

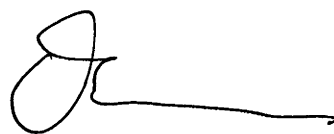
# **Testing Object-Oriented Software**

**Oscar Bosman**

A thesis submitted for the degree of  
Master of Science at  
The Australian National University

January 1999

Except where otherwise indicated, this thesis is my own original work.

A handwritten signature in black ink, consisting of a stylized initial 'O' followed by a long horizontal line that tapers to a point.

Oscar Bosman  
1 January 1999

*To Deborah, Hugh and Gareth*

---

# Acknowledgements

---

Without Dr. Brian Molinari's patience and consummate skill as a supervisor this thesis would have taken infinitely longer to reach a state that would not have been anywhere near as satisfactory. He kept me heading in the right direction and provided insights that both improved my own understanding of the subject and my ability to explain it. He patiently read and re-read drafts and ensured that my scattered and incomplete ideas were formed into a coherent whole. Prof. Heinz Schmidt motivated me to pursue the topic of testing object-oriented software, and he maintained an interest in my work and provided inspiration despite the difficulties of geography and his many other commitments.

Many people read and commented on drafts of chapters, including Jenny Adams, Neale Fulton, Lin Zucconi, John Colton and especially Kerry Taylor who provided timely feedback on the final draft. I particularly thank the other members of my supervisory panel: Prof. Krishnamurty who provided guidance in the early stages, and Dr. John Smith who patiently read and commented on everything I wrote. Any remaining faults are, of course, my own.

CSIRO DIT (now CMIS) awarded me a scholarship that funded the bulk of this work. I would like to show my appreciation to my colleagues at CSIRO with whom I had so many useful discussions. There are many other people who contributed to this work in ways large and small and I thank them all.

I would like to thank my family for the love and support that made this possible.



---

# Abstract

---

Object technology has changed software development. New programming languages have given rise to new software design techniques. Concurrently, new software development methods have been proposed. All of these changes have an impact on methods and tools for software testing. Most of the literature on testing object-oriented software has concentrated on aspects of object-oriented programming. We review this literature in the context of object-oriented programming and design.

Development processes that have been proposed for object-oriented software emphasise iteration and incremental delivery (Hutt 1994). However, they give little consideration to testing. In these processes, testing also needs to be iterative and incremental. This thesis presents an approach to integrating testing into iterative and incremental software development processes. The proposed approach inserts additional test design steps into the analysis, design and implementation activities and closes each iteration with a test review and execution step. We apply this approach to the Booch (1994) process.

New design techniques introduced by object technology offer new opportunities as sources for generating test cases. In this thesis, we propose a tool to generate test cases from one of these, namely state-transition diagrams. The generated test cases evaluate classes using the abstract notion of state expressed in a state-transition diagram. We develop a theory that reconciles this abstract view with the view of state expressed in object-oriented programs. The tool uses Chow's (1978) W-method to generate a set of test cases that "cover" the behaviour described by this design finite state machine. Then the class interface is used to generate an executable test program for the class. This test program uses sets of input values for the method arguments, and so combines domain testing with state machine based testing.

---

# Contents

---

<b>Acknowledgements</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Outline of the Thesis . . . . .	3
<b>2 Software Testing</b>	<b>5</b>
2.1 Basic Concepts . . . . .	5
2.2 Functional Testing Techniques . . . . .	7
2.3 Structural Testing Techniques . . . . .	10
2.3.1 Control-Flow Testing . . . . .	11
2.3.2 Data-Flow Testing . . . . .	12
2.3.3 Relating Test Criteria . . . . .	13
2.4 Testing using Finite State Machines . . . . .	13
2.5 Testing in the Large . . . . .	17
2.6 Adequacy . . . . .	17
2.6.1 Coverage . . . . .	18
2.6.2 Reliability Analysis . . . . .	19
2.6.3 Error Seeding and Mutation Analysis . . . . .	19
2.6.4 Process-based adequacy criteria . . . . .	20
2.7 Summary . . . . .	20
<b>3 Testing and Object-Oriented Programming</b>	<b>23</b>
3.1 Concepts in Object-Oriented Programming . . . . .	24
3.2 Testing Redefined . . . . .	26
3.3 Testing Objects . . . . .	29
3.3.1 Encapsulation . . . . .	29
3.3.2 Information Hiding . . . . .	31
3.3.3 State . . . . .	31
3.4 Inheritance . . . . .	34
3.4.1 Inheritance is a Many Splendour'd Thing . . . . .	35
3.4.2 Multiple Inheritance . . . . .	38
3.4.3 Abstract classes . . . . .	41
3.4.4 Inheritance and State . . . . .	41

---

3.5	Genericity . . . . .	42
3.6	Exceptions and Exception Handling . . . . .	44
3.7	Assertions and Software Contracts . . . . .	45
3.8	Integrating Classes and Subsystems . . . . .	47
3.8.1	Associations and Aggregations . . . . .	47
3.8.2	Inheritance and Dynamic-Binding . . . . .	49
3.8.3	Aliases . . . . .	50
3.9	Testing and Reuse . . . . .	51
3.10	Other Object Testing Issues . . . . .	52
3.11	Summary . . . . .	53
<b>4</b>	<b>Testing and Object-Oriented Software Development</b>	<b>55</b>
4.1	The Software Development Process . . . . .	55
4.1.1	Waterfall Process Model . . . . .	56
4.1.2	Evolutionary Process Models . . . . .	57
4.2	Testing and the Software Development Process . . . . .	59
4.2.1	A Process Model for Software Testing . . . . .	61
4.3	Object-Oriented Software Development Processes . . . . .	63
4.3.1	The Booch Process . . . . .	66
4.4	Testing and Object-Oriented Software Development . . . . .	67
4.4.1	Incorporating Testing in the Booch Process . . . . .	68
4.5	Other Object-Oriented Process Models . . . . .	70
4.6	Summary . . . . .	72
<b>5</b>	<b>Testing Objects as Finite State Machines</b>	<b>73</b>
5.1	State-Based Models for Class Design . . . . .	73
5.2	Representation and Implementation . . . . .	76
5.3	Testing Finite State Machines . . . . .	79
5.4	Test Case Generator . . . . .	82
5.5	Related Work . . . . .	82
5.6	Summary . . . . .	84
<b>6</b>	<b>STAT: Generating Test Cases from Object-Oriented Design</b>	<b>85</b>
6.1	State-Transition Diagrams . . . . .	86
6.2	Generating Test Cases . . . . .	88
6.3	Experience To Date . . . . .	91
6.4	Possible Extensions . . . . .	93
6.5	Summary . . . . .	94
<b>7</b>	<b>Conclusion</b>	<b>97</b>
7.1	Future directions . . . . .	98
7.1.1	Testing and Object-Oriented Programming . . . . .	98

