express constraints on the sequences of operations which may be validly applied to an instance of an object-class, and helped to express alternative post-conditions where the effect of an operation is contingent on the internal state of the object.
In addition to a general feeling of unease about the accuracy and completeness of **Figure 3.7**, a number of specific ambiguities in the documentation were identified. For example, can the details of a PATIENT APPOINTMENT be modified after the PATIENT has been admitted? Is it possible to cancel a PATIENT APPOINTMENT which has not resulted in an ADMISSION but which is less than three months old (and thus not removable by 'clearout')? To make a RETURNING PATIENT APPOINTMENT must there exist an instance of a previous, admitted PATIENT APPOINTMENT for this PATIENT ? Perhaps the answers to these questions are unimportant to users - but maybe this is not so.

Object	Operation	Pre-Condition	Post-Condition
Patient	Display Patient details	Patient present	(self evident)
	Amend Patient details	Patient present	(self evident)
Patient Appointment	Modify	state indicator = 1 or ?	state indicator unchanged
	Cancel	state indicator = 1 or ?	state indicator = 2
	Reinstate	state indicator = 2	state indicator = 1
	Admit	state indicator = 1	state indicator = 3
	Unadmit	state indicator = 3	state indicator = 1
	Clearout	date of visit earlier than 3 months ago and	
		if state indicator = 3	Patient Appointment moved to archive table
		if state indicator ≠ 3	Patient Appointment deleted from system
Sub-Types:			
New Patient Appoinment	Make new Patient Appointment	Patient not present	Patient present and Patient Appointment present with state-indicator = 1
Returning Patient Appointment	Make Returning Patient Appointment	Patient present and no Patient Appointment present for this Patient with state-indicator = 1 and Patient Appointment present for this patient with state-indicator = 3	Patient Appointment present with state-indicator = 1

Figure 3.7 : Pre and Post conditions for user operations

Object	Operation	Pre-Condition	Post-Condition
Output	Print	(self evident)	(self evident)
Sub-Types:			
Patient Medical Report	Amend	(self evident)	(self evident)
Patient Letter	Amend	(self evident)	(self evident)
G.P. Letter	Amend	(self evident)	(self evident)
Configuration Parameter			
Sub-Types:			
Screening Type	Add Display Amend Remove	(self evident)	(self evident)
Screening Nurse	Add Display Amend Remove	(self evident)	(self evident)
Screening Doctor	Add Display Amend Remove	(self evident)	(self evident)
Physical Examination Default Text	(None)		

Figure 3.7 : Pre and Post conditions for user operations (continued)

Object	Operation	Pre-Condition	Post-Condition
Patient Test Results	Add Display Amend	Patient Appointment with state-indicator = 3 present for this Patient	(self evident)
Constants	Display Amend	(self evident) (self evident)	(self evident) (self evident)
Medical Formula	(None)		
Test Result Ideal Range	(None)		
Doctor Examination Text	Create Display Amend	(self evident) (self evident) (self evident)	(self evident) (self evident) (self evident)
Test	(None)		

Figure 3.7 : Pre and Post conditions for user operations (continued)

Step 7 : Confirm the existence of operations to add, modify, and remove instances of each object type

By its nature, one would expect any information system to provide user-operations to add and delete instances of persistent objects and operations to modify the values of the attributes. An exception to this generalisation would be an object, such as a table of 'hardwired' constants, which by design the user is not allowed to change. In step 7, an instance of each object-class was 'walked' through its life-cycle and confirmed, or otherwise, the presence of these basic operations. If these operations are missing from the documentation, it may signal an inadvertent omission from the documentation or a deliberate design decision which may have important implications for the user. **Figure 3.8** shows those object-classes for which one or more of the basic operations could not be found. Once again, the full use of abstraction is applied in presenting the results.

Figure 3.8 presents a number of puzzles. While a new instance of PATIENT is automatically created when a NEW PATIENT APPOINTMENT is made, there appears to be no operation which allows the user to directly create a new instance of PATIENT. Can a new patient be added only in the context of that patient making his/her first appointment?

It appears that instances of the following objects are 'hard-wired' into the system and may not be added, changed or removed by the user : TEST, MEDICAL FORMULA, TEST RESULT IDEAL RANGE and PHYSICAL EXAMINATION DEFAULT TEXT. The first three of these object-classes specify the various tests to be applied to patients and contain parameters and algorithms for processing test results. Thus they are very central to the operation of the system. Perhaps user operations for their maintenance exist in the software, but they do not appear in the documentation.

Object	Create Operations	Amend Operations	Remove Operations
Patient	No	Yes	No
Test	No	No	No
Medical Formula	No	No	No
Test Result Ideal Range	No	No	No
Physical Examination Default Text	No	No	No

Figure 3.8 : Persistent objects for which no Create, Amend or Remove user-operations could be found.

Step 8 : Final document review session with the developer

The purpose of this final step was to confirm the validity of the analysis in steps 6 and 7 and to feed back the new 'puzzles' that had been found in the documentation. The objectives were achieved during the review.

3.6 Conclusions from developing the rigorous model

An approach to critically examining the content of a piece of draft user-documentation has just been described. There is a degree of confidence in this approach in that the process of constructing the 'rigorous model' uncovered problems which a more traditional document review would have missed.

The notation used is relatively informal but was adequate for the task. However, to cope with more complex systems the notation would need elaboration and an increase in the level of formality. Also tool support would be required to analyse documents describing larger or more complex systems.

A good knowledge of the application domain is very important. The more the model builder knows about the application domain, the speedier and more thorough the analysis. Maybe the best person to build the model is a potential user!

For the future, the idea of re-writing a user-document so that its structure and content are isomorphic with that of the rigorous model seems to be valid. One can even visualise a tool which would take the model as input and automatically produce a 'skeleton' of the restructured document for completion by a human author. The new 'de-bugged' document would have a 'logical' structure which users might find easier to comprehend than a document structured in any other way. The degree of benefit in such restructuring would probably depend on the extent to which the constructs in the model reflect the user's 'mental model' of the application domain (see *[Car 88]*). This question can only be answered by empirical research.

3.7 Case study conclusions

These are the conclusions derived from the case study.

3.7.1 Judgements reached on 'Certification specification'

Figure 3.9 shows the 'judgements' reached on each product attribute agreed in the 'Certification specification'.

Attribute	Judgement	Reference
Correctness	**marginally acceptable**	**Chapter 3,5**
Reliability	**acceptable**	**Appendix C**
Logical data security	**unacceptable**	**Appendix B**
Usability	**unacceptable**	**Appendix A**
'Ease of house-keeping'	**acceptable**	**Appendix E**
Performance	**acceptable**	**Appendix F**
Documentation	**unacceptable**	**Chapter 3**
Maintainability	**acceptable**	**Appendix D**

Figure 3.9

The overall judgement was that documentation, logical data security and the user-interface required considerable improvement before the product could be 'approved'.

3.7.2 Tango and Certification

The experience with Tango suggested that it is possible to 'certify' an off-the-shelf PC-based software product within an acceptable time scale and cost. The importance of building a well-thought-out and agreed 'Certification Specification' with the developer at the outset of a certification exercise also proved to be a valuable exercise. Also the view that all certification exercises must include 'reliability' and 'correctness' amongst the attributes to be assessed was agreed. These two attributes are fundamental, regardless of the application and the context of use. Other attributes such as portability, usability and maintainability may or may not be important in particular instances. Therefore, these attributes might not appear in a certification specification. Based on the experience with

Tango the 'certificate' must make explicit the particular attributes which were tested and approved. An unqualified,'blanket' imprimatur on a software product is a very dubious proposition. The certification process must include feedback to the developer on what needs to be done to the product to make it 'certifiable' should it fall short of acceptability on an attribute. Thus, technology transfer and product and process improvement will be encouraged.

3.7.3 Inspection and checklists

The case study placed a lot of emphasis on the use of inspection and checklists. In the absence of agreed standards for most if not all of the product attributes examined, professional judgement was often used in reaching a set of conclusions. Despite any progress that may be made in the future in the science of software metrics, professional judgement will play a big role in certification exercises.

3.7.4 Tools

During the case study I sought tools to support the approach in developing a rigorous model of Tango's functionality from its documentation. I didn't find any tools to support this task. Again no tool was readily available. A tool was needed to compute white-box maintainability metrics on Tango source code. Existing tools don't have a front end for INFORMIX 4GL so a manual approach was adopted to computing metrics. (See Appendix D). Also I needed a tool to help generate test cases from the rigorous model. Again no tool was readily available.

3.8 Summary

The Tango case study proved very useful in terms of a 'certification' exercise. Although only a prototype attempt at certifying a piece of software, it did give valuable insights into such a venture.

In Chapter five I explain my method of taking the particular case of object-oriented testing and showing how this can be made more rigorous using an object-oriented model of the system. In Chapter four I will examine state-of-the-art in the testing area..

Chapter 4

Testing state-of-the art

4.1 Introduction

Verification and validation (V&V) is one of the software engineering disciplines that help build quality into software. It is a collection of analysis and testing activities across the full life cycle as shown in **Figure 4.1.**

4.2 Software quality is defined by the IEEE *[IEE 89]* as :

(1) The totality of features and characteristics of a **software product** that bear on its ability to satisfy given needs; for example, conform to **specifications.**
(2) The degree to which software possesses a desired combination of attributes.
(3) The degree to which a customer or user perceives that software meets his or her composite expectations.
(4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

4.2.1 Verification :

(1) The process of determining whether or not the products of a given phase of the **software development cycle** fulfil the **requirements** established during the previous phase.
(2) Formal proof of program **correctness.**
(3) The act of reviewing, inspecting, **testing**, checking, auditing, or otherwise establishing and documenting whether or not items, **processes,** services, or **documents** conform to specified **requirements.**
Verification *[Fuj 89]* involves evaluating software during each life-cycle phase to ensure that it meets the requirements set forth in the previous phase.

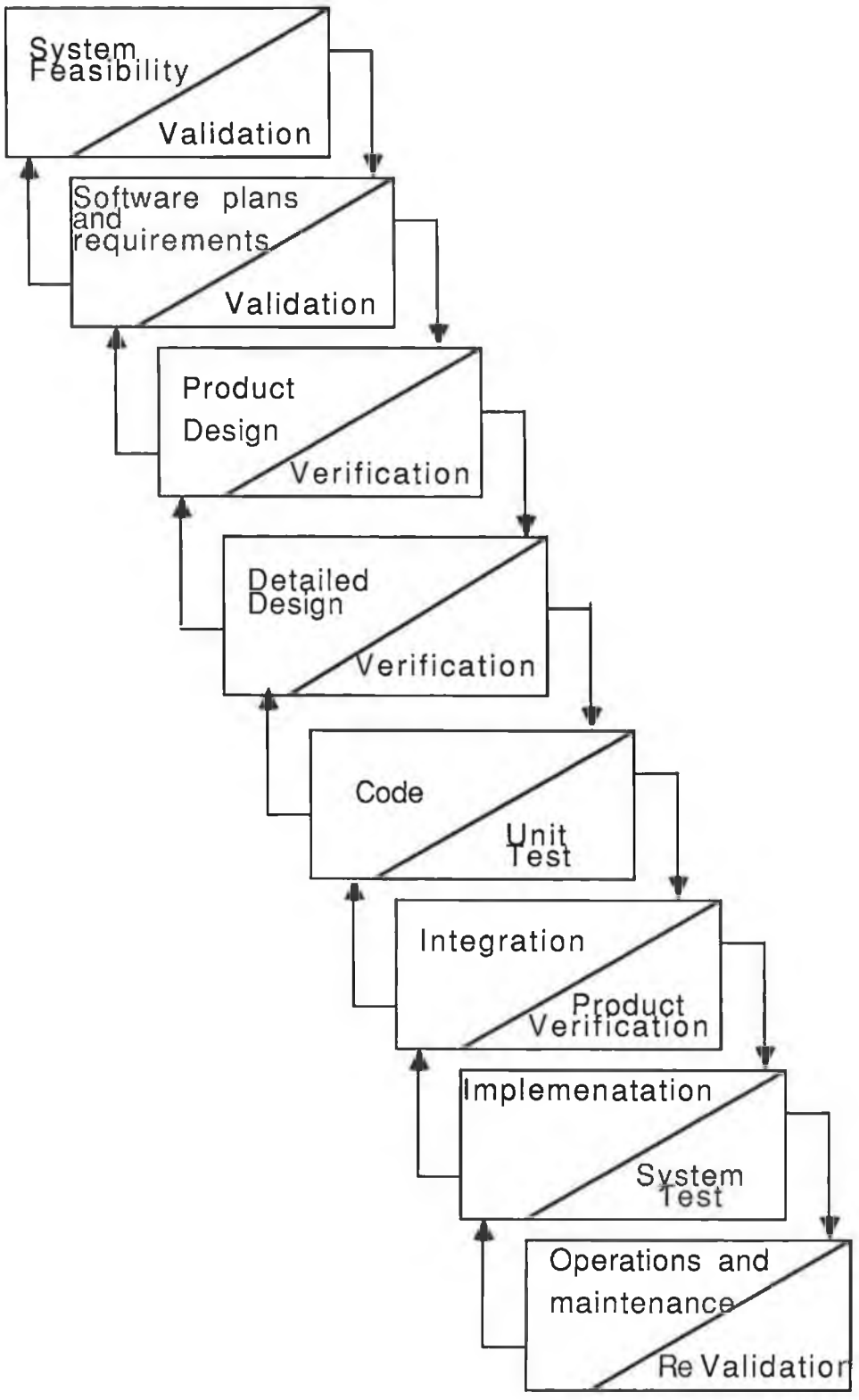

System Feasibility
Validation
Software plans and requirements
Validation
Product Design
Verification
Detailed Design
Verification
Code
Unit Test
Integration
Product Verification
Implemenatation
System Test
Operations and maintenance
Re Validation

Figure 4.1

4.2.2 Validation :

The process of evaluating **software** at the end of the **software development process** to ensure compliance with software requirements.

Validation involves testing software or its specification at the end of the development effort to ensure that it meets its requirements (that it does what it is supposed to do).

While "verification" and "validation" have separate definitions, it is possible to derive the maximum benefit by using them synergistically and treating "V&V" as an integrated definition.

V&V test activities span many phases, from requirements through installation. V&V testing continues into the operations and maintenance phase to address any changes made to the software after initial delivery. A comprehensive test-management approach recognises the differences in objectives and strategies of different types of testing. The main testing activities which are examined later in this Chapter - unit, integration, system and acceptance testing, all produce test-plan, test-design, test-case, and test-procedure documents.

In brief, unit testing verifies the design and implementation of software units or modules; Integration testing verifies functional requirements as the software units are integrated, directing attention to internal software and external hardware and operator interfaces; System testing validates the entire program against system requirements and performance objectives; Acceptance testing validates the software against V&V acceptance criteria, defining how the software should perform with other completed software and hardware. In addition to testing against system and software requirements, effective testing requires a comprehensive understanding of the system. Such an understanding is developed through systematically analysing the software's concept, requirements, design and code.

4.3 Static Analysis : The main aim of static analysis is to determine properties of the software product without actually executing the software. The software product (usually code or specification) is read in by a static analyser which produces output of three basic types.

- Analysis of the code.
- Numerical values describing some attribute of the code.
- Alternative representations of the product, (e.g. directed graphs LOGISCOPE, ASA *[Ver 90]*) or restructured code (SPADE *[PVL 90]*).

4.4 Formal technical reviews

A formal technical review is a software quality assurance activity that is carried out by software engineers. The objectives of a formal technical review are :

- to highlight errors in the function, logic or implementation of the software
- to verify that the software under test meets its requirements
- to assure that the software has been represented according to predefined standards
- to achieve software that is developed in a uniform manner
- to make software projects easier to manage

The formal technical review is actually a class of reviews which includes inspection techniques and walkthroughs.

4.4.1 Inspection techniques :

The inspection of software is a method of static testing to verify that software meets its requirements. It engages the developers and others in a formal process of investigation that usually detects more defects in the product (and at a lower cost) than does machine testing. The inspection process has been very successfully applied to areas such as software test plans, user documentation, high level design, system structure design. It suggests that virtually anything that is created by a development process, and that can be made visible and readable can be inspected *[Fag 86]*.

Software inspections are conducted by peers, and typically comprise three to six participants. The objective of a software inspection is to detect and identify software element defects. This is a rigorous, formal peer examination that does the following :
(1) verifies that the software element(s) satisfies its specifications
(2) verifies that the software element(s) conform to applicable standards
(3) identifies deviation from standards and specifications
(4) collects software engineering data (e.g. fault and effort data)
(5) does not examine alternatives or stylistic issues

4.4.2 Walkthroughs

The objective of a walkthrough is to evaluate a software element. Although long associated with code examinations, this process is also applicable to other software elements such as architectural design, detailed design, test plans/procedures, and change control procedures. The major objectives are to find defects, omissions, and contradictions; to improve the software element(s); and to consider alternative implementations.
Other important objectives of the walkthrough process include exchange of techniques and style variations, and education of the participants. A walkthrough may point out efficiency and readability problems in the code, modularity problems in the design or untestable design specifications.

4.5 Testing related definitions

These are some definitions related to the testing area *[IEE 89].*

Testing :
The process of exercising or evaluating a **system** or system **component** by manual or automated means to verify that it satisfies specified **requirements** or to identify differences between expected and actual results.

Test case :
A specific set of **test data** and associated **procedures** developed for a particular objective, such as to exercise a particular **program** path or to verify compliance with a specific **requirement**.

Test data :
Data developed to test a **system** or system **component**.

Test bed :
(1) A test environment containing the **hardware, instrumentation tools, simulators,** and other support **software** necessary for testing a system or system component.
(2) The repertoire of **test cases** necessary for **testing** a **system** or system **component.**

Test log :
A chronological record of all relevant details of testing activity.