---

7.1.2	Testing and Object-Oriented Analysis and Design . . . . .	98
7.1.3	Testing and Object-Oriented Software Development . . . . .	98
<b>A</b>	<b>An Example of STAT test cases</b>	<b>101</b>
<b>B</b>	<b>Overview of Sather</b>	<b>109</b>
B.0.1	A little background and history . . . . .	109
B.1	A brief overview of Sather syntax . . . . .	109
B.1.1	Classes . . . . .	109
B.1.2	Features . . . . .	113
B.2	Compiling and running Sather programs . . . . .	115
B.3	Programming environment . . . . .	115
B.4	Special features . . . . .	116
B.4.1	Inheritance in Sather . . . . .	116
B.4.2	Programming by contract . . . . .	117
B.4.3	Type safe down-casts . . . . .	118
B.4.4	Iteration abstraction . . . . .	118
B.4.5	Bound routines . . . . .	119
B.4.6	pSather . . . . .	120
B.4.7	The test class idiom . . . . .	120
	<b>Bibliography</b>	<b>123</b>



---

# List of Figures

---

2.1	An input domain $D$ showing a set of inputs causing failures $F$ , and test inputs ( $\times$ ).	7
2.2	An input domain partitioned into equivalence classes	8
2.3	A routine and its control-flow graph.	9
2.4	Subsumption relationships among test coverage criteria.	12
2.5	A finite state machine diagram for a traffic light.	14
2.6	A transition tree for the traffic light state machine in Figure 2.5.	16
3.1	A SORTED_LIST type.	27
3.2	Two states of a SORTED_LIST{INT} object	28
3.3	Specification for a SORTED_LIST class.	30
3.4	A Sather implementation of the SORTED_LIST class.	32
3.5	A state-transition diagram showing design states for SORTED_LIST.	33
3.6	An alternative state-transition diagram for SORTED_LIST.	34
3.7	Sather implementation of ACCOUNT - OVERDRAFT_ACCOUNT hierarchy	36
3.8	A signature clash resulting from multiple inheritance.	39
3.9	A signature clash resulting from convergent inheritance.	40
3.10	Inheriting design states in the QUEUE hierarchy.	43
3.11	The classes involved testing a polymorphic method call enqueue.	49
3.12	Changing the internal state of a class through an alias.	50
4.1	The waterfall process model	56
4.2	The spiral process model	58
4.3	The "V" model of software development	60
4.4	The major activities in each testing process	62
4.5	Coordinating testing and software development phases	63
4.6	The fountain model of software development	64
4.7	The Booch macro and micro processes	65
4.8	Adding testing to the Booch micro process	68
5.1	State-transition diagram for a TRAFFIC_LIGHT.	74
5.2	State-transition diagram for a STACK.	75
5.3	State-transition diagram for a BOUNDED_STACK.	75
5.4	Different views of object/class behaviour.	77

---

5.5	A <b>STACK</b> class implemented in Sather . . . . .	78
5.6	A <b>BOUNDED_STACK</b> class implemented in Sather . . . . .	79
5.7	Eliminating concurrent states. . . . .	80
5.8	A transition tree for <b>BOUNDED_STACK</b> . . . . .	81
6.1	A state-transition diagram for an <b>ACCOUNT</b> object. . . . .	86
6.2	Transition tree for the <b>ACCOUNT</b> state-transition diagram. . . . .	87
6.3	Test cases derived from the transition tree for <b>ACCOUNT</b> . . . . .	87
6.4	The Sather interface for querying an object's state. . . . .	88
6.5	Code generated for event sequence no. 3 in Figure 6.3. . . . .	89
6.6	Test data for the class <b>MONEY</b> . . . . .	90
6.7	Code generated for event sequence no. 11 in Figure 6.3. . . . .	91
6.8	A state-transition diagram for testing the class <b>TIME</b> . . . . .	92
6.9	Test data for the class <b>TIME</b> . . . . .	92
6.10	A state-transition diagram for an <b>OVERDRAFT_ACCOUNT</b> object. . . . .	93
B.1	An abstract class . . . . .	110
B.2	A concrete class . . . . .	111
B.3	Sather string class <b>STR</b> reuses the implementation of array of char. . . . .	112
B.4	Exception handling . . . . .	114
B.5	A "hello world" program in Sather . . . . .	115
B.6	Pre- and postconditions in Sather . . . . .	117
B.7	An example of a <b>typecase</b> statement . . . . .	118
B.8	A dot product of vectors (showing the use of <b>iters</b> ) . . . . .	118
B.9	Using a bound routine in an "applicative" context . . . . .	119
B.10	Part of a test class for the class <b>STACK</b> . . . . .	120

---

# Introduction

---

Software has bugs. This is a fundamental truth of software development today and in the foreseeable future. Until we can automatically manufacture software from requirements that perfectly capture the current and future needs of all potential users, this will continue to be so. Despite the many advances that have been made in the software development process, it is still the case that software has to be individually hand-crafted to suit its particular purpose. As a consequence, quality must be evaluated separately for each individual software item. Testing remains one of the major means of ensuring software quality.

An alternative approach to improving the quality of a product is to evaluate and improve the process of developing that product. This is a motivation for the increasing adoption of object-oriented technology. Proponents of object-oriented methods argue that they improve software quality, and reduce the need for testing and maintenance effort (Wirfs-Brock et al. 1990, Meyer 1988).

While there appears to be insufficient documented proof of these claims there is increasing anecdotal evidence of improved productivity and reliability. Recent conferences on object-oriented technology have devoted at least one session to business's experiences of the benefits of object-orientation. Many of the adopters of object technology first see benefits in porting, maintaining and enhancing software due to the modularity of software implemented in object-oriented programming languages. They also see benefit in the better modelling provided by object-oriented analysis, and the closer integration of analysis, design and implementation (Taylor 1992, Osmond 1994).

Part of the success of object technology is that it has given software developers techniques to handle increasingly complex software. It does this through interfaces that hide the complexity of lower levels, and classification whereby it is possible to abstract common elements and address them separately from the individual differences. But with an increase in complexity of systems comes an increase in their modes of failure.

There are, broadly speaking, three ways in which object-orientation has changed software development. The most obvious is the introduction of new programming languages built upon the concepts of objects and classes, inheritance, and polymorphism. The emergence of this programming paradigm has been followed by a profusion of



object-oriented analysis and design notations that support the discovery and design of objects and classes. More recently, some important work on the software development process (Boehm 1988, Gilb 1988, Booch 1986) has led to a number of object-oriented software development methods to facilitate object-oriented analysis and design.

The aim of this thesis has been to examine the effect of object technology on software testing. The discussion presented has been broken down into the same three areas of object-oriented programming, object-oriented analysis and design, and the object-oriented software development life-cycle.

Object-oriented programming is, in several respects, substantially different from procedural programming. Several kinds of errors in procedural programs do not occur in object-oriented programs. This is as a result of the use of encapsulation, polymorphism, stronger type checking, and other features of object-oriented programming languages. But these concepts also introduce new sources of errors. Further, a number of widely used testing techniques focus on the source code, so we need to rethink the application of those testing techniques to object-oriented programs. The research literature on testing object-oriented software has to date mainly concentrated on the effect of object-oriented programming constructs on testing.

Object-oriented design has introduced new notations and techniques, as well as adapting earlier ones. Objects and classes have become the main focus instead of functions of data. The artifacts of software design, like all those of software development, are potential sources for test cases. So there is an opportunity for testing to take advantage of these new design methods in designing test cases.

In this thesis we propose a tool to generate test cases from one artifact of object-oriented design: the state-transition diagram. State-transition diagrams describe the behaviour of objects and classes at a more abstract level than their implementation in an object-oriented programming language. We also develop a theory that reconciles the design view of object state with their implementation.

The introduction of object technology provides an opportunity to change the software development life-cycle, as well as changing the subprocesses within the life-cycle. No single model has emerged for an object-oriented life cycle, but there is an emphasis on iteration, incremental development and the evolution of the software product. Changing the process requires a reassessment of test activities and how they fit into the new life-cycle.

This thesis describes a framework for incorporating testing activities into an iterative and incremental software development process. We demonstrate the framework by applying it to the widely used software development process described by Booch (1994).

---

## 1.1 Outline of the Thesis

In this thesis we have assumed that the reader has a basic familiarity with software engineering and object-oriented technology, but most concepts are briefly defined when they are introduced or references are given. Chapter 2 presents a brief introduction to software testing which provides the necessary background and terminology for the rest of the thesis.

Chapter 3 reviews the current literature on testing and object-oriented programming. We discuss in turn encapsulation, object state and behaviour, inheritance, genericity and object interaction, and examine the implications for testing.

Process models for object-oriented software development are discussed in Chapter 4, and a strategy for incorporating testing into them is proposed. This chapter is derived from a paper given at the International Conference on Testing Computer Software, Washington D.C., June 1996 (Bosman 1996).

In Chapter 5 we return to the state-based testing of classes. We develop a theory that relates the design view of class state, in the form of state-transition diagrams, to the programming language view. This chapter is derived from a joint paper with Prof. Heinz Schmidt, given at the TOOLS Pacific '95 conference (Bosman & Schmidt 1995a). It is also available as a technical report (Bosman & Schmidt 1995b).

Chapter 6 proposes a tool to generate test cases from a state-transition diagram. This is an application of the ideas described in Chapter 5.

For examples of code we have used Sather, but the underlying ideas are more widely applicable. Sather affords a clear, concise, readily understandable, pure object-oriented view of the issues. Much of the work on testing object-oriented programs has been clouded by the peculiarities of C++. Sather permits the presentation of these ideas less muddled by implementation issues. The examples are intended to be straight forward, and the reader should have little problem translating them into their favourite object-oriented programming language. The main features of Sather are summarised in Appendix B.

---

# Software Testing

---

Software testing does not have underlying theoretical foundations. It is a collection of practices that have been found to discover defects in software. In examining software testing we find ourselves in a situation not unlike Kilgore Trout's dog standing on a mirror: when eventually it looks down, it thinks it is standing on thin air and jumps a mile (Vonnegut 1972). Despite the fact that testing does find bugs in software and does, in practice, improve its reliability, we have no sound basis for relating the current testing techniques to the quality of software.

In this chapter, we do a little "looking down" at the fundamental concepts and techniques of software testing. After first defining terms that will be used in the rest of this thesis, we give an overview of some of the many techniques traditionally used for testing procedural programs. These testing techniques will be re-examined in an object-oriented setting in the next chapters.

## 2.1 Basic Concepts

Testing is the process of exercising software to detect differences between its actual and required behaviour (IEEE 1990). The required behaviour may be explicitly stated in terms of a requirements document or formal specification. It is immaterial to this discussion how these requirements are derived or what form they take. Testing only requires that we have some means of recognising correct and incorrect behaviour of the software under test.

To introduce some notation, let  $P : D \rightarrow R$  be a program with inputs from domain  $D$  and outputs in range  $R$ . That is,  $D$  is the set of all (valid and invalid) inputs that can be applied to  $P$ , and when applied to  $P$  produce outputs in the set  $R$ , so for  $d \in D$ , we have  $P(d) \in R$ . Note that  $D$  usually models some real-world set and is typically unbounded: for example, the inputs for a compiler includes all strings of text of finite length. In practice it must be possible to represent the elements of  $D$  in a computer system, so  $D$  is the set of strings of length up to some nominated value  $k$ . While  $D$  is finite, for realistic values of  $k$  it is very large.