Test plan :
A **document** prescribing the approach to be taken for intended **testing** activities. The plan typically identifies the items to be tested, the testing to be performed, test schedules, personnel requirements, reporting requirements, evaluation criteria, and any risks requiring contingency planning.

Test report :
A **document** describing the conduct and results of the **testing** carried out for a **system** or system **component.**

Test phase :
The period of time in the software life cycle during which the components of a software product are evaluated and integrated, and the software product is evaluated to determine whether or not requirements have been satisfied.

Test procedure :
Detailed instructions for the set up, operation, and evaluation of results for a given test.

Automated test generator :
A software tool that accepts as input a computer program and test criteria, generates test input data that meet these criteria, and sometimes determines the expected results.

Software reliability :

(1) The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to, and use of the system, as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered.

(2) The ability of a program to perform a required function under stated conditions for a stated period of time.

4.6 Testing in general

Testing is an activity where a program, or a module of a program, is subjected to a sequence of inputs and the outputs are compared to the expected outputs. Myers *[Mye 79]* states a number of rules that serve well as testing objectives :

- testing is a process of executing a program with the intent of finding errors
- a good test case is one that has a high probability of finding as yet undiscovered error
- a successful test case is one that uncovers an as yet undiscovered error.

If testing is carried out successfully, it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification and that performance requirements appear to have been met. Also, data which is collected during testing provides a good indication of software reliability, and some indication of software quality as a whole.

The testing process can be a costly, time consuming activity that can take up as much as 50% of development costs. In testing safety critical systems (e.g nuclear reactor monitoring, flight control) it can cost three to five times as much as all the other steps in the software engineering process combined.

Software testing is perhaps the most critical element of software quality assurance given that it represents the review of specification, design and coding. As in other stages of the software development process, it is also error prone. Testing can be used only to detect the presence of errors. It cannot be used to achieve the opposite - to prove conclusively their absence*[Rat 87]*.

4.6.1 Software Reliability and testing

Software reliability can be viewed as the probability that a given program will operate correctly in a specified environment for a specified duration. Several models have been proposed for estimating the reliability of programs. They can be broadly categorised into

software reliability growth models and statistical models.
Reliability growth models predict a program's reliability on the basis of its error history. Statistical models estimate it by determining the response (success/failure) of a program to a random sample of test cases, without correcting any errors which may be discovered during this process.
Of course testing is very closely related to the assessment of the reliability of programs, and in fact one important application of software reliability theory is to provide feedback regarding the amount of testing necessary in order to achieve a reliability objective *[Goe 85]*.

4.7 What is software testing ?

When testing, the objective would appear to be to enhance the quality of the product under test. Also, when carried out effectively in a commercial sense, software testing contributes to the delivery of higher quality software products, more satisfied users, and lower maintenance costs. From a scientific point of view, good testing will result in more accurate and reliable results. Inadequate, ineffective testing will naturally lead to low quality results, unhappy users, increased maintenance costs, unreliable, and inaccurate results. Software testing is an active process. If after performing a test, no errors have been revealed, then one should not passively assume that there are no errors present in the software - rather the testing process has failed to find the errors which one believes are present in the software *[Par 89]*.

4.7.1 Why Test ?

Software testing may in itself be an expensive process, but the cost of not testing is potentially a lot higher. In a safety critical system where human lives are at stake this is self evident. System economics isn't adequate for determining whether a product should be released or not in relation to safety critical systems
However, it is economics which are both the driving force and the limiting factor in testing. The most serious errors in a piece of software, are those which are not discovered but remain dormant when the system goes live.
Given the potential loss risks, be they commercial or life-threatening, it is obvious why software producers might want to enhance the quality of the delivered software. It is infeasible to exhaustively test all but the most simple piece of software. One must remember that although the aim of any software tester will be to highlight all errors, the reality is that there is no way of highlighting the errors that remain. No piece of software would ever be released on the market if the developers were pledged to certify that it was

totally free of errors.

4.7.2 When should testing be done ?

Traditionally software testing has centred on the testing of program code. However, errors are not made only in the final coding of the program - they can be made throughout the system development life-cycle. If testing is done only after coding is completed, then errors from as early as the requirements stage might well be embedded within the software. When testing is done only at this late stage, then the earlier in the life-cycle the original error was made, the more costly will be its removal. Testing should be a continuous process throughout the system development life-cycle.

4.7.3 Testing and Quality

Testing aims to improve the quality of the product under test. This is achieved through finding and removing errors, which if were present, would detract from the operational efficiency or accuracy of the product and thereby detract from the product's quality. In defining quality we can take the definition according to Crosby *[Cro 79]*.
Quality is conformance to requirements.
Other definitions include:
Quality is fitness for purpose.
Quality is fulfilment of requirements.
These definitions all have one thing in common - they relate the finished product back to its original design intentions. The testing process is an integral part of any quality programme. To establish whether a product conforms to its requirements, it must be tested against those requirements.

4.8 Testing techniques

Any engineered product (and most other things) can be tested in one of two ways:
(1) knowing the specified function that a product has been designed to perform, tests can be conducted to demonstrate that each function is fully operational;
(2) knowing the internal workings of a product, tests can be conducted to assure that "all gears mesh"; i.e. that the internal operation performs according to specification, and all internal components have been adequately exercised. The first approach is called black box testing and the second, white box testing *[Pre 88]*. Regardless of the testing technique employed the main objective is to derive a set of test cases that will uncover defects in the software.

4.9 Black Box Testing :

This form of testing is used to check conformance of the behaviour of the software with a reference such as requirements, system specification, or user manual. (Also referred to as functional testing or closed box testing). The term black box is used, as the program (or the functions exercised by the program) are treated as a black box whose functionality is determined by observing the outputs to corresponding inputs.

The aim of black box testing is to test a program from its specification. No reference is made to the internals of the program i.e. the software code. **See Figure 4.2.** The tester is provided with the specification of the software component under test. This is then used to derive appropriate test cases. Black box testing is based on a specification of all possible input to, and output from the software under test, as well as a description of the processing of the data. An example of pure black box testing is with regard to the functional parts of the acceptance test, where the end user tries out the features and functions of the overall system.

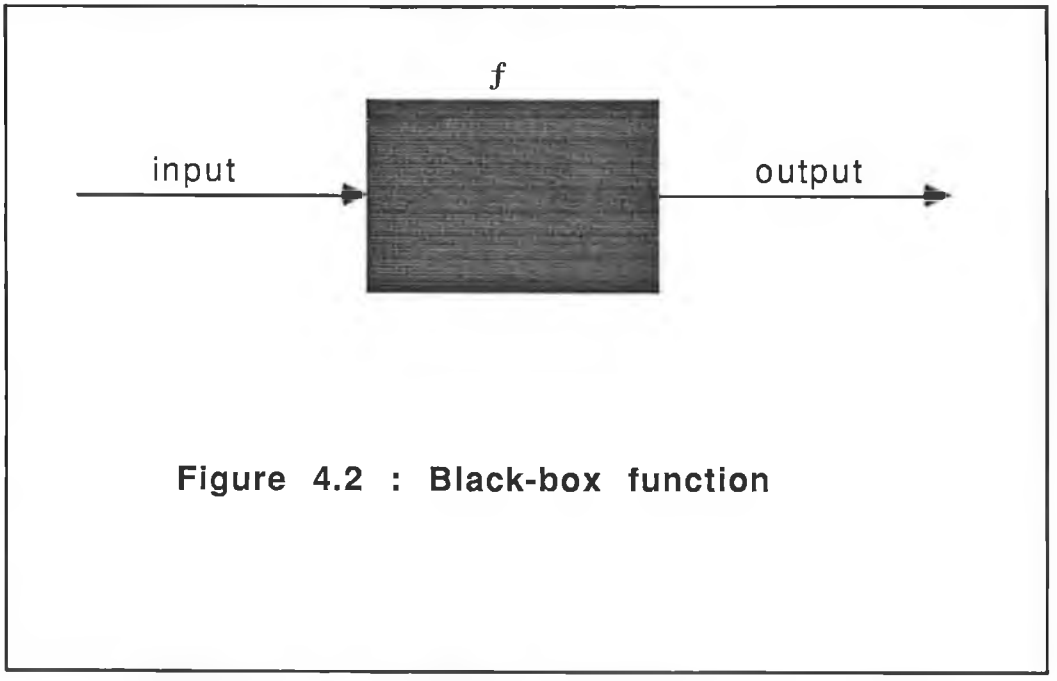


Figure 4.2 : Black-box function

There are advantages with black box testing as no knowledge about the internal software is needed so the tester doesn't have to examine the software structure. Therefore, less effort is required compared to other testing techniques.

There are disadvantages in the technique such that the results gained are purely qualitative.

(i.e. no qualitative statement about failure behaviour or a degree of correctness can be given). One still cannot get clues from the program about which test inputs exercise the program the best.
Black box testing is complementary to all white box verification methods.

4.9.1 Equivalence class Partitioning :

In attempting to minimise the number of test cases, the input space of a program is partitioned into equivalence classes with respect to the program's input specifications *[Mye 79], [Red 83].* An ideal test case will uncover a class of errors that might otherwise require execution of many cases before the general error is observed.
Test case design for equivalence class partitioning is based on an evaluation classes for an input condition. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition (e.g. yes or no). Equivalence classes can be defined according to the following :
(1) If an input is Boolean, then one valid class and one invalid class are defined.
(2) If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
(3) If an input condition specifies a member of a set, one valid equivalence class and one invalid equivalence class are defined.
(4) If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.

4.9.2 Boundary value Analysis :

Boundary value analysis is a test case design technique that complements equivalence class partitioning. This approach to black box testing involves choosing test cases that lie on or near the boundaries of an input domain. For some undefined reasons a larger proportion of errors tend to occur at the boundaries of the input domain, rather than at the centre.
The following are the guide-lines applied in selecting test cases for this technique:

(1) If an input condition specifies a range bounded by values **x** and **y**, then test cases should be designed with values at **x** and **y**, just above and just below **x** and **y.**

(2) If an input condition specifies a number of values, test cases should be developed that exercise the maximum and minimum numbers. Values just above and below the maximum and minimum are also tested.

(3) If a data structure within a program has a prescribed boundary, then a test case that exercises the structure at its boundary should be applied.

(4) Boundary value analysis also derives test cases from the output domain. *(1)* and *(2)* are applied to output conditions.

4.9.3 Data validation testing :

This technique uses the idea of filling in the gaps left by other black box testing methods. It is similar to other methods in that it is heuristically based (i.e. a set of guide-lines are provided to assist in testing, but no formal analysis is used). An example of this technique is when commands are typed using a keyboard, then the following data validation tests would be appropriate.

(1) Generate a system interrupt immediately after a command has been entered
(2) Omit all commands (i.e. just carriage return)
(3) Type in a partially correct command and then terminate the command entry
(4) Provide syntactically correct input that is out of sequence or specified at the wrong time
(5) Specify commands with the incorrect syntax
(6) Give correct commands, but with too much qualifying data

4.9.4 Random testing :

Here the program is tested by randomly selecting some subset of all possible input values. There has been strong disagreement about its usefulness in the testing process *[Mye 79]*.

4.9.5 Cause effect graphing :

This technique provides a concise representation of logical conditions and corresponding actions *[Elm 73]*. It involves analysing the program specification to determine its affect on various types of input. The technique involves four steps :

(1) Causes (input conditions) and effects (actions) are listed for a module, and an identifier is assigned to each
(2) A cause-effect graph is developed
(3) The graph is converted to a decision table
(4) Decision table rules are converted to test cases

Cause effect graphing is useful for exercising combinations of conditions. As in equivalence class partitioning, the identification of the causes from the specification is far from straightforward *[Par 89]*.

4.10 White box testing :

White box tests are defined taking into account the internal structure of the program being tested and, in addition to the comparison of achieved and expected outputs.(Also referred to as structural testing, glass-box testing,open box testing). White box testing has received the most attention in terms of testing research, and there are a number of methods associated with it - far more than with black box testing.

The main aim of white box testing is to "cover" or "exercise" the code with a degree of thoroughness (e.g. every statement is executed). Initially it may not appear obvious why white box testing is necessary given that black box testing checks the specification against the implementation. Why more testing ? There may be parts of the software which are not fully exercised through the black box functional tests.

4.10.1 Statement testing

This method of white box testing requires test data to be constructed which causes each statement in the code to be executed

4.10.2 Arc and Path testing

The starting point in this form of white box testing is to derive a program flow graph which represents a program in terms of arcs (showing flow of control) and nodes (representing decisions). Arc (or branch) testing requires the construction of test data to cause each arc (a point in the program where a decision is made, such as an "if" statement or "while" condition) to have a true and a false outcome.A path is defined as consisting of a number of arcs *[Chu 87]*.

(1) Given the flow graph representation of the following constructs as shown in

Figure 4.3

Sequence

If

Until

While

(2) Assuming that GOTO statements are not used in a program, it is a straightforward process to derive the flow graph for any program by substituting these representations for program statements.

(3) An exhaustive white box test would involve all possible combinations of all arcs. Apart from the most simple of software structures this is an impossibility. So the objective of

structural testing can be reworded to : not to test all possible arc combinations but to make sure that all possible arcs are traversed at least once.

(4) The number of independent paths in a program can be ascertained by applying the **Cyclomatic complexity,***[McC 76]* to the program flow graph. An independent path is defined as one which traverses at least one new arc in the flow graph. In programming terminology this means exercising one or more new conditions. When used in the context of path testing, the value computed for cyclomatic complexity also identifies the upper bound for the number of tests that must be conducted to assure that all statements have been executed at least once.

The cyclomatic complexity V(G) is defined as :

NUMBER OF ARCS - NUMBER OF NODES + 1

The cyclomatic complexity is termed a software metric that provides a quantitative measure of the logical complexity of a program. It sets an upper bound on the number of test cases that must be designed and executed to guarantee coverage of all program statements *[Li 87],[McC 89].*

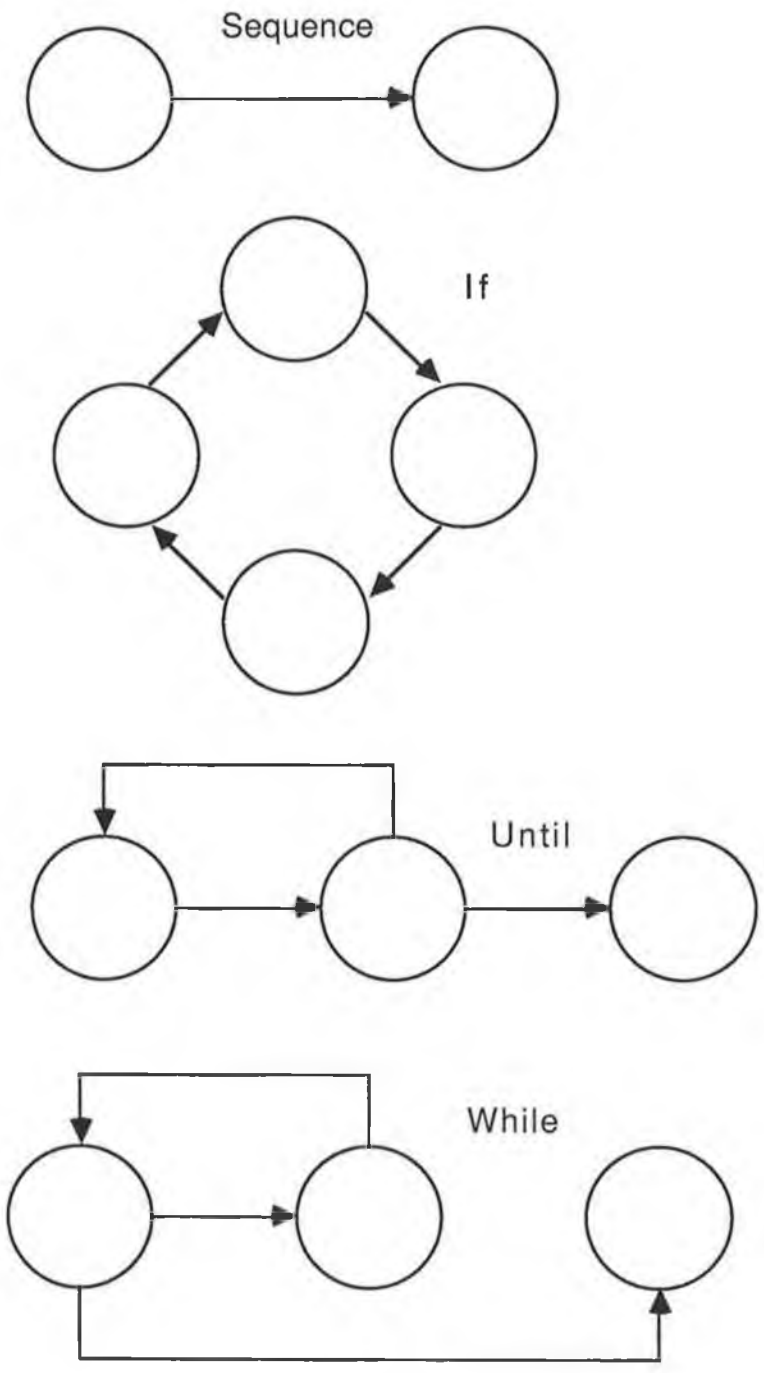


Figure 4.3 : Flow graph representations

Deriving test cases using McCabe's metric

(1) Develop flow graph.

(2) Determine the cyclomatic complexity V(G).

(3) Determine the set of linearly independent paths. The set of V(G) gives the number of linearly independent paths through the program control structure.

(4) Develop test cases that will exercise each independent path. It is important to choose test data so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is then executed and compared to expected results. Once all test cases have been exercised, it is important to check if all statements in the program have been executed at least once.