Now let  $S : D \rightarrow R$  be a function defining the requirements on the output of  $P$ , as derived from its specification. We say that  $P$  is *correct*, if for all  $d \in D$ ,  $S(d) = P(d)$ .

A *test case* is an input value and its expected output, or in our formalism, a test case is a pair  $(d, r) \in D \times R$  such that  $S(d) = r$ . We say that a test case is *successful* if it causes the program to produce an incorrect result, that is if  $P(d) \neq r$  (Myers 1979, Pressman 1992)<sup>1</sup>. The goal of testing is to find successful test cases, those  $(d, r)$  for which  $S(d) = r$  and  $P(d) \neq r$ .

In standard software engineering terminology (IEEE 1990), the situation where a program produces an incorrect result for some  $d \in D$  is called a *failure*. A failure is a run-time manifestation of a *fault* in the program code. A fault is caused by an *error* in the implementation process; for example, a misunderstanding of the specification or a typographical mistake.

Marick (1994) points out that in order for a fault to manifest as a failure, a test case must meet three conditions: it must exercise the code containing the fault (*reachability*), it must cause the faulty code to produce a different internal state to that which would be produced by a correct version (*necessity*) and, the incorrect internal state must propagate to some externally visible behaviour (*propagation*).

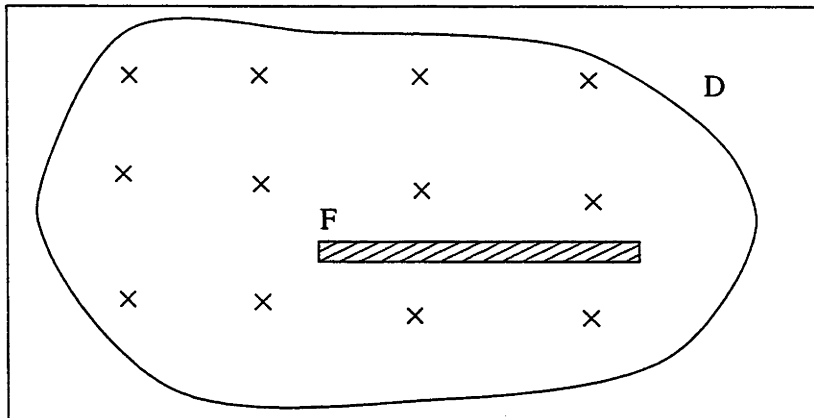
Test cases are collected into *test suites*. A *test criterion* (also called *test requirement* or *test condition*) describes some aspect of the software that should be tested. For example, test criteria for a list search routine could include a match at the beginning of a list, a match at the end, and no match. Test criteria are used to guide the test process in that, having identified test criteria for a program, a tester creates test cases by selecting specific values that satisfy test criteria. Based on the above example test criteria, a tester might select the test cases:

- find  $a$  in the list  $(a, b, c, d, e)$ , expected result: 1
- find  $e$  in the list  $(a, b, c, d, e)$ , expected result: 5
- find  $h$  in the list  $(a, b, c, d, e)$ , expected result: error

A test criterion may be satisfied by several test cases and one test case may satisfy several criteria.

In realistic applications,  $D$  is typically huge and resources available for testing limited, so the sample of test inputs ( $T \subset D$ ) is necessarily sparse. If there are relatively few faults in the software then the set of inputs causing a failure ( $F \subset D$ ) will likely also be small, so the probability of a selecting test case causing a fault ( $T \cap F \neq \emptyset$ ) is very low. This is the situation illustrated in Figure 2.1. The set of test inputs, marked by the  $\times$ 's, was selected according to some criteria that misses the failure-causing inputs in the shaded area. The art of software testing is in developing and applying practical techniques for identifying test criteria that are likely to find faults.

<sup>1</sup>Some authors use the reverse definition, e.g. (Ghezzi et al. 1991)



**Figure 2.1:** An input domain  $D$  showing a set of inputs causing failures  $F$ , and test inputs ( $\times$ ).

Testing has been described as the “accretion of confidence” (Macro 1990). But as we have just seen there is no real basis for this confidence and this view of testing is little more than a psychological crutch for the software developer. Dijkstra made this clear when he pointed out that:

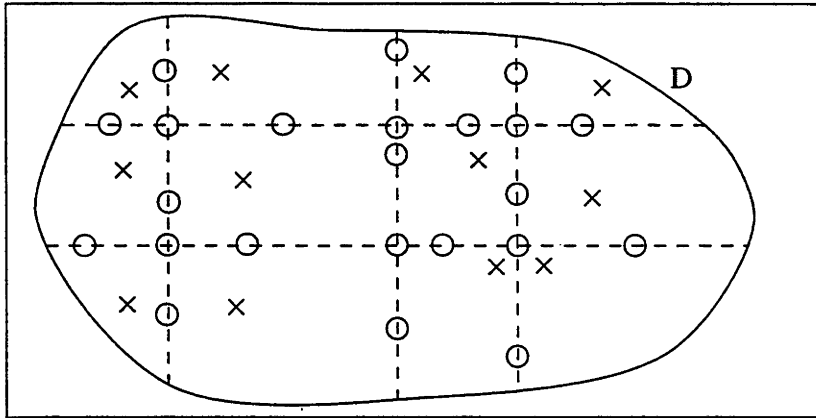
Program testing can be used to show the *presence* of bugs, but never to show their absence. (in Dahl et al. 1972)

Testing techniques are usually divided into those that analyse the specification without reference to its implementation (*functional* or *black-box*) and those that analyse the code (*structural*, *white-* or *glass-box*). The next two sections describe some techniques in each category. Section 2.4 describes an alternate testing technique based on finite state machines. After individual units of code have been tested, they must be integrated to produce a functioning system. Section 2.5 of this chapter looks at integration testing and system testing. Finally we discuss techniques for assessing the adequacy of a test suite or the likelihood that it has not missed significant faults.

## 2.2 Functional Testing Techniques

Functional testing refers to techniques that derive test cases without reference to the code. Since these techniques ignore the implementation language, they can be equally applied to software developed in an object-oriented programming language. We briefly review some of the more well-known techniques.

Perhaps the most commonly used functional testing technique is *specification-based testing*. This requires some form of requirements specification document. It involves analysing the document to detect each required function, output or behaviour of the software and devising a test case to determine its presence in the program. When



**Figure 2.2:** An input domain that has been partitioned into equivalence classes, showing test inputs suggested by equivalence partitioning ( $\times$ ) and boundary-value analysis ( $\circ$ ).

the requirements specification is presented in a formal specification language it is possible to automate this process and a number of tools have been developed based on this approach. *Documentation-based testing* is similar, but instead of working from a requirements document, it analyses the user documentation such as user manuals, reference manuals, user guides, and on-line help.

*Random testing* is a functional testing technique that generates test cases by randomly selecting inputs from the program domain  $D$ . It is not highly regarded as an effective method for finding faults. As we have seen in Figure 2.1, if there are relatively few faults in the software then the probability of a test case causing a fault ( $P(T \cap F \neq \emptyset)$ ) is very low. On the other hand, if there are many faults, for example during earlier stages of development, then random testing may be reasonably successful, and although there is no reason why it should be any more successful than other techniques, it may be easier to generate test cases randomly. Random testing is, however, the basis for reliability analysis (see Section 2.6.2 below), and is an essential ingredient in the cleanroom software development process (Lokan 1993).

In *equivalence partitioning* the input domain of the program is partitioned into classes that are expected to produce similar behaviour or output. Test cases are then selected so that all equivalence classes are represented in the test suite. In Figure 2.2 the input domain  $D$  has been partitioned into subdomains indicated by the dashed lines. Test inputs  $\times$  are selected from each subdomain.

The underlying assumption is that in terms of errors in the logic of the program, we can reasonably expect all values in an equivalence class to be handled in the same way and exercise the same logic and thus any one of them will serve as a test case. For example, if the software under test is a square root function, and the input domain is the set of representable real numbers, this would be partitioned into negative and

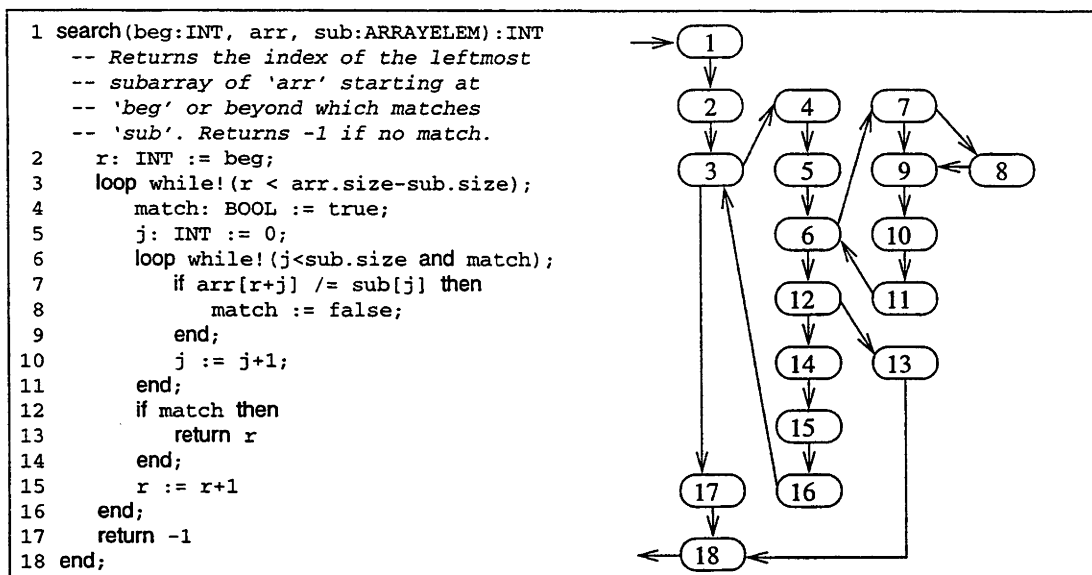


Figure 2.3: A routine and its control-flow graph.

non-negative numbers, being invalid and valid inputs respectively. Another example is the list search in Figure 2.3 and described in the next section; if we treat the comment as a specification then equivalence partitioning would find five equivalence classes:

- the sequence `sub` does not occur in the array `arr`,
- the start index, `beg`, is before the start of all occurrences of `sub` in `arr`,
- `beg` is after the start of all occurrences of `sub`,
- `beg` is between two occurrences of `sub` in `arr`, and
- a class of invalid values, in which `beg` is not in the range of indices for `arr`.

*Boundary value analysis* is based on the recognition that many errors occur at the boundaries of equivalence partitions. A common example is the “off-by-one” error, that can occur when iterating through arrays accessing one past the last element. In Figure 2.2, boundary value analysis selects test case inputs  $\bigcirc$  on the boundaries between subdomains. Note that an intersection between boundaries is also a boundary. In the square root example in the previous paragraph, boundary value analysis would suggest zero and numbers a small increment above and below zero as test cases. In the substring search example, boundary value analysis leads to test cases where `beg` is the first or last index of `arr`, the beginning of an occurrence of `sub` in `arr`, where `sub` occurs at the beginning and end of `arr`, where `sub` is a single character or the same as `arr`, and so forth.

By regarding blocks of statements as “black-boxes”, these two techniques can also be applied to code. Each predicate in the code partitions the set of inputs according to which branch they would cause to be executed. When used in this way, equivalence partitioning is similar to branch coverage (see Section 2.3.1 below). Boundary value analysis suggests test cases that are at the edges of these partitions. Given the statement

```
if a > 0 then ...
```

test cases should be selected that cause  $a$  to be 0, a small increment above 0 and a value below 0.

There are several other functional testing techniques described in the literature. Cause-effect graphing relates input conditions to their effects (Myers 1979, Pressman 1992, Ghezzi et al. 1991). When the input has a well defined syntax, such as compilers or text processors, syntax testing (Hetzel 1988, Ghezzi et al. 1991) can be used to generate test cases that have examples of all constructs in the input grammar. State-based testing is discussed separately in Section 2.4.