4.10.3 Data flow testing

So far we have looked at statement, arc and path testing, which require that the test data cause every node, arc or path in the program's flow graph to be executed. According to Frankl and Weyuker, *[Fra 88]* statement and arc testing can fail to expose many common errors, and path testing is usually infeasible since programs with loops have infinitely many paths. A number of test data adequacy criteria which are based on data flow (DF) analysis, some of which "bridge the gap" between arc testing and path testing have been proposed. These criteria are based around the idea that there should be little confidence in a system which believes that a variable has been assigned the correct value at some point in the program, if no test data causes the execution of a path from the assignment to a point where the variable's value is subsequently used.

A family of test data data adequacy criteria, based on the analysis of the DF characteristics of the program being tested is described in *[Rap 82]*. These criteria, termed data flow testing criteria were originally defined for a fundamental programming language having simply

- assignment statements
- conditional and unconditional statements
- I/O statements.

They require that the test data exercise certain paths from a point in a program where a variable is defined to points where the variable is subsequently used. There is a tool ASSET *[NYU 90]* which performs DF testing on programs written for such a language.

4.11 Testing Systems

Black box and white box testing techniques in terms of individual computer programs have been examined so far. This is related to unit testing which is defined in the next section. It is unreasonable to try and test systems as a single unit except in the case of small computer programs. Software systems are constructed in terms of sub systems; subsystems are in turn comprised of modules which are made out of procedures. The testing process is akin to the software coding process in many ways in that it should progress in stages. The stages should in turn logically proceed from the previous stage. **See Figure 4.4** which identifies one possible testing implementation. In Chapter five there is a discussion on object-oriented unit testing which is based on the area of object-oriented design.

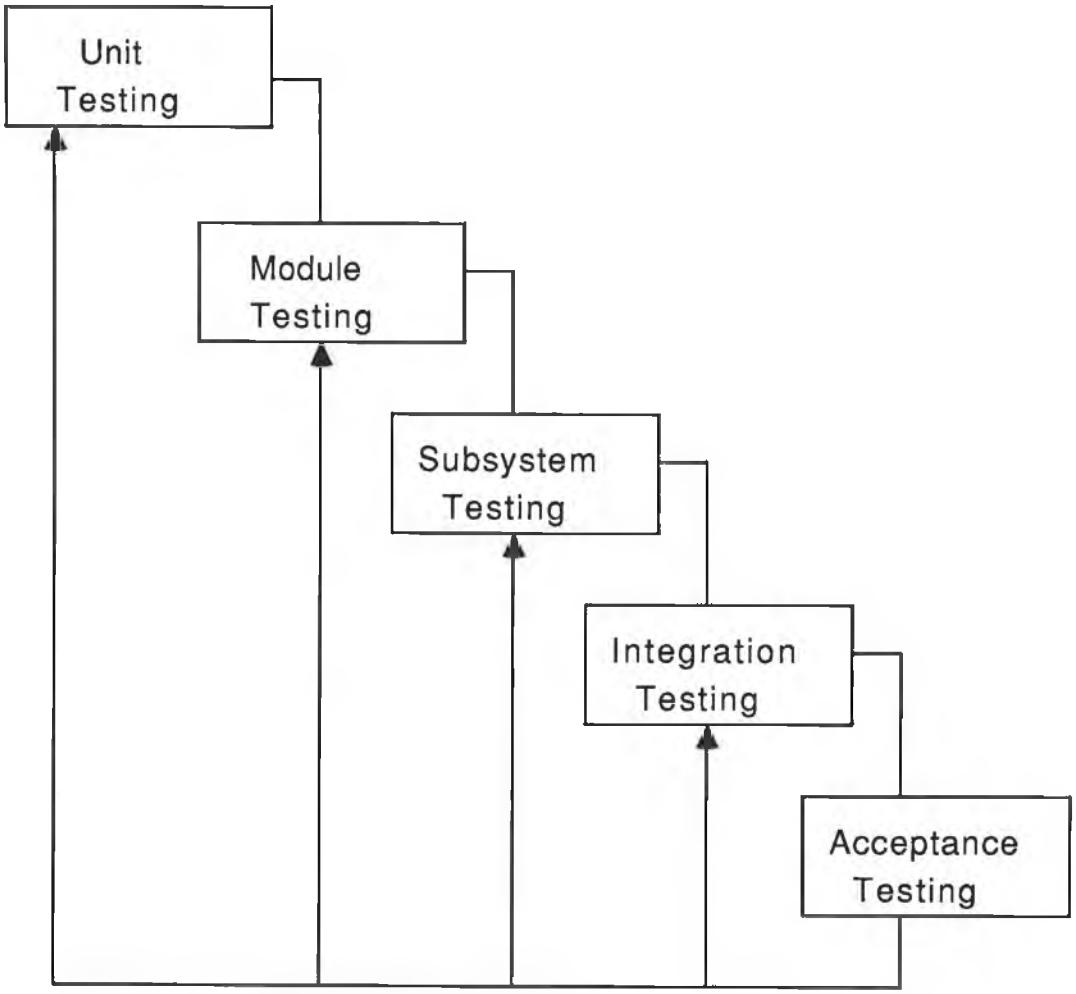


Figure 4.4 : Stages of testing

4.11.1 The testing process

Unit testing :

This is the basic level of testing where individual components (e.g. functions or objects) are tested to ensure that they operate correctly. Unit testing demands that each component is independent of all other system components during testing.

Module testing :

After each program unit has been tested then module testing can take place. It should be possible to test a module as an independent entity, without the presence of any other system modules.

Subsystem testing :

This is where modules are integrated together to form subsystems. This stage of testing should concentrate on detecting interface errors.

Integration testing :

Integration takes place when the subsystems are merged together to make up the whole system. This stage focuses on finding errors which result from unanticipated interactions between subsystems and components. This looks on the functional performance of the entire system and considers the system itself a black box.

Top-down and Bottom-up testing

Assume we have a tree structure which represents the relationships between subsystems. See **Figure 4.5** which demonstrates that subsystem 1 calls subsystems 2, 3, 4 but is not called by any subsystem itself.

Top-down testing :

In this type of testing subsystems are examined starting from the top of the tree and then progressively work through to the subsystems at the bottom of the tree. See **Figure 4.5**.

Bottom-up testing :

Bottom-up reverses the process of top-down. The components making up a subsystem are tested individually. Then they are integrated to form a subsystem and this is subsequently tested. This form of testing is more appropriate during the latter stages of the software life-cycle when all subsystems are available. See **Figure 4.5**.

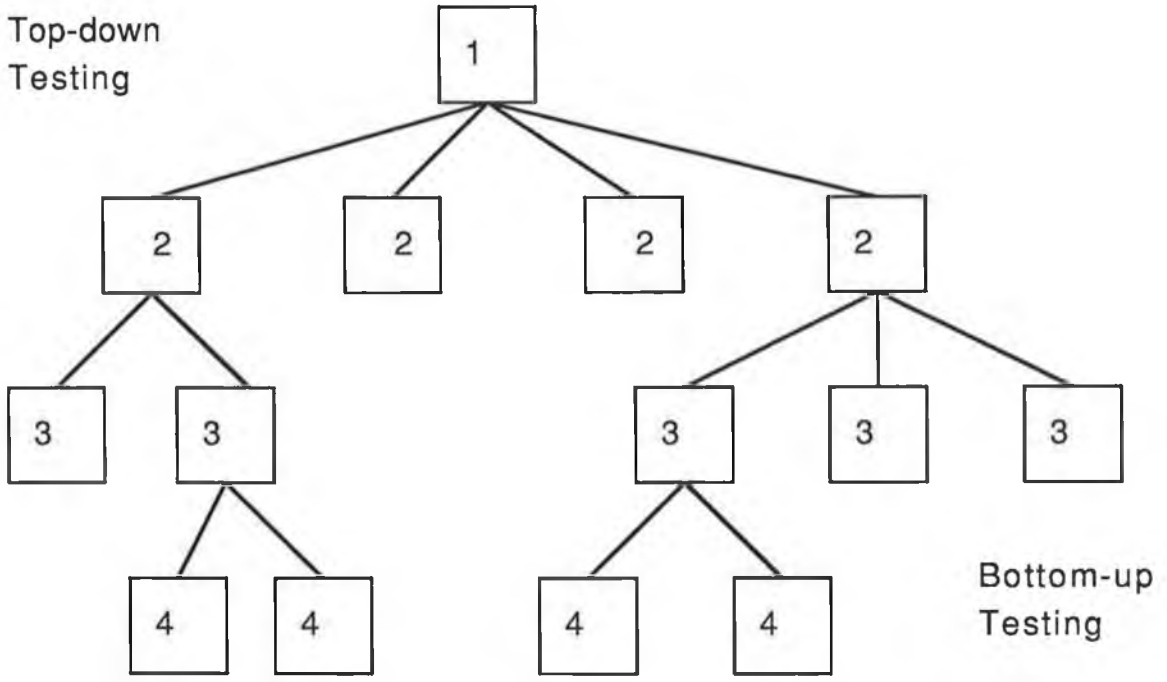


Figure 4.5

Incremental testing :

Incremental testing introduces one subsystem at a time. The system starts off as one subsystem and this is tested in an appropriate way. If this subsystem is deemed satisfactory then another subsystem is introduced and further testing is administered. The process continues until all subsystems are integrated into the complete system. The rationale behind this form of testing is that if a subsystem introduced produces errors in the system, then the likelihood is that these errors are attributable to the latest subsystem added.

Acceptance testing :

Acceptance testing concentrates on the readiness of a system for full installation. Three levels of testing can be applied at this point. The first is known as alpha testing. The user identifies the functions to be tested via requirements/design/code documents. Beta testing involves the role of the analyst and the use of tools. The goal of beta testing is identical to that of alpha testing but tools are used to check to see if all functions have been tested. Gamma testing is at the highest level. This uses tools of two kinds. The first kind of tool determines if all functions have been tested and the second kind checks to see if all expressions are tested over expression fault revealing tests.

4.12 Test planning

As testing is such an important stage, a test plan may be introduced. Usually test planning will be concerned with laying out standards for the testing process rather than describing the actual testing.

- The system should be explicitly specified
- Test cases should be derived at the design and implementation stage
- Set out the resources required for testing, the costs of testing,the scheduling of the tests.

If only a small part of the system is incomplete then system testing cannot start.

A test plan might take the form of a test case folio *[Het 88]*.

The following should be identified :

Test group/test case
Testing objectives
Test data
Expected results
How will the test be conducted ?
Who will administer the test ?
Actual results

4.13 Types of test cases

Figure 4.6 shows the different types of test cases.

Type	Source
Requirements based	Specification
Design based	Logical System
Code based	Data structures and code
Randomized	Random generator
Extracted	Existing files or test cases
Extreme	Limits and boundary conditions

Figure 4.6

Requirements based test cases

The most direct source of test cases is from the functional specification of what the program is supposed to do. The specification details how the program should behave, what inputs it accepts, what processing it achieves, and the outputs it produces. E.g. If the specification states that the program produces four different kinds of report then it is necessary to derive test cases that will will produce all four.

Design and code based test cases

Design based test cases are based upon studying the software design and identifying areas or features that are not adequately covered by the requirements based tests. The reason is to supplement the requirements based tests and to cover the design interfaces and paths within the software. Code based test cases are based on the actual software code and data structures such that every program statement is exercised at least once and every decision is exercised over all outcomes.

Extracted and randomized

These techniques both produce large volumes of data quite easily, but two limiting constraints subsist; notably that it is difficult to produce expected data results for all the test data and it is also takes a lot of time to review all of the data produced.

Extreme

It is well known within the resting area that the "best" test cases are those that have the highest probability of finding faults if they are present. Exceptional conditions, extremes, boundary conditions, and abnormal cases all make superb test cases because they represent error-prone areas *[Het 88]*.

4.14 Tools for Testing

Tools have evolved rapidly during the last decade. Extensive development has taken place both in the research laboratories and the commercial marketplace, as witnessed by the richness and variety of tools available.

Ever since the first compilers, editors and debuggers were developed, tools have played an important role towards improving productivity. As the product development stage became more complex, new tools were developed to handle additional tasks. Eventually, the application of tools spanned the entire life-cycle.

Software is now viewed as the set of objects developed during all phases of the project life-cycle. This view has evolved over two stages. :

(1) the greater awareness of the importance of the analysis and design phases developed during the 1970s

and

(2) the development of tools in the 1980s to support these activities.

The first stage started when developers realised that the analysis and design phases offered strong leverage for the prevention and detection of errors. The second stage started when early CASE(Computer-aided software engineering) tools provided analysis and design work benches to help automate the front end of the development process.
Presently, we are entering the third stage where the application of tools is broadening and deepening throughout the life-cycle, and where tools have become more refined and integrated.

4.15 Tools supporting testing

Tools address the different aspects of the testing approach:

(1) They provide a controlled environment in which testing can take place. If the target system is simple, it is often possible to simulate it as a more powerful host, thus gaining access to a richer set of tools. (e.g. RUTE *[GEE 90]*) by simulating the final execution environment as a way of expediting test execution.

(2) They are used for test data selection. A decision is needed on which combinations of input values will most thoroughly exercise the system and will most likely uncover defects. (e.g. T *[PPE 90]*) which uses heuristics and models of common defects to generate test cases. This testing tool also addresses the notion of minimising the testing overhead (especially the cost of redundant testing).

(3) They perform the testing through capturing and organising the resulting output (e.g. Xray/DX *[MRH 90]*). Most such tools work on the final executable code, monitoring its operation for conformance to specifications.

4.16 Types of tools

Tools that can reduce test time are always going to be valuable in the area of software development. Although current usage of automated tools is limited in terms of software testing , it is highly probable (especially with the emergence of CASE) that their use in applications will be increased somewhat in the future and that they won't remain just as

research vehicles. Tools can be classified into two groups according to the analysis they perform: static analysis tools and dynamic analysis tools. There is also another family of test related tools that neither perform direct test nor use any specific testing technique. These are called test support tools *[Lut 90]*.

4.16.1 Static Analysis Tools

These tools focus on requirements/design documents and on structural aspects of programs i.e. those characteristics of a program that can be discerned without actually executing it. Static analysis tools analyse program structure characteristics without regard to the execution of the program that is being tested. Static analysis of programs may include a combination of some of the functions listed below:

Code Auditing

This refers to the examination of source code to determine whether or not specified programming standards have been followed. Typical standards include adhering to structured design and coding, or using a standard coding format.

Consistency Checking

A consistency check determines whether or not units of program text are internally consistent in the sense that they implement a uniform notation or terminology and are consistent with a specification.

I/O Specification Analysis

This method generates test input data through analysis of I/O specifications.

Data Flow Analysis

This consists of the graphical analysis of collections of (sequential) data definition and reference patterns to determine constraints which can be placed on data values at various points of execution of the source program.

4.16.2 Dynamic Analysis Tools

These tools support the testing process by directly executing the program under test. They produce certain information on the executed program (e.g which parts of the program are being executed the most). Some of the tools in this category are :

(1) Symbolic evaluators which accept symbolic values and execute them according to the expression in which they appear in a program. They are used to support test data generation, path analysis, and detection of data flow anomalies (ATTEST).

(2) Test data generators assist a user in generating test data. There are three different kinds of test data generators - pathwise test data generators, data specification systems and random test data generators.

4.16.3 Test Support Tools

They function by simulating an environment for running module tests. They provide a standard notation for specifying test cases and automating the test process. Some systems compare the actual output with the expected output and highlight any discrepancies *[DeM 88]*.

Test harness

These tools are also known as automatic test drivers or testbeds. A test harness can be used to apply a selected set of test cases to a program and to verify the correctness of the resulting test output. A harness requires that the user constructs a set of test cases. Each test case should provide values for input variables and describe the expected values of output variables. Sophisticated test harnesses allow a user to start up a system at some intermediate point of execution and to test values at intermediate points *[How 87]*.

Comparators

This kind of tool compares two versions of data to identify the differences between the two versions. The data may be program code, output of an execution, or data files. Comparators serve primarily as tools for validating modified software to assure that the revised software contains only particular modifications. The use of a comparator helps limit the scope of reverification that must be performed on modified programs (e.g. DIFFS *[SCS 90]*).

Chapter 5

Object-oriented black box testing

Introduction

In this chapter I further examine the Tango case study and use it towards a way of developing my object-oriented testing approach.

5.1 Functional Testing

Functional testing is a process of verifying that the functions of a system are present as specified.

5.1.1 Functions

According to Howden *[How 87]*, mathematically all functions are of the form $f : a \longrightarrow b$ where f is a function which, for every object of type a, returns an object of type b. In the simplest cases, types are sets of simple objects, such as integers or reals, and in more complex cases they are structured objects having different components.

A function transforms an object of one type into an object of another type. An object can be defined as data of type real, integer or it could be even structured in the form of an array, file etc.,. Data types of the same kind are those that have the same properties and which allow the same set of operations to be applied to them. Examples of common data types are integers and strings of characters. The same set of operations such as add, divide, multiply, and subtract can be applied to integers.

5.1.2 Software development and functions

Functions and their associated data are the basic conceptual units that are used to build software. They have many uses, not only in software programs but at the requirements and design levels they can also be useful. They may be both formally and informally defined.

5.1.3 Variables and data structures

Different programming languages contain different kinds of built-in data types and data structures. Variables and data structures can be looked at in two ways. One way is as denoting objects of a certain type, and the other way is as storage structures for holding

objects of the appropriate type. Sometimes, a variable can be used for several different types of data.

5.1.4 States

States are associated with variables and data structures in programs. Variables and data structures can be in different states depending on the types of data they contain. Some states are history states, in the sense that they denote what has been done to the data in a data structure. An example of this is if it was necessary to check if the student number in a new student record (that was read into a data structure) is the same as the old student number. The student record is in the state unchecked before the check, and after the new record has been read in. Afterwards it is in the checked state. Generally, the state of a system or program is associated with the contents of all its variables and data structures. State assertions describe the relationships between data. State assertions and functions have a complementary relationship. Functions transform one program state into another and a program can be described in terms of either the sequences of state transforming computations it performs, or the sequences of states it goes through during its sequences of computations.