Unlike purely structural testing techniques, functional testing techniques can expose errors of omission in the implementation. Structural and functional testing are to some extent complementary and are ideally used together. It is usually recommended that functional tests are devised first and then these are supplemented by structural tests.

## 2.3 Structural Testing Techniques

Structural testing techniques examine the implementation of software in order to derive test conditions. Typically these methods classify structures found in the code, and then require that the test suite *cover* these constructs, that is, exercise all occurrences of them. The simplest example is to recognise functions and procedures in the code and ensure they are all exercised by the test suite. This is known as *call coverage*.

Control-flow testing and data-flow testing are the two main structural testing techniques. They focus on the flow of control in a program and, in the latter, the flow of data through a program. Mutation coverage is a novel technique that focusses on exposing likely errors. These techniques are described in the rest of this section.

The structural testing techniques described in this section can be, and are, applied to object-oriented programs. However object-oriented programming languages have additional structure and the effectiveness of structural testing techniques for object-oriented programs needs to be reconsidered (see Chapter 3).



### 2.3.1 Control-Flow Testing

Control-flow testing derives test criteria from the control-flow graph of a program. Figure 2.3 shows a sub-string matching routine, written in Sather<sup>2</sup>, and its control-flow graph. Each node in the control-flow graph represents a statement and arrows indicate the possible execution sequences of the statements.

Control-flow test criteria are well known and widely used, and described in many texts on software testing or software engineering: see for example, (McDermid 1991, Pressman 1992, Sommerville 1992, Myers 1979) or, for a thorough discussion, (Beizer 1990). The most commonly used control-flow test criteria are:

**Statement coverage** requires that the test suite executes all statements in a program.

**Branch coverage** (or *edge coverage*) requires the test suite execute all alternative paths at a branch. For example, in Figure 2.3, any path that includes the sequence of statements corresponding to the nodes (... ,6,7,8,9,...) exercises all statements in that part of the control-flow graph, but not the branch (... ,7,9,...).

**Condition coverage** requires that each simple term in a compound boolean expression evaluate to all possible combinations of true and false. Compound boolean expressions are those containing the operators `and` or `or`, such as the expression `j < sub.size and match` on line 6 in Figure 2.3.

**Path coverage** requires that the test suite exercise all possible paths in the control-flow graph. If the program contains loops then there is infinite or at least impractically large number of paths, so path coverage is impossible to achieve. In practice the weaker loop coverage criterion is used.

**loop coverage** requires that there are test cases that exercise each loop at least 0, 1 and the maximum number of times (or where the loop has no maximum, a large number of times).

There are a large number of commercial tools available that can evaluate the statement coverage, branch coverage and even loop coverage of a test suite for a number of programming languages. These work by instrumenting the code under test to measure the number of times the program counter reaches each statement, branch or loop. A tester can use the results to find sections of code that were not exercised and then design new test cases to reach those untested sections.

---

<sup>2</sup>In the figure, the expression `while! (...)` is a Sather iterator (see Appendix B). In this example the two statements containing it are semantically equivalent to while loops.

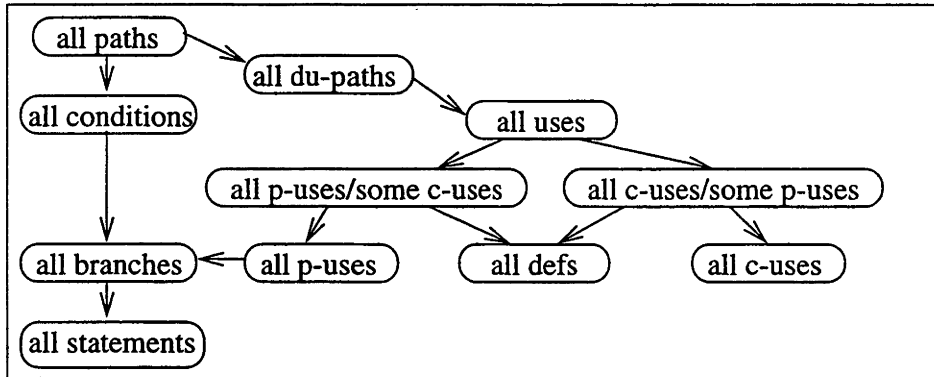


Figure 2.4: Subsumption relationships among test coverage criteria.

### 2.3.2 Data-Flow Testing

Data-flow testing (Frankl & Weyuker 1988, Ntafos 1988) is actually a collection of structural testing techniques that attempt to find errors by exercising the flow of data through a program. It borrows from the data-flow analysis techniques used in optimising compilers. In the control flow graph of a program or routine, a *definition* of a variable  $x$ ,  $\text{def}(x)$ , is a node in which the value of  $x$  is set or modified (for example, the statements 2 and 15 in Figure 2.3 are both definitions of the variable  $r$ ). A *use* of  $x$  is a node in which the value of  $x$  is used. If  $x$  is used in a calculation, this is a *c-use* (we write  $\text{cu}(x)$ ), or if in a predicate, it is a *p-use* ( $\text{pu}(x)$ ). The reason for the distinction is that predicates occur at a branch in the routine's control-flow graph. In Figure 2.3, statements 7 and 15 are c-uses of  $r$  and statement 3 is a p-use of  $r$ . A *du-path* is a path in the program's control flow graph in which there is a definition of the variable  $x$  followed by a use (for example, in Figure 2.3, the sequences of statements (2, 3) and (15, 16, 3, 4, 5, 6, 7) are both du-paths for the variable  $r$ ).

Static data-flow analysis, such as performed by the UNIX utility `lint`, seeks to detect those paths with successive definitions of the same variable, or with uses lacking a preceding definition, as they indicate implementation faults. In testing, data-flow test criteria assume that these have been removed, and only consider du-paths.