5.1.5 Faults

Functional faults relate to incorrect expressions, and incorrect input/output behaviour is the end result.Structural faults relate to incorrect program or module structure and incorrect function sequences or function interactions is the result.

5.1.6 Functional testing theory

There is a mathematical model which states for every function f written by a programmer, there exists a correct version of that program f^*. When testing a function one uses selected input and the result is compared to the expected correct output (the output that would be produced by f^*). This theory assumes the existence of an input/output oracle which, for an input x for the function f and output from the function $y = f(x)$, it is possible to determine if $y = f^*(x)$.

5.2 Functional Testing of Tango

The objective is to (black box) functionally test Tango. Tango documentation is informal consisting of a user oriented functional specification. From this documentation, a rigorous model of claimed functionality was constructed and is described in chapter three. The first question that must be answered if Tango is to be functionally tested is :
What are the functions of Tango ? The documentation doesn't consist of any requirements or design information, and the view of the system is strictly an object-oriented via the rigorous model.

5.2.1 Trapping the functionality of Tango

Identifying functions to be tested

Assume that one operation is comprised of many separate functions. i.e.

$$o_i = \{f_{i1}, f_{i2}, \ldots\ldots f_{im}\}$$

Each function within an operation can be defined separately as :

$$f_{i1} : d_{i1} \longrightarrow d_{i1}'$$
$$f_{i2} : d_{i2} \longrightarrow d_{i2}'$$
$$f_{im} : d_{im} \longrightarrow d_{im}'$$

where f_{im} is a function m belonging to operation i, which for every data type d_{im} returns a data type d_{im}'.

The total number of operations in the system is given by

$$\sum_{k=1}^{n} X_k \qquad (X = 1.....N)$$

X : object

N = number of objects

n = number of operations on each object

The number of functions in the system can then be expressed as :

$$\sum_{k=1}^{n} \sum_{j=1}^{m} X_{kj} \qquad (X = 1.....N)$$

N = number of objects

m = number of functions within an operation

n = number of operations on each object

The change in data for an object **X** after applying one operation is equivalent to applying a series of functions. These functions are embedded in the software and they may be explicit or implicit. However, there is no 'direct access' to these functions. Functional testing implies that the code implementing these functions are tested; in theory functional testing cannot be achieved Knowledge is limited, in that one only knows that the functions exist so the closest one can get at a function is via an object operation. This leads into object-oriented testing which is examined later in the chapter.

5.2.2 Applying functional testing theory to TANGO

Given the Tango functionality TANGOF and assuming there is a version of Tango called TANGOF* which is functionally correct. The total input data is X and the output data is Y. Let $X = \{x_1, x_2, ...x_n\}$ and $Y = \{y_1, y_2, ...y_n\}$.

Therefore, y_n = TANGOF(x_n)

y_n = TANGOF*(x_n)

For every object operation O_i represented by a number of functions $\{f_{i1}, f_{i2}, ...f_{im}\}$ there exists a correct version of that operation O_i^*. When testing an operation, selected test data is chosen, and the result of applying that operation is compared to it's expected correct output.Given an input x for the operation O_i and output from the operation $y = O_i(x)$, it is possible to determine $y = Oi^*(x)$.

5.3 Object-oriented unit testing

Although object-oriented environments are being used much more frequently in software development, there has been little attention focused on object-oriented testing. Here is a brief explanation of unit testing on modules developed with an object-oriented language *[Fie 89]*.

5.3.1 Objects

An object is the basic building block of of an object-oriented environment, and it is used to model some entity in an application. An object is composed of data and methods. The data constitutes the information in the object. The methods, which are the same as procedures and functions in procedural languages such as Pascal or C, manipulate the data. Generally, objects of the same kind or class usually exist in most applications. Each object in an object-oriented environment is an instance of some class.

5.3.2 When should testing begin ?

In a procedural language such as Cobol or Pascal, a complete unit might not exist until several functions or procedures are implemented. However, in an object-oriented environment, once a class has been defined and coded, then it can be considered a complete unit, and ready for use by other modules in the system. This really means that unit testing must be considered a lot earlier in an object-oriented environment.

5.3.3 What testing techniques should be used ?

Since the paradigm surrounding object-oriented programming emphasises the external behaviour of data abstractions rather than the internals, one would expect to employ black box, functional testing techniques. However, a more thorough white box testing technique is actually needed.

5.3.4 What should be tested ?

Ideally one would like to completely path test all classes of objects, particularly for critical application systems. However, the resources needed to achieve this may be quite extensive, and in working towards this, tradeoffs are likely to be made. But the testing process can be refined somewhat and a minimum set of test cases can be derived.

5.4 Object-oriented unit testing of Tango

The rigorous model described in chapter three captures an object view of the system and not a functional one. Initially object oriented unit testing was used to validate the rigorous model. This form of testing highlighted any potential anomalies or discrepancies in the model as well as showing some object operations provided by the software but not mentioned in the documentation. Each object class was individually exercised against the actual functionality of Tango as depicted in **Figure 5.1.** Generally, the functionality present in the software matched the claims made in the product description. However some significant discrepancies were uncovered between the documents and actual product behaviour. The software includes a number of extra un-documented functions. Also, some of the more important object operations depend on pre-conditions which are not documented. The functionality of some of the operations is not clear. The complete set of results are shown.

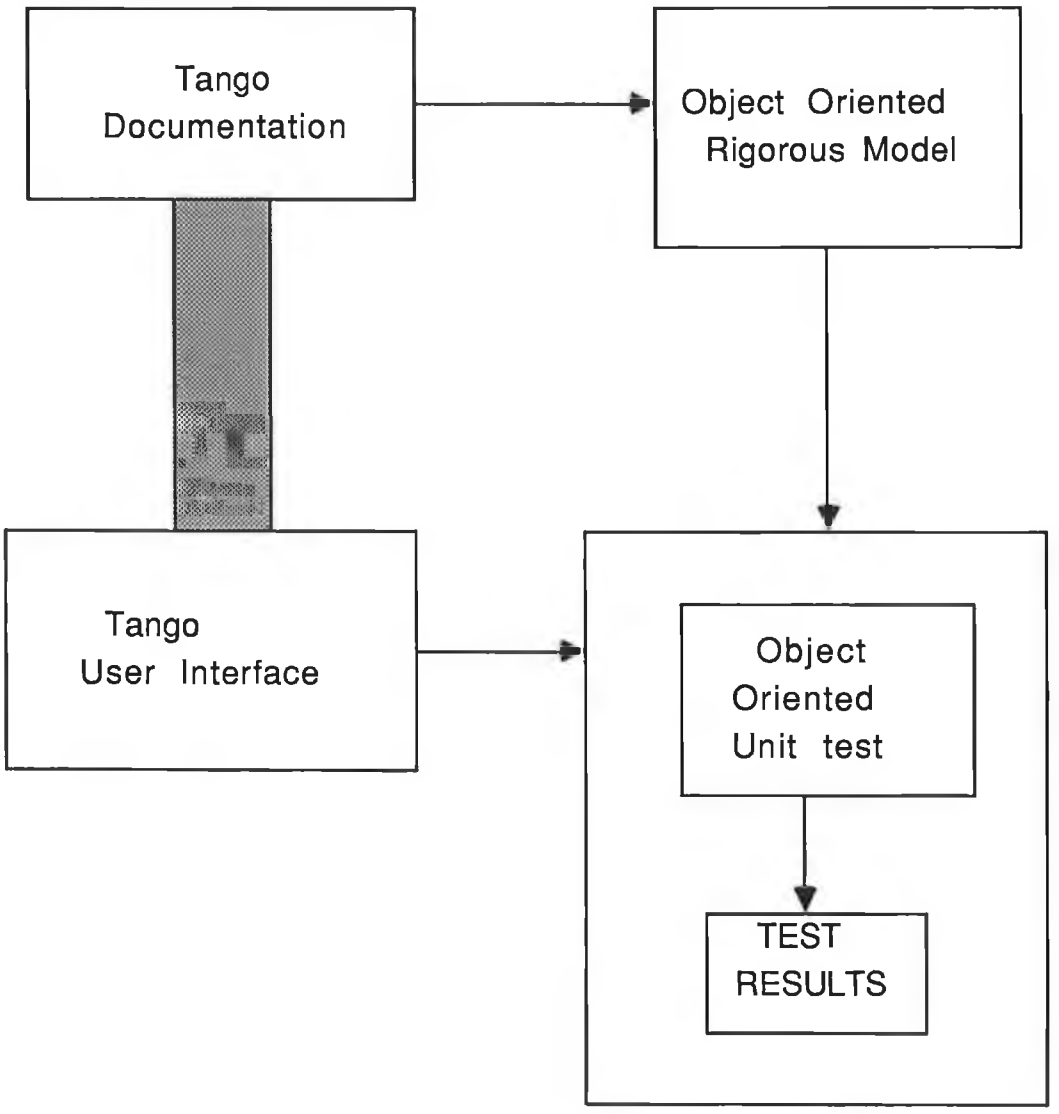

Tango
Documentation
Object Oriented
Rigorous Model
Tango
User Interface
Object
Oriented
Unit test
TEST
RESULTS

Figure 5.1

5.4.1 Object-oriented test results

Object :
Patient

There are extra operations 'Make a patient record' and 'Delete a patient record'

Object :
Patient Appointment

A pre-condition for the 'Modify' operation is that there does not exist a patient appointment on file which was cancelled previously;

Pre-conditions for the 'Cancel' operation should be that there does not exist a cancelled patient appointment and an admitted patient;

Object sub-types :
New Patient Appointment
Returning Patient Appointment

To 'Make a new patient appointment' or to 'Make a returning patient appointment' a pre-condition is to 'Make a patient record' for that patient.

Object :
Test

No operations identified.

Object :
Patient Test Results

The add and amend operations are not made explicit. Only 'Add ' exists which may or may not involve an 'Amend'.

Object :
Configuration Parameter
Object sub-types :
Screening Type
Screening Doctor
Screening Nurse
Physical Examination default text
Constant

Physical Examination default text has undocumented attributes 'Number' and 'Area code' along with operations 'Add', 'Change', 'Delete', and 'Display'.

Object :
Output
Object sub-types :
Label
Report
Invoice
Letter

The amend operation on some of the sub-types (**Patient Letter**) and (**G.P. Letter**) is not provided in Tango. There is some extra **Label** output produced consisting of individual or batch '**Bill**' labels. Tango also produces an individual patient '**Archive**' report.

5.5 Test Cases

Test cases are input and output specifications along with a statement of the function under test.
I have decided on a test case form consisting of
(1) **Test Case i.d.** : identification of the test
(2) **Testing objectives :** what the test achieves
(3) **Input :** the test data that drives the test
(4) **Expected Output :** what the anticipated result is
(5) **Actual Output :** the actual result
(6) **Results :** Comparison of *(4)* and *(5)* and what action, if any is necessary.

5.6 Generating test cases

In an ideal situation, it would be desirable to generate test cases to cover every possible input and every possible permutation of situations the system could ever face; the system would then be tested exhaustively to ensure that its behaviour would be error free. A slight problem with this is that it doesn't work. The number of test cases for a typical large complex system is massive, reaching a figure of the order 10^{100} test cases. With this figure and conducting a test case every millisecond, it would take longer than the projected age of the universe to exercise these test cases fully. As a consequence, nobody carries out truly exhaustive tests on anything other than the most trivial system; the best one can hope to do is to generate test cases that will exercise (or cover) a large percentage of the different decision paths that the system can take.

5.7 Counting the number of test cases

Chusho *[Chu 89]* has a method of counting the minimum number of test cases. He proposes a systematic test case generation method for functional testing called AGENT (automated generation method of test cases). AGENT consists of the the following components:

- a functional diagram (FD) which formally expresses the functional specification of a program
- a mechanical procedure for generating test cases from the FD. An FD model is composed of a state transition model and a Boolean function model.

5.7.1 Test case form

A test case in this instance constitutes a sequence of states passed through in testing and a pair of conditions in each state which must be satisfied by the input and output data. In generating test cases from the FD, Chusho proceeds to say that it is important that the number of test cases is practical, and that the criteria for test case generation is clear.

5.7.2 Test case notation

A CST (condition structure tree) is developed using the notation in **Figure 5.2** for the sequencing that can apply to state transitions.

(1) Certain state transitions will follow in sequence (AND/Sequence)
(2) Other state transitions will be selected on a choice basis (OR/Selection)
(3) Some state transitions will be repeated (Iteration)

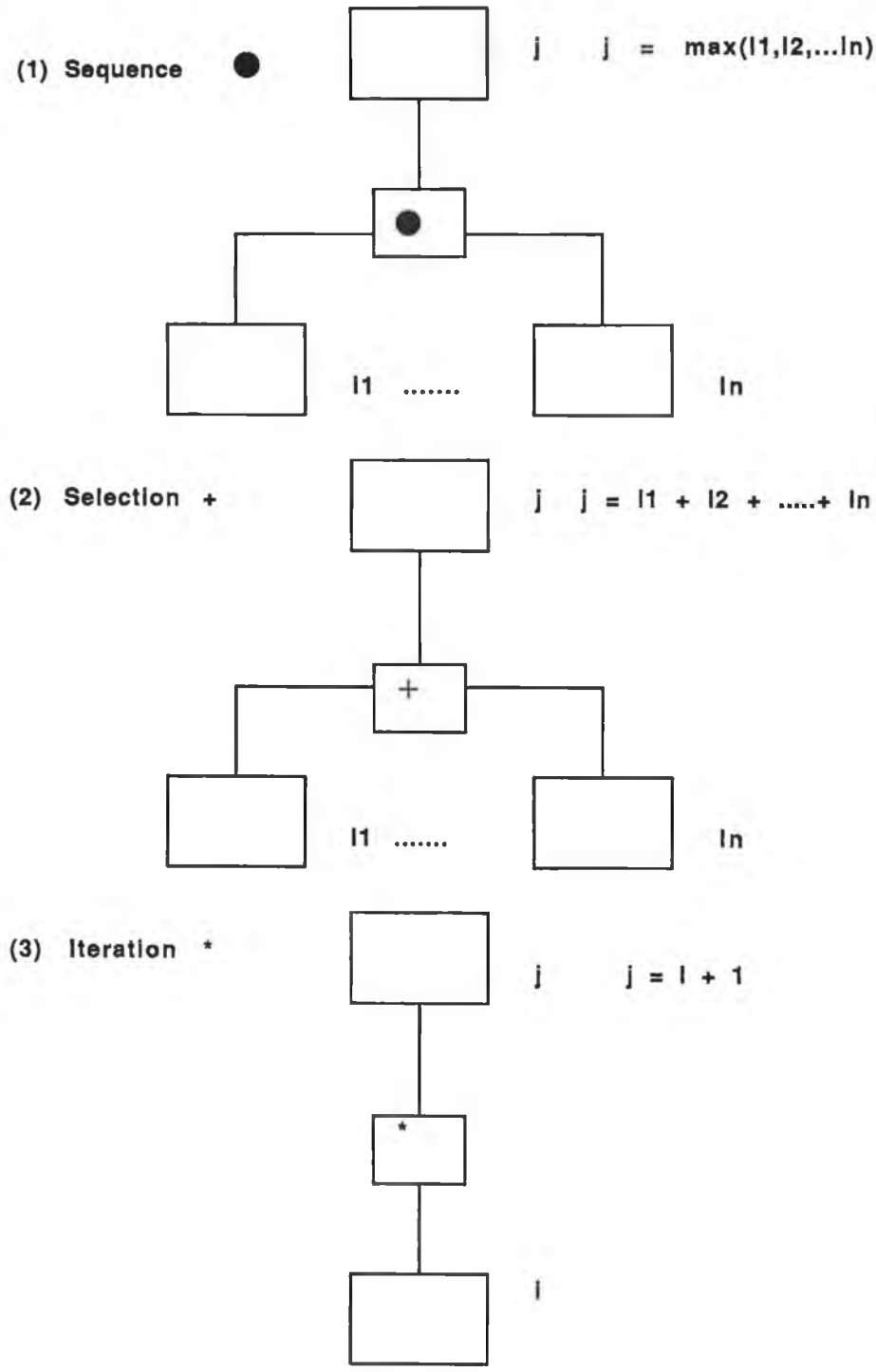

(1) Sequence
j
j = max(I1,I2,...In)
I1 .....
In
(2) Selection +
j
j = I1 + I2 + ...+ In
+
I1 .....
In
(3) Iteration *
j
j = I + 1
*
I

Figure 5.2

5.7.3 Rules for counting the number of test cases

- Sequence : $j = \max(i_1, i_2, \ldots . i_n)$
- Selection : $j = i_1 + i_2 + \ldots . + i_n$
- Iteration : $j = j + 1$

5.7.4 Deriving test cases

Each node of the CST is retrieved in post order, and the number of test cases in each node is calculated using the rules above. The test cases are then synthesized in numerical order from 1 to n where n is the number of test cases in the root node of the CST, while test cases are made up in each node from the children's test cases.

5.8 Estimating the number of Tango test cases

When using object-oriented unit testing, each object was treated individually. Now, testing is considered in terms of the model as a whole.

The total number of operations supported by the model is

$$\sum_{k=1}^{n} X_k$$

$$(X = 1 \ldots\ldots\ldots N)$$

N = number of objects

n = number of operations on each object

The total number of operation sequences produced by the model is of the order :

$$\prod_{1}^{N} \sum_{k=1}^{n} X_k \qquad (X = 1 \ldots\ldots\ldots N)$$

E.g. given a model with 4 objects each object having 3 operations would produce a total of 81 possible sequences of operations! The objective is to generate a minimum number of test cases that will fully exercise the system. In the following sections this issue will be discussed in more detail.

5.9 Minimising the number of test cases

Hetzel *[Het 88]* shows a way of examining the test cases derived logically from a program specification by dropping any that are redundant or dominated by any other cases. He says that there is no harm in having a few too many test cases. On the other hand, there is no point in using test cases that provide little or no additional information. He uses the idea of a functional test matrix as a way of selecting a minimal set of functional test cases that "covers" the program functions. The functional test matrix relates program functions to selected test cases or situations. Ostrand and Balcer in describing their strategy for test case generation *[Bal 88]* for a category-partition method minimise the number of test cases through

(1) Analysing the program specification

(2) Partitioning the different significant test cases that can occur within each category

(3) Determining the constraints among the choices of test cases within each category. The tester decides how different choices interact, how the occurrence of one choice can affect the existence of another, and what special restrictions might affect any choice.

5.10 Specifying test cases for Tango

I propose the following test specification for TANGO

(1) Take as input the validated rigorous model description consisting of objects, attributes on objects, operations on objects and pre and post conditions for each object operation.

(2) Produce test cases that will take the form of the valid sequences of operations along with a set of input and expected output data for each test case.

Using the test case form I presented in section 5.5 the Tango test cases appear as follows.

(1) **Test Case i.d.** : Unique test case number
(2) **Testing objectives :** To check that the sequence of operations perform correctly
(3) **Input :** Sequence of operations to be exercised along with pre-condition(s) and post-condition(s)
(4) **Expected Output :** Expected result from applying this operation sequence
(5) **Actual Output :** This is the actual result of applying the specified sequence of operations
(6) **Results :** The **Actual Output** is compared with the **Expected Output**. Comments are made on the adequacy of the test and the results are documented.

5.11 Minimising the number of Tango test cases

Each object operation has a set of private pre- and post- conditions. It is possible to make use of these conditions in that a post-condition of one operation may be linked to a finite number of pre-conditions of other operations. Therefore, the links between operations that depend on pre-and post-condition dependencies can be made explicit. (i.e. a post-condition $post_i$ of operation O_i ($i = 1.....n$) could be the pre-condition pre_{i+x} of operation O_{i+x} ($x > 0$). This is one possible way of constructing, as well as minimising a set of valid test cases.

5.12 Graph terminology

A directed graph or digraph $G = (V, E)$ consists of two sets: a finite set V of elements called vertices and a finite set E of elements called edges. Each edge is associated with an ordered pair of vertices. The symbols $v_1, v_2, ... v_n$ are used to represent vertices and the symbols $e_1, e_2, ...$ to represent the edges of a directed graph. A vertex is represented in the form of a circle and an edge is represented by a line segment connecting the circles that represent the end vertices of the edge. In addition each edge is assigned an orientation indicated by an arrow which is drawn from the initial to the terminal vertex. A walk W in G is a sequence of vertices $v_1 v_2 .... v_n$ such that $n > 0$. The length of a walk $W = v_1 v_2 .... v_n$, denoted $|W|$, is the number n of vertex occurrences in W. Note that a walk of length zero has no vertex occurrences; such a walk is called empty.

5.12.1 Applying Directed Graphs

Processing such a graph is akin to travelling around in a city with many one-way streets. Often the edge direction reflects some type of precedence relationship in the application being modeled. For example, a directed graph might be used to model a manufacturing line, with vertices corresponding to jobs to be done and with an edge from *x* to node *y* if the job corresponding to node *x* must be done before the job corresponding to node *y*.

5.13 Modelling Tango using graph notation

Figure 5.3 shows a new model representation for the validated Tango objects Patient, Patient Appointment, and Patient Test Results. For clarity purposes, the model only shows the sub-object types Patient Medical Report and Admissions List of the object Output. The object sub-types of Constants, which are Screening Doctor, Screening Nurse, Screening Type and Physical Examination Default Text operate as 'stand alone' objects so they are shown separately as in **Figure 5.4.** Objects that don't play a significant role in the general functionality of Tango are omitted.

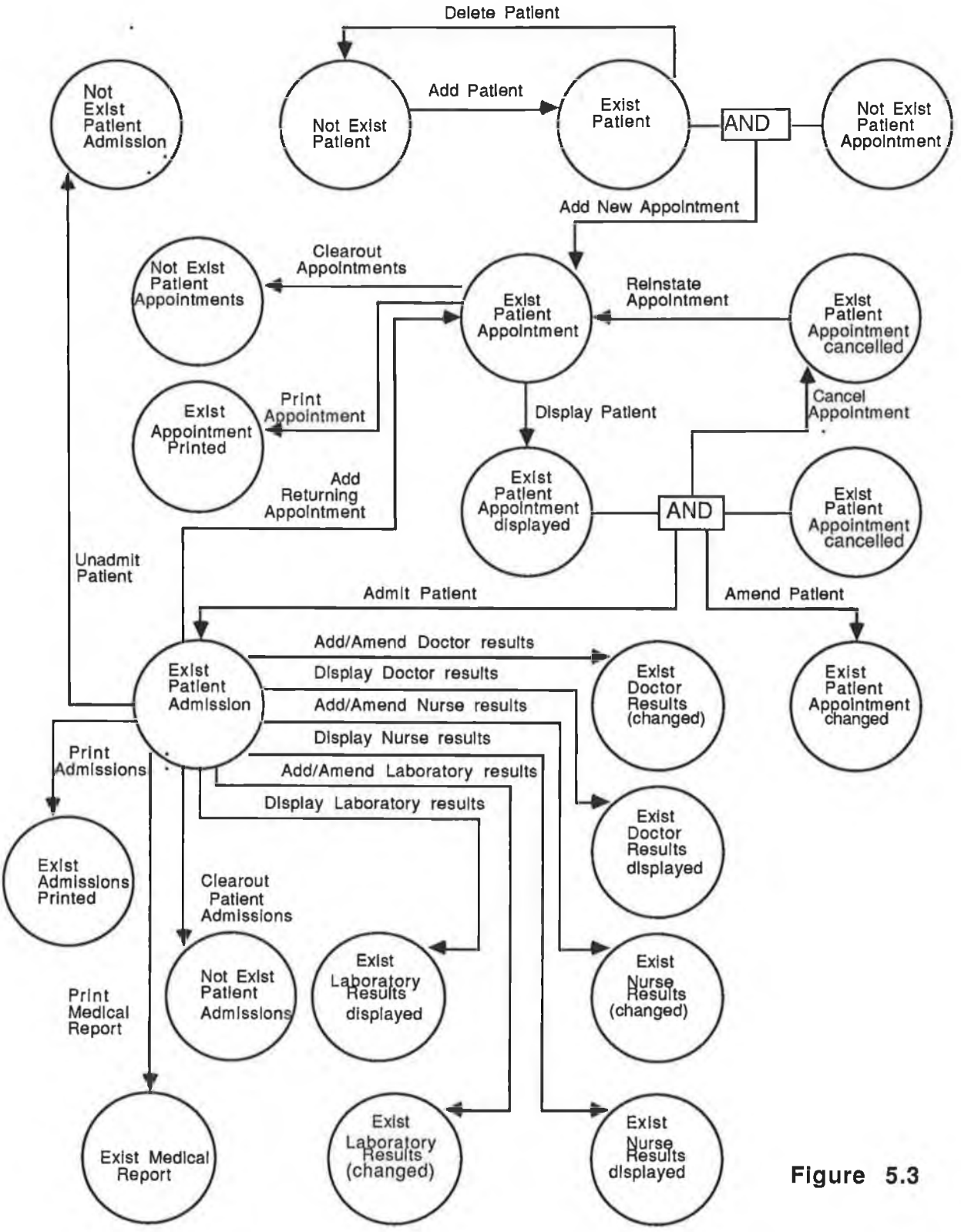
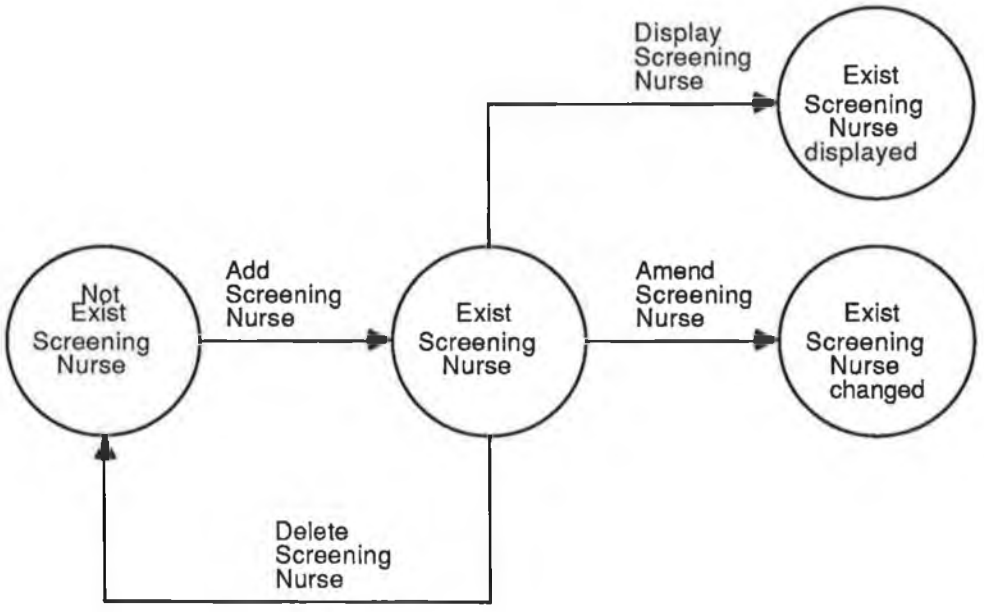

Delete Patient
Not Exist Patient Admission
Not Exist Patient
Add Patient
Exist Patient
AND
Not Exist Patient Appointment
Add New Appointment
Not Exist Patient Appointments
Clearout Appointments
Exist Patient Appointment
Reinstate Appointment
Exist Patient Appointment cancelled
Exist Appointment Printed
Print Appointment
Display Patient
Cancel Appointment
Add Returning Appointment
Exist Patient Appointment displayed
AND
Exist Patient Appointment cancelled
Unadmit Patient
Admit Patient
Amend Patient
Exist Patient Admission
Add/Amend Doctor results
Display Doctor results
Add/Amend Nurse results
Display Nurse results
Add/Amend Laboratory results
Display Laboratory results
Exist Doctor Results (changed)
Exist Patient Appointment changed
Print Admissions
Exist Doctor Results displayed
Exist Admissions Printed
Clearout Patient Admissions
Not Exist Patient Admissions
Exist Laboratory Results displayed
Exist Nurse Results (changed)
Print Medical Report
Exist Medical Report
Exist Laboratory Results (changed)
Exist Nurse Results displayed

Figure 5.3


Display
Screening
Nurse
Exist
Screening
Nurse
displayed
Not
Exist
Screening
Nurse
Add
Screening
Nurse
Exist
Screening
Nurse
Amend
Screening
Nurse
Exist
Screening
Nurse
changed
Delete
Screening
Nurse

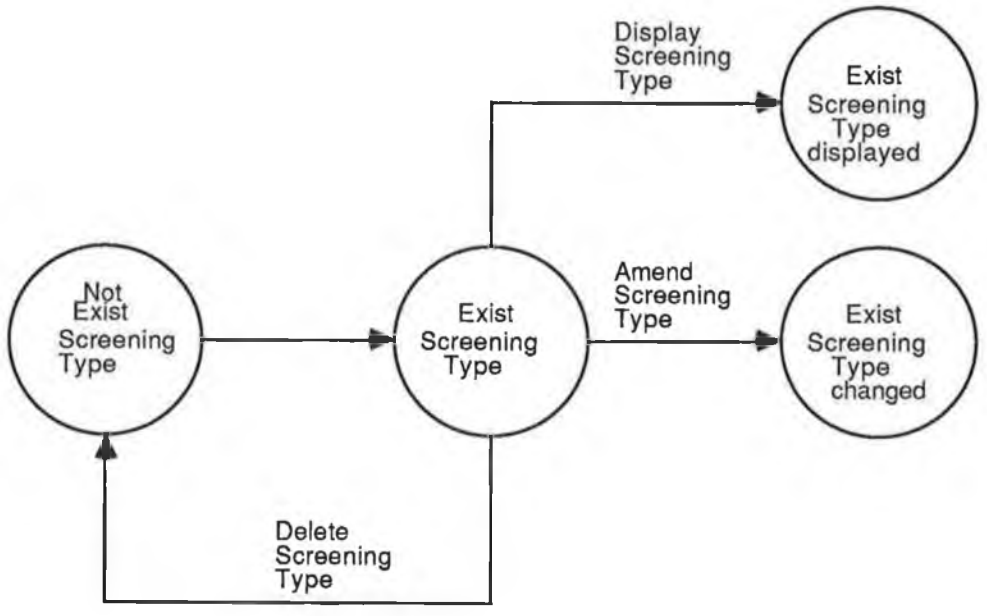
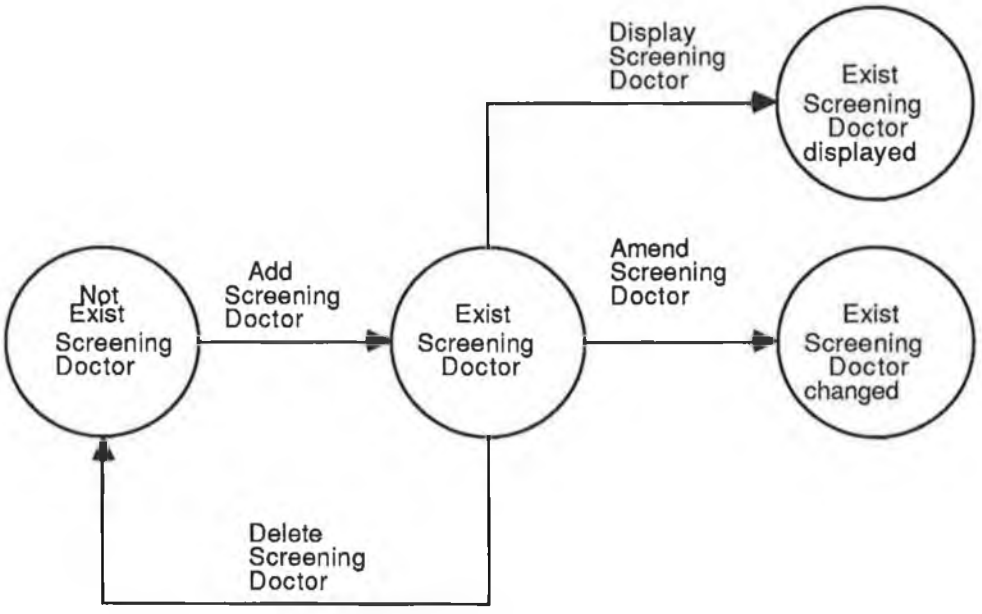

Display
Screening
Type
Exist
Screening
Type
displayed
Not
Exist
Screening
Type
Exist
Screening
Type
Amend
Screening
Type
Exist
Screening
Type
changed
Delete
Screening
Type

Figure 5.4



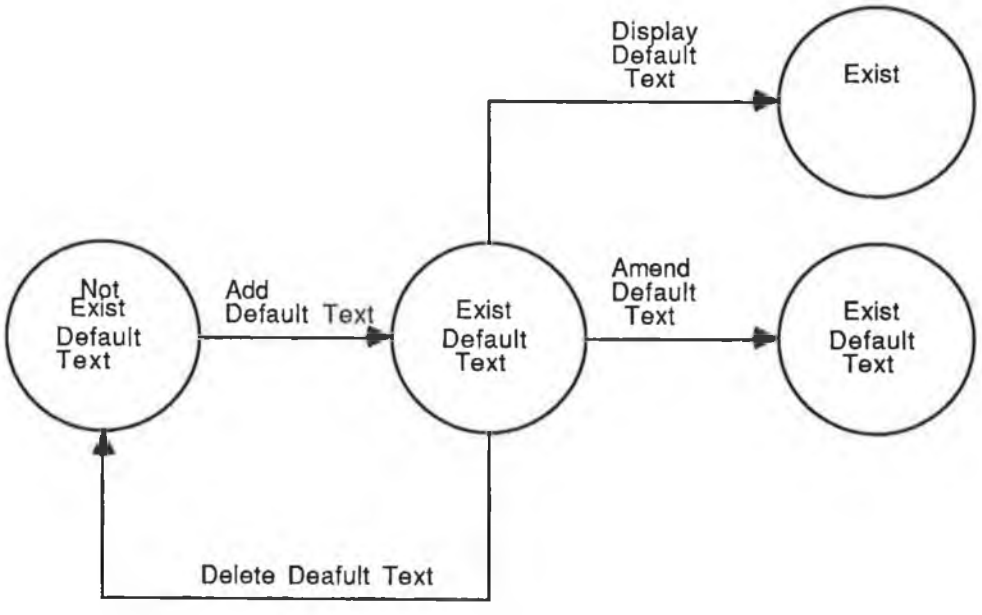


Figure 5.4 (continued)

The model in **Figure 5.3** was constructed by representing the pre-and post- conditions and operations of the various objects to form a directed graph. The conditions are used to act as vertices shown as circles in **Figure 5.3** while operations depending on these conditions are used to link these vertices and are represented as edges. An example drawn from **Figure 5.3** is where a pre-condition of the operation 'Add Returning Appointment' (represented by an edge) for a patient, is that the patient was previously admitted which is represented by the vertex labeled 'Exist Patient Admission'. Where two or more pre-conditions are required for an operation to be exercised they are shown as been ANDed together. It can be seen from the diagram which sequences of operations are legal and how test cases might be constructed to exercise the model. I will now examine an approach to this construction.