A test suite satisfies *all du-paths coverage* if, for all variables in the program, every du-path is exercised. It satisfies *all defs coverage* if for every variable, every definition is exercised. Similarly, a test suite exercising all uses of all variables satisfies *all uses coverage*. If a test suite exercises the subset of du-paths that includes all c-uses or p-uses, it satisfies the *all c-uses coverage* or *all p-uses coverage*, respectively. If a test suite satisfies both the all c-uses and the all defs coverage, that is, if for any  $\text{def}(x)$  there is no path to any  $\text{cu}(x)$ , then a path to some  $\text{pu}(x)$  is exercised, it satisfies *all c-uses/some p-uses*. The *all p-uses/some c-uses* is similar.

### 2.3.3 Relating Test Criteria

We say that one test criterion  $A$  *subsumes* another,  $B$ , if every test suite that satisfies  $B$  also satisfies  $A$ . Clearly the subsumes relation is transitive and reflexive. Of the control-flow test criteria, branch coverage subsumes statement coverage, and both are subsumed by condition coverage. Path coverage subsumes all other control-flow testing criteria.

The subsumption relationships between the various data-flow and control-flow coverage criteria are illustrated in Figure 2.4 (Ntafos 1988). An arrow indicates that one criterion subsumes another, so for example, path coverage subsumes du-path coverage which in turn subsumes all uses coverage.

## 2.4 Testing using Finite State Machines

Finite state machines are a useful tool for specifying, designing, implementing and testing software. They can describe interfaces, such as network protocols and GUIs, and simple grammars such as regular expressions. Several tools exist for generating working code from finite state machine descriptions. Methods for specifying and building reliable software based on refining finite state machine specifications have been proposed (Leveson et al. 1994, Zucconi & Reed 1996).

Finite state machines (Hopcroft & Ullman 1979) represent a system as a set of states and transitions between states. The output of a finite state machine depends upon its current state as well as the input, and in response it may also change its state. Our development below and elsewhere in this thesis is based on the Mealy machine which associates outputs with transitions, rather than the Moore machine which associates outputs with states. The two have been shown to be equivalent (Hopcroft & Ullman 1979).

Formally, a (deterministic) finite state machine is described by the sextuple

$$F = \langle I, O, S, s_o, f, g \rangle$$

where

$I$  is the set of input (or event) symbols,

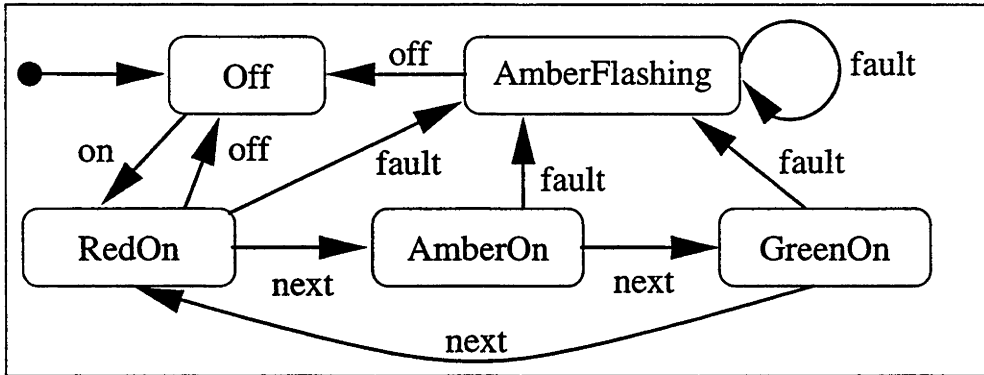
$S$  is the set of state symbols,

$O$  is the set of output symbols,

$s_o \in S$  is the initial state,

$f : I \times S \rightarrow S$  is the transition function, and

$g : I \times S \rightarrow O$  is the output function.



**Figure 2.5:** A finite state machine diagram for a traffic light.

Both the transition function  $f$  and the output function  $g$  can be partial, that is, not all combinations of states and inputs need cause a transition or produce an output. For a finite state machine  $S, I$  and  $O$  are all finite. When it is started, a finite state machine is in the state  $s_0$ . If there is no sequence of transitions from  $s_0$  to a state, then that state is *unreachable*. *Dead states* are those for which there is no sequence of transitions back to  $s_0$ . A finite state machine with no dead or unreachable states is *strongly connected*. In practice, almost all finite state machines used in software are strongly connected (Beizer 1990, page 373).

Figure 2.5 shows a finite state machine for a traffic light. It has five states each indicated by a labelled box. The transitions are represented by arrows and labelled with the inputs that cause them. Initially it is in the *Off* state. Turning the power on (input *on*) causes it to transition to a state in which the red signal is on (state *RedOn*). The *next* input (from a timer and/or traffic sensor in the road) causes it to cycle through the green (state *GreenOn*), amber (state *AmberOn*) and red signals. In any state it can accept a *fault* input which sets the FSM in the *AmberFlashing* state. Further *fault* inputs return it to the *AmberFlashing* state, until turning the power off (input *off*) sets it in the *Off* state.

Finite state machines succinctly describe the behaviour of systems and therefore are a good source of test criteria. Beizer (1990) and Marick (1994) recommend that, where possible, testers create a finite state machine model of the software to be tested, if one wasn't produced in its development. Where there is a tool to generate test cases from a finite state machine specification, such as the one which will be described in Chapter 6, this can be a very efficient practice.

There are a number of test criteria that can be used for finite state machine specifications, including:

- entering all states,

- exercising all transitions,
- exercising all pairs of transitions,
- exercising all sequences of n transitions and
- exercising all paths in a transition tree.

Deriving transition trees is an important testing technique since a test suite derived from transition tree can show the equivalence of an implementation and its finite state machine specification. Thus we describe this in detail here. In Chapters 5 and Chapter 6 we use this technique to generate test cases for testing classes.

We say two finite state machines with the same sets of input and output symbols,  $Spec = \langle I, O, S, s_o, f, g \rangle$  and  $Impl = \langle I, O, S', s'_o, f', g' \rangle$ , are *equivalent* if any sequence of inputs from  $I$  produces the same sequence of outputs on both finite state machines (Hopcroft & Ullman 1979). Chow (1978) showed that if one finite state machine  $Spec$  is minimal, we can select a finite set of input sequences  $T \subset I^*$  such that if  $Spec$  and  $Impl$  are equivalent over  $T$  then they are equivalent for all input sequences  $I^*$ . A number of refinements have reduced the size of the test set  $T$  generated by Chow's "W-method" (Fujiwara et al. 1991, Bernhard 1994). In essence the W-method derives a transition tree for  $Spec$ .