5.14 Using the graph model as a basis for testing

As mentioned previously, the model is defined in the form of a directed graph **G = (V, E)** where the elements of **V** are known as vertices or nodes and the elements of **E** are the edges. Yates and Malevris proposed a selection strategy aimed at reducing the number of infeasible paths generated during edge testing *[Mal 89]*. My objective is to achieve graph coverage through the following steps. Here is my proposal:

(1) Select a set ξ of paths through **G** which covers the edge set **E**; In section 5.8 I estimated the number of possible paths in the model to be of the order

$$\pi_{l}^{N} \sum_{k=1}^{n} X_k \qquad (X = 1.....N)$$

This estimation consists of many infeasible paths. e.g. one cannot have the operation sequence of 'Add Patient' followed by 'Display Laboratory Results'.

(2) Clearly the number of infeasible paths needs to be minimised. This can be achieved through deriving from ξ a corresponding set, ξ', of paths through **G** and constructing a set of test cases, $T(\xi')$ that will exercise **G** using each path of ξ' in turn; ξ' is derived through documenting the legal sequences of operations.

(3) Execute **G** with $T(\xi')$. (i.e. Applying a valid set of test cases that exercise **G**)

5.15 Adjacency Matrix theory

Let $G = (V, E)$ be a directed graph. Let $V = \{v_1, v_2, \ldots v_n\}$ and $E = (e_1, \ldots . e_k)$. The adjacency matrix $M = [m_{ij}]$ of G is an $n \times n$ matrix m_{ij} defined as follows :

$$m_{ij} = \begin{cases} 1, \text{ if } (v_i, v_j) \in E \\ 0, \text{ otherwise} \end{cases}$$

5.16 Developing an adjacency matrix for Tango

Figure 5.5 is identical to **Figure 5.3** except that a letter is now associated with each edge. Although the theory of an adjacency matrix states that (v_i, v_j) is set to 1 if there is an edge from vertex v_i to vertex v_j, I have altered the rule slightly such that (e_l, e_m) is set to 1 if there is a vertex from edge e_l to edge e_m.

The adjacency matrix in **Figure 5.6** is constructed directly from **Figure 5.5**. It shows a matrix entry of 1 if there is a pre or post condition(vertex) linking two operations(edges). The adjacency matrices for the graphs in **Figure 5.4** are self evident and are not discussed any further in this chapter.

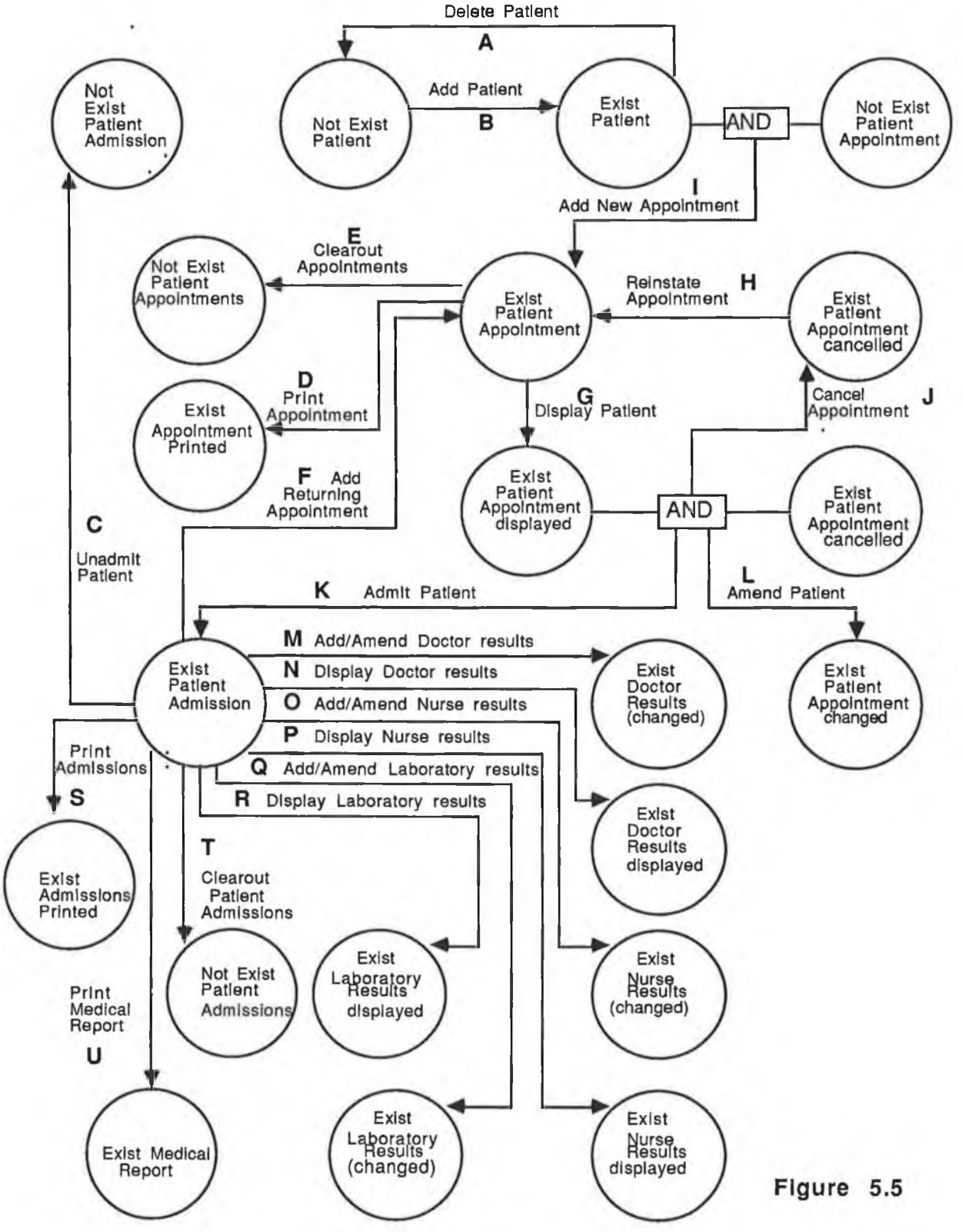

Delete Patient
A
Not Exist Patient Admission
Not Exist Patient
Add Patient
B
Exist Patient
AND
Not Exist Patient Appointment
I
Add New Appointment
E
Clearout Appointments
Not Exist Patient Appointments
Exist Patient Appointment
Reinstate Appointment
H
Exist Patient Appointment cancelled
D
Print Appointment
Exist Appointment Printed
G
Display Patient
Cancel Appointment
J
F Add Returning Appointment
Exist Patient Appointment displayed
AND
Exist Patient Appointment cancelled
C
Unadmit Patient
K Admit Patient
L
Amend Patient
M Add/Amend Doctor results
N Display Doctor results
O Add/Amend Nurse results
P Display Nurse results
Q Add/Amend Laboratory results
R Display Laboratory results
Exist Patient Admission
Exist Doctor Results (changed)
Exist Patient Appointment changed
Print Admissions
S
Exist Doctor Results displayed
T
Clearout Patient Admissions
Exist Admissions Printed
Not Exist Patient Admissions
Exist Laboratory Results displayed
Exist Nurse Results (changed)
Print Medical Report
U
Exist Medical Report
Exist Laboratory Results (changed)
Exist Nurse Results displayed

Figure 5.5

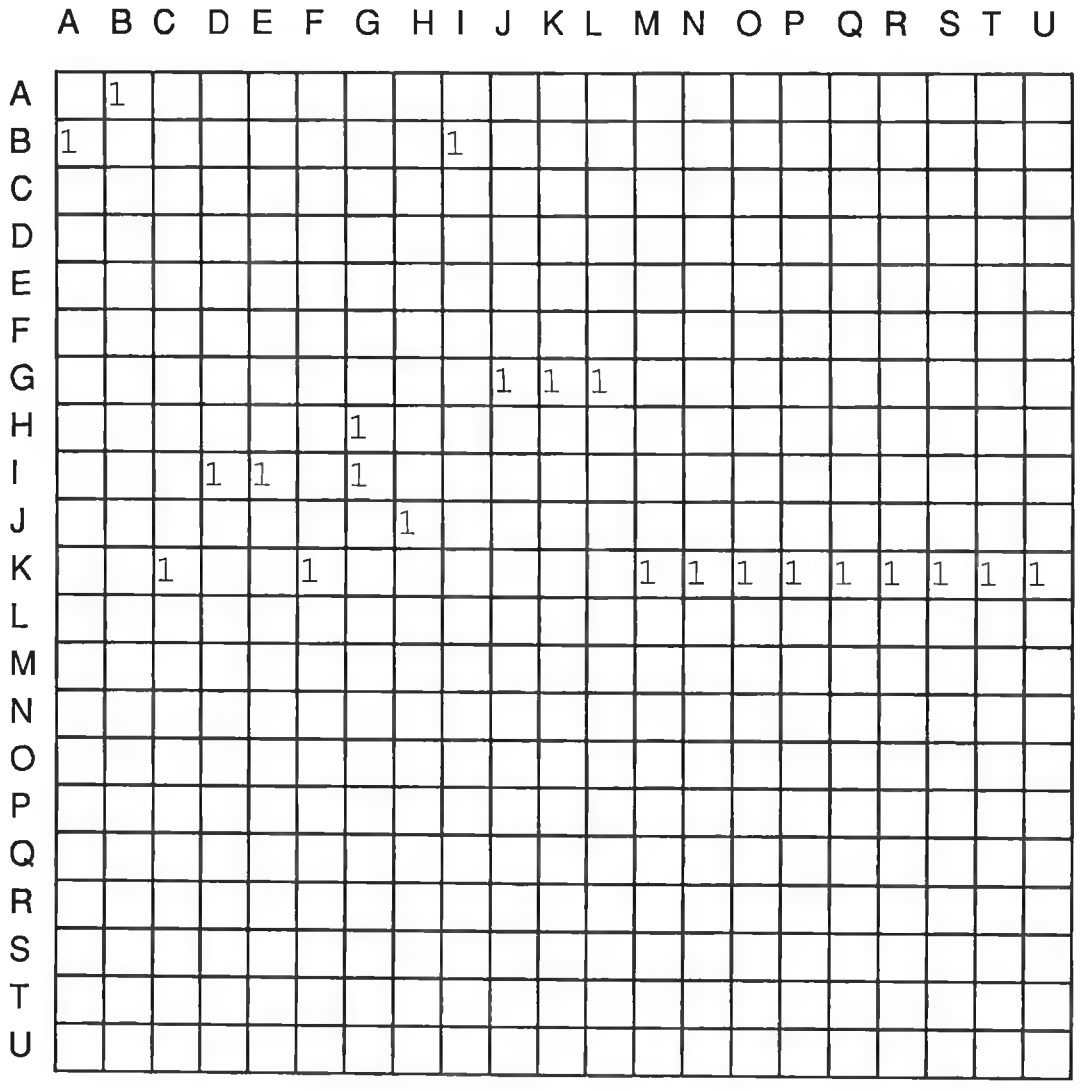


Figure 5.6

5.17 Interpreting the adjacency matrix

As stated, an adjacency matrix entry equal to 1 in **Figure 5.3** signifies a link between two operations. An adjacency list can be easily derived from an adjacency matrix. A link from x to y is represented by the appearance of a y on x's adjacency list. The adjacency list for **Figure 5.6** is shown below in **Figure 5.7**.

A : B
B : A I
G : J K L
H : G
I : D E G
J : H
K : C F M N O P Q R S T U

Figure 5.7

Reading the adjacency list literally signifies that **B** is followed by **A. A** and **I** are followed by **B** etc.,. From this the first set of test cases can be derived. An example of a test case derived from **Figure 5.7** is:

(1) **Test Case i.d.** : 2 B A (i.e. Test case number 2, exercising edges B followed by A)
(2) **Testing objectives :** to check that the operation 'Add patient record' followed by the operation 'Delete patient record' perform correctly.
(3) **Input :** Add patient record/Delete patient record
(4) **Expected Output :** The patient record doesn't exist
(5) **Actual Output :** (The result of applying the above **Input**)
(6) **Results :** The **Actual Output** is compared with the **Expected Output**. Comments are made on the adequacy of the test and the results are documented.

5.18 Manipulating the adjacency matrix

The (i,j) entry m_{ij}^r of M^r is equal to the number of directed walks of length r from v_i to v_j. **Figure 5.8** shows the result of squaring the adjacency matrix M of **Figure 5.6** (i.e. M^2). This matrix shows the number of directed walks of length two from operation i to operation j.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
A	1																				
B		1		1	1		1														
C																					
D																					
E																					
F																					
G								1													
H										1	1	1									
I										1	1	1									
J							1														
K																					
L																					
M																					
N																					
O																					
P																					
Q																					
R																					
S																					
T																					
U																					

Figure 5.8

The adjacency list derived from the adjacency matrix in **Figure 5.8** is shown in **Figure 5.9**. This can be interpreted as previously. Test cases are derived which gradually cover the model. The next step in the testing process would involve cubing the matrix M (i.e. M^3), deriving the adjacency list and exercising the test cases. These test cases would list the operations of length three from operation k to operation l.

A : A
B : B D E G
G : H
H : J K L
I : J K L
J : G

Figure 5.9

5.19 General conclusions

In section **5.14** I explained that my objective was to derive a set of test cases that would exercise the model through identifying the set $T(\xi')$. I used adjacency matrix theory to represent the directed graph in **Figure 5.3**. The test cases that were derived from the adjacency matrices and lists, can be viewed as being members of the valid set of test cases. These test cases will eventually fully exercise the model. They can be represented in the form $M^r \varepsilon T(\xi')$. In Chapter six I make some overall conclusions and recommendations on my testing approach.

Chapter 6

Conclusions And Future Directions

6.1 Justification of object-oriented testing method

How can informally developed software with no functional specification, an informal development trail, no design documents be tested ?
The answer is it cannot be tested, unless you try to reconcile what documents you have with the actual functionality.
This is the core of my method as discussed earlier. Developing the rigorous model in Chapter three was the first attempt at this reconciliation of claimed functionality versus actual functionality. Although building the model figured more as a documentation metric than a testing metric it proved to be very useful for object-oriented unit testing. This helped to validate the model through exercising each of the object classes against the actual functionality of the product. The results of this testing uncovered many problems with the documentation.
A new testing model was constructed using the validated model of objects, attributes and pre- and post- conditions. The basis of this model was that operations were linked via these conditions so different operations were sometimes dependent on the same pre-condition(s). I used the established methods of graph theory and adjacency matrices to represent my testing model. As you can see the model is rather easy to read and it describes the system in a very realistic way. i.e. If I want to add a patient appointment for a patient, then a patient record for that patient must already exist, and I must also display that patient record. So it reads quite literally!
One of my initial objectives was to devise a method that would help to generate test cases and to exercise adequate functional coverage of the system. The testing model helped me to achieve my objective. Through representing the model in the form of an adjacency matrix and subsequently as an adjacency list, test cases were systematically generated.

6.2 Applying other testing methods

In classical functional testing, the observable functions implemented by a system or program are tested over typical input-output scenarios. Functional testing is often augmented with special case and boundary tests. Software testing still is the principal way of locating errors in programs and it is the bench-mark by which the pre-delivery level of confidence in a program's veracity is established. Even though there are many testing techniques available (see Chapter four), none of them can actually guarantee to isolate all sources of program error. Howden *[How 78]*, recommends that several techniques should be applied in order to arrive at the desired confidence level. (i.e. we have tested this product as much as we possibly can).

In order to strengthen the proposed object-oriented testing methodology I have described in Chapter five,some established testing methods can be used to enhance the method making it perhaps more trustworthy. Black-box testing methods such as Category-Partitioning and Boundary Value Analysis are just two examples which could be used.

6.2.1 Applying the Boundary Value Analysis method

The following test case is derived from the testing model.

(1) **Test Case i.d.** : 7 B
(2) **Testing objectives :** to check that the operation 'Add patient record' performs correctly.
(3) **Input :** Add patient record
(4) **Expected Output :** The patient record exists
(5) **Actual Output :** (The result of applying the above **Input**)
(6) **Results :** Document the comparison of **Actual Output** with **Expected Output**

Through using the Boundary-Value Analysis method the **Input** section of this test case could be augmented e.g. If I know that the system is to support patient record numbers from 1 to 1000, then when I'm adding a patient record, I will allocate patient numbers at the boundaries (i.e. -1,0 and 999,1000,1001).

6.2.2 Applying the Category-Partition Method method

This method uses the notion of a test frame which consists of a test case, the commands necessary to set up the test and instructions for checking the test. Using the same test case example the test frame produced might look like:

Test Frame:

Test Case i.d. : 7 B
Input : Add patient record
Constraints : *(1)* Not Exist Patient
(2) Exist Patient

Commands to set up the test:
Using constraint *(1)* Add patient record
Actual Output : (The result of applying the above **Input**)
Expected Output : The patient record exists.

Using constraint *(2)* Add patient record
Actual Output : (The result of applying the above **Input**)
Expected Output : Patient record already exists.

Instructions for checking the test:
Document the comparison of **Actual Output** with **Expected Output** in each of the two cases above.

6.3 Formal Methods

Gerhart *[Ger 89]* discusses Formal Methods in relation to developing trustworthy computer systems. What transpired is that Formal Methods are as important to software engineering as are Applied Mathematics to Aeronautical engineering. Here are some of the relevant points she makes :

- safety critical systems are compelled to work the first time, whereas other systems are phased in gradually
- software engineers are not subject to the professional certification criteria or indeed the codes of practice of other engineering fields.
- no set of standard engineering techniques for safety critical systems has been agreed upon.

This introduces the notion of formalising the testing method I developed.

6.3.1 The model and states

What I have effectively achieved in terms of the testing model, is to define a system in terms of its operations along with the pre- and post- conditions on these operations. Thus having pre condition **pre i** and applying operation **ope i** then the post condition **post i** will be true. So the state of the system changes from **pre i** to **post i.** Through documenting a system for every pre condition, post condition and operation, a system can be completely defined. This is akin to the notion of an FSM (Finite State Machine). An FSM is often drawn as a state transition diagram showing how the system moves from one state to another, or as a matrix in which the dimensions are state and input, and the matrix cells contain the action and new state resulting from receipt of the input in the given state.

The value of FSMs are such that the system can be modelled as a black box taking as input a sequence of operations. The advantages of such a representation are that certain properties can be checked both mechanically and reliably. It is also quite easy to work with such a representation.

6.3.2 Formalising the testing method

In attempting to formalise my testing method some of the ideas of VDM *[Jon 90]* can be applied. VDM (Vienna Development Method) is a method which enables the specification of sequential systems in terms of pre- and post- conditions of states. System data types consist of state specification (satisfying an explicitly given invariant) together with a collection of operations.
Referring to the rigorous model of Chapter three the operation
Add New Appointment on object **Patient Appointment** is represented as:

Add New Appointment
ext wr Appointment Number/Date of Visit/Time of Visit/etc.,
rd Patient Number/Patient Surname/Patient Address
/Date of birth/phone number/etc..
pre condition Exist Patient **OR** Not Exist Patient Appointment
post condition Exist New Appointment

It can be seen that each operation is classed as having a set of private variables which may be either read(**rd**) or write(**wr**) access. Also the pre- and post- conditions relevant to that operation are documented. The notion of a pre-condition can be linked to the set of private variables attributable to that operation. In the example above a pre-condition of the **Add New Appointment** operation is stated as **Exist Patient OR Not Exist Patient Appointment.** The pre-condition **Exist Patient** can be expanded to the data represented by this statement, (i.e. patient i.d, patient address, age, marital status...). All of this data must be valid for the pre-condition to hold.

6.4 Developing the rigorous model - an alternative approach

In Chapter three I developed a rigorous model from a user oriented functional specification. The model served many functions notably highlighting problems with the documentation. The final model also formed the basis for object-oriented unit testing. When this testing was completed, the model was used towards a way of deriving test cases. Constructing the model should not be thought of in a post-development sense (i.e. wait until product is developed with it's user documentation and then reverse engineer a rigorous description). The model could be built during the design stage of the product by the developers themselves or even the users.

6.5 Manipulating the model

I have represented the model as an abstraction of the rigorous model constructed in Chapter three. This representation proved useful for applying adjacency matrix theory to derive test cases. The model can be viewed in some other ways as a further means to support the testing process.

6.5.1 A White box view

The model can be viewed as a white box structure consisting of If/Then/Else/Or possibilities. From the model or graph it can be seen how the model is driven using operations. A typical white box structure derived from the model might be represented in the following way by taking some of the operations and conditions.
E.g.
If pre-con = Exist Patient **And** Not Exist **And**
If operation = Add New Appointment
Then post-con = Exist Patient Appointment;

If pre-con = Exist Patient Appointment **And**
If operation = Display Patient
Then post-con = Exist Patient Appointment displayed;

If pre-con = Exist Patient Appointment displayed **And**
Not Exist Patient Appointment cancelled **And**
If operation = Admit Patient
Then post-con = Exist Patient Admission **Else**
If operation = Amend Patient
Then post-con = Exist Patient Appointment changed **Else**
If operation = Cancel Patient Appointment
Then post-con = Exist Patient Appointment cancelled;

6.5.2 The Cyclomatic Complexity of the Model

The cyclomatic number *[War 89]* and the accompanying program control flow graph can be used to identify test cases. The cyclomatic number corresponds to the number of test paths which correspond to the basic paths derived from the control flow graph. With this information, test cases can be generated for a program. Therefore, the cyclomatic number could be derived from a white box structure as shown in **6.5.1** or even directly from the adjacency matrix representation of the model. This could be achieved in the following way.

Figure 6.1 is identical to **Figure 5.3** except each vertex is now labelled with a number. **Figure 6.2** is an adjacency matrix representation of the model shown in **Figure 6.1**. This time the adjacency matrix is developed with the idea that (v_i,v_j) is set to 1 if there is an edge from vertex v_i to vertex v_j. Referring to **Figure 6.2**, each row with two or more entries represents a decision vertex. Therefore, by totalling the number of 1s in rows 2, 5, 7 and 9 minus the number of relevant rows(i.e 4) the cyclomatic number for the model can be obtained. In this case the cyclomatic number is 12. Obviously, when the matrix is further manipulated to show the operations that can be reached over a distance of two and three etc., different cyclomatic numbers will be produced.

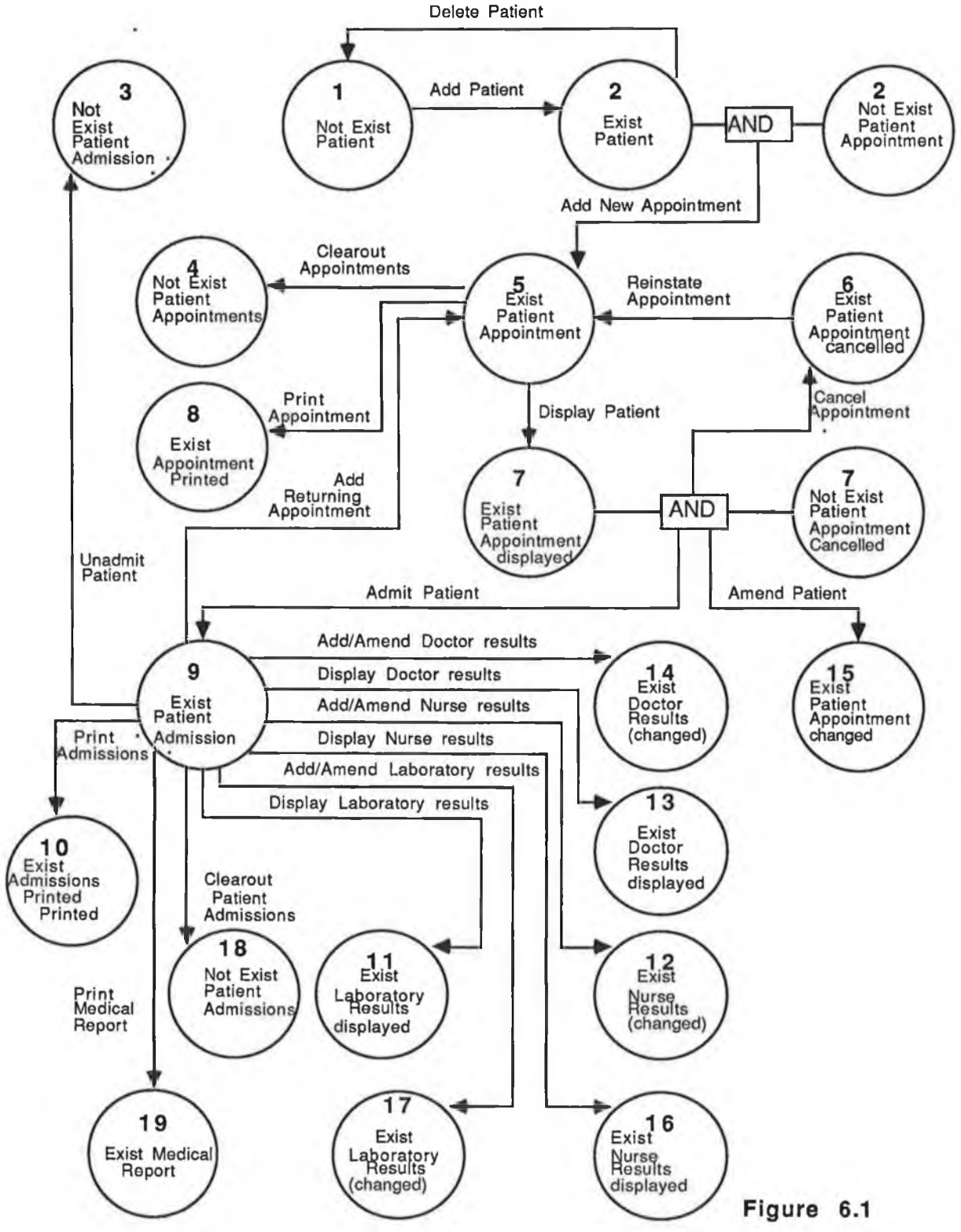

Delete Patient
3
Not Exist Patient Admission
1
Not Exist Patient
Add Patient
2
Exist Patient
AND
2
Not Exist Patient Appointment
Add New Appointment
4
Not Exist Patient Appointments
Clearout Appointments
5
Exist Patient Appointment
Reinstate Appointment
6
Exist Patient Appointment cancelled
8
Exist Appointment Printed
Print Appointment
Display Patient
Cancel Appointment
Add Returning Appointment
7
Exist Patient Appointment displayed
AND
7
Not Exist Patient Appointment Cancelled
Unadmit Patient
Admit Patient
Amend Patient
9
Exist Patient Admission
Add/Amend Doctor results
Display Doctor results
Add/Amend Nurse results
Display Nurse results
Add/Amend Laboratory results
Display Laboratory results
Print Admissions
14
Exist Doctor Results (changed)
15
Exist Patient Appointment changed
13
Exist Doctor Results displayed
10
Exist Admissions Printed Printed
Clearout Patient Admissions
18
Not Exist Patient Admissions
11
Exist Laboratory Results displayed
12
Exist Nurse Results (changed)
Print Medical Report
19
Exist Medical Report
17
Exist Laboratory Results (changed)
16
Exist Nurse Results displayed

Figure 6.1

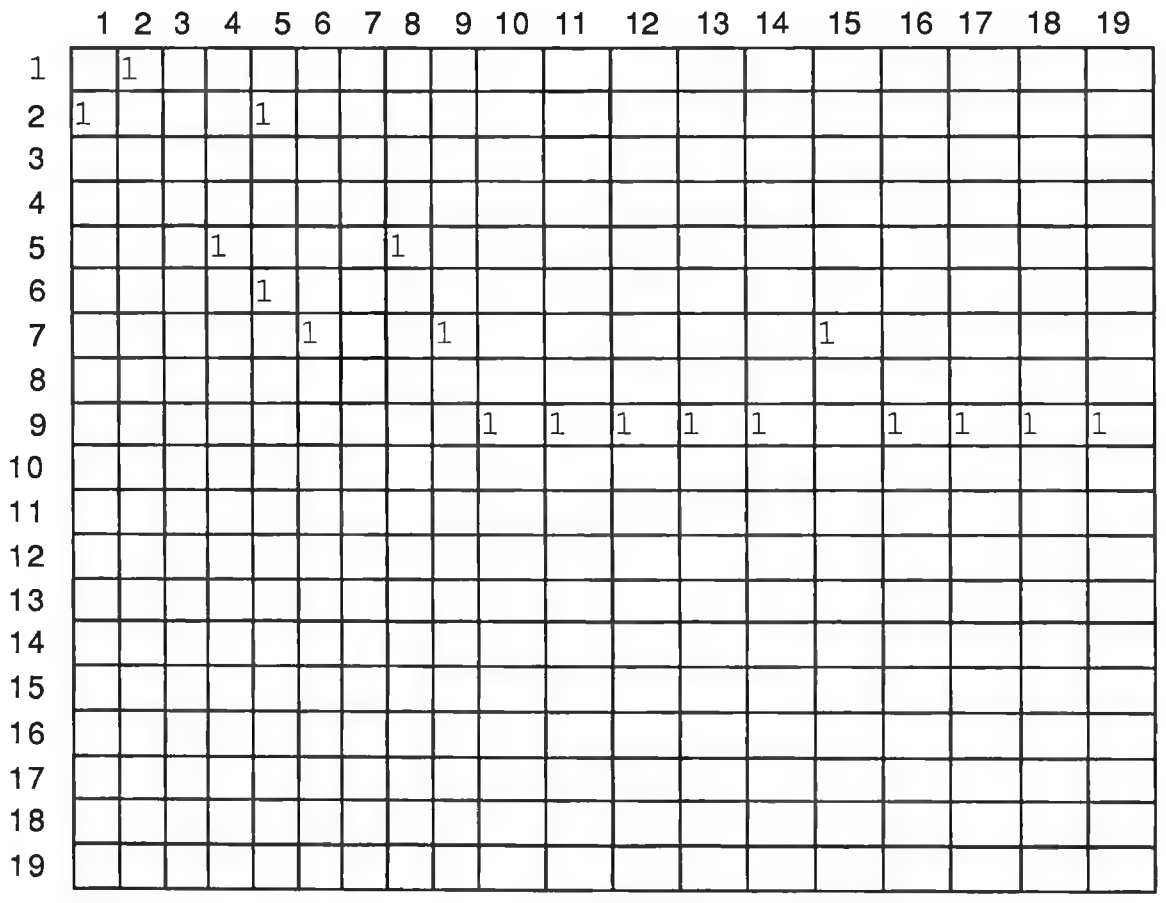

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

Figure 6.2

6.6 Future Directions

The obvious next stage would be tool support for my testing method. Initially the rigorous model would have to be developed manually but an object-oriented tool could be used to support the validated testing model, in terms of conditions and operations. From there, one could see a tool that would automatically generate test cases using my object-oriented testing approach.

6.7 General Conclusions

The method I developed applies itself well to a large number of existing software products with no functional specification. I started out with a user-oriented manual/specification and I ended up with a valid set of test cases that couldn't have possibly been constructed from simply reading the documentation. The model shows an accurate concise view of the system. I could see how the model might become part of the documentation as it interprets the functionality of the system very clearly.
I showed how test cases can be derived from the model and how it accommodates other testing methods. The fact the model could be developed during the design stages of the product adds credibility to the method. In this way the functional testing problems that arise could be overcome, as the tester would have knowledge of the functions within an operation. The system could then support a white box testing approach. At the moment the model is really used to implement a black box object-oriented testing method. Incorporating a white box method along with the existing black box technique would contribute to a very comprehensive testing methodology, one that would ensure even better functional coverage.

APPENDIX A

CHECKLIST FOR ASSESSING THE QUALITY OF A HUMAN-COMPUTER INTERFACE

Introduction

The use of checklists to assess the quality of a user-interface is very much a 'second-best ' approach. Empirical research has shown that conclusions reached by certifiers and system designers through the use of 'checklists' are generally less valid and reliable than assessments based on actual user experience of the interface. Despite this caveat, experience has shown that inspection guided by a checklist can reveal any serious deficiencies in a user-interface.

The checklist questions shown overleaf are cross-referenced to the components of quality in figure 1 to which they refer *[Joh 89], [Shn 86], [Whi 88]*.

In making an overall judgement on the adequacy of the product's interface,reference should be made to the agreed Certification Specification for the product.

	YES	UNCERTAIN	NO
The whole 'style' and 'feel' of the interface is appropriate for the application. (1.1,1.7)			
Menus,functions,commands,keywords etc. have well-chosen names and are easy to remember. (1.1,1.2,1.3)			
Icons are well designed and have an obvious meaning. (1.1,1.2)			
There is consistency in design etc., of: (1.3)			
data entry panels and forms			
data display panels			
fields			
menus			
command lines,prompts etc.			
error messages			
'help' information			

There are consistent,easy-to-remember
conventions for : (1.3)

using selection mechanisms
(e.g. mouse,cursor control)

use/meaning of function keys

use/meaning of 'buttons',scroll-bars etc.,

The user is unlikely to get 'lost' in the software
(i.e. navigation is not too mentally demanding).
(1.5)

If the user DOES get lost,he/she can easily escape to
the 'top' or other 'home' position without doing damage.
(1.5)

The design of menus,icons,commands,selection mechanisms etc.
lets the user move quickly to where he/she wants to go.
(1.5)

Input data validity checking is adequate.
(1.6)

Error messages are sufficient and helpful.
(1.6)

Users can't 'crash' the system by mistake.
(1.6)

User mistakes are signalled positively an appropriate mode (e.g.
'bleep',flashing panel)
(1.6)

Internal system problems and exceptional conditions are
signalled positively and in an appropriate mode (e.g. 'bleep' etc.)
(1.6)

Adequate user-help is provided (on-line or hard-copy).
(1.2,1.6)

The user is never left wondering if the system has 'died'
(i.e. there is some immediate response to all user-actions,the
system reports progress of lengthy internal operations).
(1.4)

Advanced functions are 'hidden' or otherwise packaged so that
beginners are not overwhelmed.
(1.2)

Inexperienced users are offered suitable default values
and/or assistance in setting parameter values.
(1.2)

Adequate 'fast-track' is available to experienced users
(e.g. command language,user-definable macros).
(1.2,1.5)

Display screens and panels are well laid-out and 'uncluttered'.
(1.8)

Character-sets,icons,graphics etc. have good legibility.
(1.8)

Window manipulation functions are adequate.
(1.8)

'WYSIWYG' is used where needed.
(1.4)

Response-times and screen display-rates are adequate.
(1.4,1.8)

OVERALL JUDGEMENT ON THE QUALITY OF THE INTERFACE :

ACCEPTABLE

UNDECIDED

UNACCEPTABLE

COMMENT :

APPENDIX B

LOGICAL SECURITY CHECKLIST FOR SINGLE USER MICROCOMPUTER-BASED SYSTEMS HOLDING SENSITIVE OR CRITICAL DATA

In making a judgement on acceptability, reference should be made to the agreed Certification specification.

E=usually considered essential
D=usually considered desirable

1. LOGICAL ACCESS CONTROLS

1.1 Identification of user :

(E) Must the user provide a user-id which is validated by software before access is allowed?

(E) Is the user-id at least four characters long?

1.2 Authentication of user :

(E) Must the user provide a password which is verified against the user-id?

(E) Is the password at least six characters long?

(E) Is display of the password suppressed?

(E) Is the user allowed only a pre-set number of attempts to enter a password (e.g. three)?

(D) Is the table of user-id/passwords encrypted?

(E) Is a user able to change his/her password?

(D) Does the system force a user to periodically change his/her password?

1.3 Authorisation of user :

1.3.1 Access to data :

(E) Does the system allow restriction of individual user's access rights to read,copy,update and delete specified data on the basis of password?

(D) Does the system keep a log of all accesses to specified data showing user-id,date,time and operations invoked?

(D) Does the system automatically sign-off a user after a period of inactivity?

1.3.2 Access to functions :

(E) Does the system allow restriction of individual user's access to specified functions/procedures on the basis of password?

(E) Does the system keep a log of all accesses to specified functions showing user-id,date and time?

(D) Does the list of functions displayed to a user list only those functions the user is authorised to perform?

(E) Are users prevented from reading or amending the software?

2. APPLICATION CONTROLS

(E) Can the system maintain an audit-trail of specified transactions stamped with user-id,date and time?

(E) Does the system support any needed reconciliation controls on inputs,processing and outputs?

(E) Is input data adequately validated for format,range and sequence?

(E) Is there adequate functionality for backing-up data and software?

(D) Does the system force the user to take back-ups of data and software at appropriate intervals?

(E) Is there adequate functionality to allow the user to restore the system after a crash?

(D) Is there a facility for automatic recovery after a crash (through,for example,the use of parallel transaction files)?

3. OPERATING SYSTEM SECURITY CONCERNS

(E) Is the system secure against users attempts to use operating system services rightly reserved for 'super-user'?

OVERALL JUDGEMENT ON LEVEL OF LOGICAL SECURITY

ACCEPTABLE

UNACCEPTABLE

COMMENT :

APPENDIX C

TANGO - FAILURE REPORT

(1) Time of start of session (HH:MM) :

(2) Date of occurrence (DD/YY/MM) :

(3) Time of occurrence (HH:MM) :

(4) Failure type :
(Please tick whichever is appropriate)

(i) Adding/Amending a patient record
(ii) Entering test results
(iii) Doing a 'backup'
(iv) Doing a report
(v) Using a system utility

(5) System identifier :

(i) Hardware :
(ii) Operating System :

(6) Severity:
(Please tick whichever is appropriate)

(i) Complete system 'crash'
(ii) Hardware failure
(iii) TANGO system failure (e.g. accessing a file)
(iv) Needed to 'rebuild' system fully
(v) Needed to 'rebuild' some of the system

(7) Time of start of session (HH:MM) :

(8) Part of system which failure occured:
(Please tick whichever is appropriate)

(i) Patient Appointment/Admission
(ii) Nurses Functions
(iii) Doctors Functions
(iv) Laboratory Results
(v) Reports and labels
(vi) System Utilities

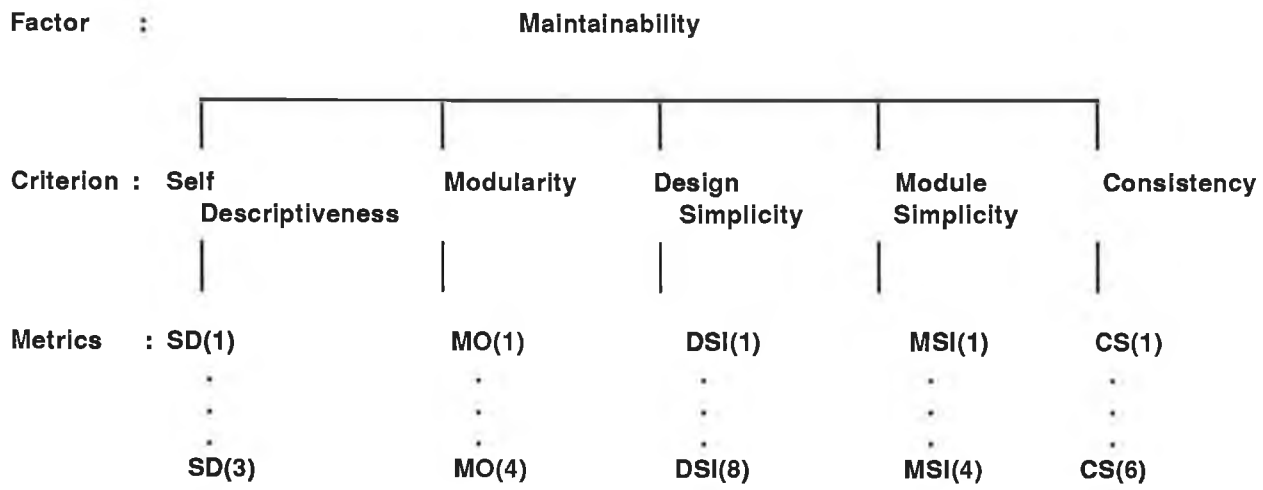
APPENDIX D

Maintenance and Maintainability

Maintenance is modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.

Maintainability is the ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements *[IEE 89]*.

The quality specification of maintainability defined as in [UKM 89] is shown below.



Referring to Li and Cheung *[Che 87]* and Ward in *[War 89]* the following checklists relating to each criterion are established.

Self-descriptiveness :

SD(1) Is there a standard such that each module contains the following :

(i) a description of the module's function

(ii) the author's name

(iii) the version number

(iv) the version date

(v) a documented list of the inputs accepted by that module

(vi) a documented list of the outputs produced by that module

(vii) a description of the processing to be done in that module

(viii) the pre-requirements that must be satisfied before that module can be executed

SD(2) Does the standard used support adequate in-line comments ?
(i.e. a consistent frequency of good quality comments in each module)

SD(3) Does the standard used support a strict format for the structure of each module ?
(i.e. Does each module have declarative / input / processing / output sections respectively)

Modularity :

MO(1) Are all software modules and functions developed according to a structured technique ?
(e.g. through the use of DFDs or structure diagrams)

MO(2) What type of coupling exists ?
Note : (Coupling is the relationship that exists between two or more modules)

Type of coupling

(1) Content coupling : One module references the contents of another module

(2) Common coupling : Modules reference a shared global data variable

(3) External coupling : Modules reference the same externally declared variable

(4) Control coupling : One module passes control to another module

(5) Stamp coupling : not as in **(1)..(5)**

MO(3) Is output data always passed back to the calling module ?
(An example might be through parameter calls using global variables)

MO(4) Is control always returned to the calling module when execution is complete ?
(i.e. the code doesn't form a "daisy chain" of statements.

Design simplicity :

DSI(1) Is the system and program documentation of good standing quality ?
(i.e. flow charts, hierarchical diagrams...)

DSI(2) Is every module written identified in documentation ?

DSI(3) Is there a description of every module in the documentation ?

DSI(4) Is there a description of "why" every module exists ?

DSI(5) Is there a statement of "what" the module does ?

DSI(6) Are all sub-ordinate modules identified in sequence ?

DSI(7) Is there a description of the "relationship" of each module with another module or modules ?

DSI(8) Is there a description of how other modules "interact" with a particular module ?

Module simplicity :

MSI(1) Is each module well structured ?
(i.e. "non-spaghetti" structure)

MSI(2) Is the flow of control within a module from top to bottom ?
(i.e. typically non-looping within a module.

MSI(3) Does each data variable have a single use ?

MSI(4) Does each module perform a single, explicit function ?
(e.g. we might have a 'control' module solely concerned with calling other modules in some sequence).

Consistency :

CS(1) Do the input procedures and input formats for each module have a set standard ?
(Do they take a similar structure in each module ?)

CS(2) Do the output procedures and output formats for each module have a set standard ?
(Do they take a similar structure in each module ?)

CS(3) Is error handling dealt with in a consistent manner in each module ?
(Are inputs validated before accepted ?)

CS(4) Is the amount of error handling adequate for each module ?
(i.e. Are all possibilities covered ?)

CS(5) Is the naming of all data variables standardised ?
(Are meaningful variable names used?)

CS(6) Is there a section in each module which deals explicitly with the declaration and definition of all global variables in that module ?

Applying McCabes complexity metric to Tango

The McCabe metric is a number that represents the complexity of a module. It is based on the number of decision statements in the module. It has been found that if the complexity measure of a module exceeds 10, the expectation of that module being error-prone also increases and hence the more difficult to maintain.

The cyclomatic complexity of a module can be represented numerically as in *[War 89]*. The general form of the complexity metric is :

V(G) = No. of edges - No. of nodes + 1

I applied McCabe's metric to some INFORMIX 4GL(Tango) source code modules given that the language emphasises some procedural constructs. I took a random selection of 40 modules and we established the cyclomatic complexity for each. The total complexity was 200, giving an average module complexity of 5. On an average basis this number is well within the acceptable range. Only two of the modules had complexity values greater than 10 (i.e. 17 and 21).

Therefore, on a 'complexity' level it would suggest that Tango software is easily maintainable given its low module complexity numbers and an average module complexity of 5.

APPENDIX E

ASSESSMENT OF 'EASE OF HOUSEKEEPING' REQUIRED BY THE PRODUCT

By 'housekeeping' is meant operations such as :

Product Installation

Customisation

Installation of enhancements/new releases

Taking security backups

Restoring files/databases from backups
(where these are the responsibility of the user)

The assessment should include both the operation and the adequacy of any supporting documentation or on-line help. The criteria used when assessing an operation should be those defined in the agreed certification specification.

OPERATIONS EXAMINED	ACCEPTABLE	NOT ACCEPTABLE
(1)		
(2)		
(3)		
(4)		
.		
etc.,		

COMMENTS :

APPENDIX F

ASSESSMENT OF 'PERFORMANCE' OF THE PRODUCT (excluding 'reliability')

'Performance' attributes fall into at least three categories :

Processing times (e.g. response times,throughput rates)

Limiting Values (e.g. maximum parameter values,maximum file sizes)

Resource Requirements (e.g. minimum hardware/software requirements to support the product)

The performance attributes assessed and the criteria for acceptance should be those agreed in the Certification Specification.

ATTRIBUTE ASSESSED	NOT ACCEPTABLE	ACCEPTABLE

COMMENTS :

Bibliography

[Bal 88] Balcer M J and Ostrand T J, The category-partition method for specifying and generating functional tests, Communications of the ACM, June 1988.

[Bal 89] Balcer M J and Hasling W H, et al,Automatic Generation of Test Scripts from Formal Test Specifications, Proceedings of the ACM SIGSOFT, Third Symposium on Software Testing, Analysis, and Verification, ACM press, Dec 1989.

[Boe 81] Boehm B W, Software Engineering Economics, Prentice Hall, New York, 1981.

[Boo 86] Booch G, Object-Oriented Development, IEEE Transactions on Software Engineering, Feb 1986.

[Car 88] Carroll J M and Olson J R, Mental Models in Human-Computer Interaction, Helander M,(editor), Handbook of Human-Computer Interaction, Elsevier Science Publishers, Amsterdam,1988.

[Chi 90] Chikofsky E J , Cross J H, "Reverse Engineering and Design Recovery : A Taxonomy", IEEE Software, Jan 1990.

[Chu 87] Chusho T, Test Data Selection and Quality Estimation based on the concept of essential branches for Path Testing, IEEE Transactions on Software Engineering,1987.

[Chu 89] Chusho T, Functional Testing and Structural Testing, Japanese perspectives in software engineering, System development laboratory, Hitachi Ltd., Kawasaki, Japan, 1989.

[Coa 90] Coad P and Yourdon E, Object-Oriented Analysis, Yourdon Press,New Jersey,1990.

[Cro 79] Crosby P, Quality is free, McGraw-Hill, 1979.

[DeM 88] DeMillo R A, McCracken M W, et al, Software Testing and Evaluation, Benjamin Cummings Publishing Company, Inc, 1987.

[Ell 87] Ellison J R, andPritchard J A T, Security in Office Systems,, NCC Publications,1987.

[Elm 73] Elmendorf W R, Cause Effect Graphs in Functional Testing, IBM Technical report, IBM Dev. Div, New York, 1973.

[ETN 89] ETNOTEAM, Case study definition report for VICTOR, SCOPE internal document, 1989.

[Fag 86] Fagan M E, Advances in Inspections, IEEE Transactions on Software Engineering, July 1986.

[Fie 89] Fiedler S P, Object-Oriented Unit Testing, Hewlett-Packard journal, April 1989.

[For 90] Forester T and Morrsion P, Computer Unreliability and Social Vulnerability, FUTURES magazine, Jun 1990.

[Fra 88] Frankl P G and Weyeuker E J, An Applicable Family of Data Flow Testing Criteria, IEEE Transactions on Software Engineering, Oct 1988.

[Fuj 89] Fujii R U and Wallace D R, Software Verification and Validation : An Overview, IEEE Software, May 1989.

[GEE 90] **RUTE** (not publicly available)
General Electric.

[Geh 86] Gehani N and McGettrick A D, Software Specification Techniques, Addison-Wesley, New York,1986.

[Ger 89] Gerhart S L, Assessment of Formal Methods for Trustworthy Computer Systems, Proceedings of the ACM SIGSOFT, Third Symposium on Software Testing, Analysis, and Verification , ACM press, Dec 1989.

[Goe 85] Goel A L, Software Reliability Models : Assumptions, Limitations and Applicability, IEEE Transactions on Software Engineering, Dec 1985.

[Gui 89] Guillemette,R.A. The CLOZE Procedure:An Assessment of the Understandability of Data Processing Texts, Information and Management,Volume 17,1989.

[Het 88] Hetzel B, The Complete Guide to Software Testing, QED Information Sciences, Inc., 1988.

[How 87] Howden W E, Functional Program Testing and Analysis, McGraw-Hill Book Co., 1987.

[IEE 89] Software Engineering Standards, (Third edition), Institute of Electrical and Electronic Engineers, Inc., 1989.

[Joh 89] Johnson G and Ravden S, Evaluating the Usability of a Human-Computer Interface,Holsted Press (New York),1989.

[Jon 90] Jones C.B,Systematic Software Development using VDM Prentice Hall,New Jersey,1990.

[Li 87] Li H F and Cheung W K, An Empirical Study of Software Metrics, IEEE Transactions on Software Engineering, Jun 1987.

[Lut 90] Lutz M, Testing Tools, IEEE Software, May 1990.

[Mal 89] Malevris N and Yates D F, Reducing the effects of infeasible paths in branch testing, Proceedings of the ACM SIGSOFT, Third Symposium on Software Testing, Analysis, and Verification,1989.

[McC 76] McCabe T J, A Complexity Measure, IEEE Transactions on Software Engineering, Dec 1976.

[McC 89] McCabe T J and Butler C W, Design Complexity Measurement and Testing, Communications of the ACM, Dec 1989.

[Mey 88] Meyer,B.,Object-Oriented Software Construction, Prentice Hall,New Jersey, 1988.

[Moy 90] Moynihan J A, O' Connor N M, Towards a method for assessing the functionality of a user-oriented functional specification, (To be presented at the IFIP Working Conference on Approving Software Products (ASP-90)), September 17 - 19, 1990.

[MRH 90] **Xray/DX** (publicly available)
Microtech Research.

[Mye 79] Myers G, The art of Software Testing, Wiley - Interscience, Chichester, 1979.

[NYU 90] **ASSET**
Computer Science Department,
New York University
New York,
USA.

[Par 89] Parrington N and Roper M, Understanding Software Testing, Ellis Horwood Ltd., 1989.

[PPE 90] **T** (publicly available)
Poston Programming Environments.

[Pre 88] Pressman R S, Software Engineering, A practioners approach, McGraw-Hill Book Co., 1988.

[PVL 90] **SPADE**
Program Validation Limited,
26 Queens Terrace,
Southampton,
S0 1, 1BQ,
UK.

[Rap 82] Rapps S and Weyeuker E J, Selecting Software Test Data Using Data Flow Information, IEEE Transactions on Software Engineering, Dec 1982.

[Rat 87] Ratcliff B, Software Engineering Principles and Methods, Blackwell Scientific Publications, 1987.

[Red 83] Redwine S T, An Engineering Approach to Software Test Data Design, IEEE Transactions on Software Engineering, Mar 1983.

[Ric 89] Richardson D, Aha S et al., Integrating Testing Techniques Through Process Programming, Proceedings of the ACM SIGSOFT, Third Symposium on Software Testing, Analysis, and Verification, ACM press, Dec, 1989.

[SCO[1] 89] SCOPE Technical Annexe, SCOPE consortium, Jan 1989.

[SCO[2] 89] Software Certification : State of the Art, SCOPE consortium, Sep 1989.

[SCO[1]90] SCOPE Phase 2 definition report, SCOPE consortium, Apr 1990.

[Sed 83] Sedgewick R, Algorithms, Addsion-Wesley Publishing Co., 1983.

[Shn 86] Shneiderman B, Designing the User-Interface,Addison-Wesley,1986.

[Ski 90] Skidmore S Farmer R. et al,SSADM Models and Methods, National Computing Centre of Great Britain, Manchester, 1990.

[Suf 89] Sufrin B, Effective Industrial Application of Formal Methods,in : Ritter G X, (editor),Information Processing 89-Proceedings of the IFIP 11th. World Computer Congress, North-Holland,Amsterdam,1989.

[Swa 81] Swamy M N and Thulasiraman, Graphs, Networks and Algorithms, John Wiley and Sons, Inc., 1981.

[UKM 89] Specification of software quality attributes
Software quality evaluation guidebook,
UK Managing dealer Microinfo Ltd,
P.O. Box 3, Omega Park, Alto, Hampshire, GU34 2PG, England,

[Ver 88] Verbruggen R, Object-Oriented Design, Does it Exist?, in Advance Papers of Third International Workshop on CASE, Cambridge, MA, Jul 1988.

[Ver 90] **LOGISCOPE, ASA**
Verilog,
150, Rue N. Vauquelin,
F 31081 Toulouse Cedex,
France.

[War 89] Ward W T, Software Defect Prevention Using McCabes Complexity Metric, Hewlett-Packard Journal, Apr 1989.

[Whi 88] Whiteside J and BennettJ, et al.,Usability Engineering:Our Experience and Evolution,in Helander,M. (editor),Handbook of Human Computer Interaction,Elsevier Science Publishers,1988.

[Woo 88] Woodcock J, and Loomes M, Software Engineering Mathematics , Pitman, London,1988.