To construct a transition tree for a finite state machine we can proceed as follows:

1. The initial state is the root node of the tree.
2. For each new, non-terminal node, an edge is drawn for every transition out of that state to a new node representing its target state.
3. For each node just drawn, if it represents a state already visited, then it is marked as terminal.
4. Repeat until there are no new, non-terminal nodes.

The transition tree thus constructed for the traffic light finite state machine in Figure 2.5 is shown in Figure 2.6. The test cases are formed by taking all paths and subpaths in the transition tree. The test case input is the sequence of events from the transitions along the path and the result of the test case is the state in the end node of the path. A set of test cases from the transition tree in Figure 2.6 is:

event sequence: $\emptyset$ ;	final state: <i>Off</i>
event sequence: on;	final state: <i>RedOn</i>
event sequence: on, off;	final state: <i>Off</i>
event sequence: on, fault;	final state: <i>AmberFlashing</i>
event sequence: on, fault, off;	final state: <i>Off</i>
event sequence: on, fault, fault;	final state: <i>AmberFlashing</i>

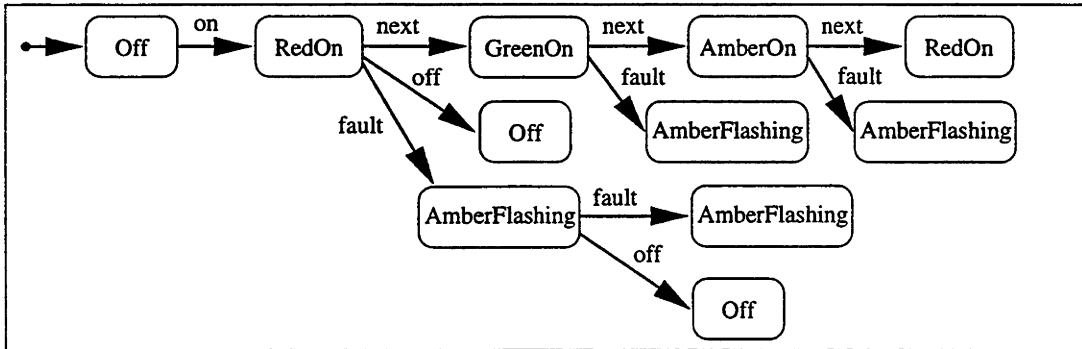


Figure 2.6: A transition tree for the traffic light state machine in Figure 2.5.

- event sequence: on, next; final state: *GreenOn*
- event sequence: on, next, fault; final state: *AmberFlashing*
- event sequence: on, next, next; final state: *AmberOn*
- event sequence: on, next, next, fault; final state: *AmberFlashing*
- event sequence: on, next, next, next; final state: *RedOn*

Test suites derived from transition trees quickly get large as the number of states and inputs increase. While the length of each test sequence is bounded by the number of states, the minimal number of tests is at most  $|S|^2 \times |I|^{|S|-|S|+1}$  and the total length of all test sequences is at most  $|S|^2 \times |S'| \times |I|^{|S|-|S|+1}$  (Chow 1978). This is practical for small state machines, but large state machines must be factored or otherwise reduced. More recent work has developed algorithms that reduce the size of the required test suite under certain conditions (Fujiwara et al. 1991, Bernhard 1994).

The test suite derived from a transition tree tests that an implementation of a finite state machine supports all the transitions in its specification. But it is also an error if the implementation allows extra transitions (sometimes called “sneak paths”). The test criteria for additional transitions and states is to try all illegal events for each state. In Figure 2.5 the event *b* cannot be accepted while in state *State 2*. To test for the possibility of a sneak path from *State 2* to *State 1* we would add the following path to the above test suite:

- event sequence: on, fault, next; final state: *error*

State charts are a development of the state machine diagramming notation that allows for hierarchies of nested state machines, conditional transitions, and other extensions (Harel 1987). State charts have been incorporated in to several object-oriented development methods (also known in this context as “object charts” or “state-transition diagrams”), where they are used for describing the dynamic behaviour of objects (see,

---

for example, Rumbaugh et al. 1991, Booch 1994). The state-based testing of classes is discussed in Chapter 3.

## 2.5 Testing in the Large

The test techniques so far described can be applied to single programs. Larger software systems can be complex to test as well as build. One aim of software engineering is to develop methods for handling this complexity. Testing is, in the main, a software process issue and these aspects are further examined in the context of object-oriented software development in Chapter 4. However, it will be helpful to outline the main issues and provide a few definitions here.

Large systems are developed in modules which are individually tested. If a module under test depends upon the services of another, that functionality can be provided, for the purposes of testing, by a *stub*. Usually a *test driver* is required to supply inputs, run the module and collect the results.

Once individual functions and modules are tested, they must be integrated into the complete system. The concern of *integration testing* is to validate the intermediate stages. As modules are combined and the items being tested become larger, there is a trend toward functional testing over structural testing and an increasing focus on testing module interfaces (Harrold & Soffa 1991). The central issue is the order of combining modules. The two most often described schemes are *top-down* and *bottom-up*. In the former, the process begins with the system interface; in the latter, with those core modules that provided services for many other modules. Stubs and drivers are incrementally replaced by modules and the configuration tested until the system is complete. Hetzel (1988) recommends integrating small skeletons support increments of functionality and then building out from these skeletons. In top-down integration the interface is most heavily tested, in bottom-up it is the core services.

When complete, the system as a whole is tested. This phase is called *system testing*. As well as confirming the correct functioning of the software system, this may also involve testing other factors such as performance, reliability, and security.

## 2.6 Adequacy

If a test suite fails to uncover any new errors, one of two conclusions can be drawn: either the program is of sufficient quality and reliability or the test suite is inadequate. A test suite is *adequate* if it could expose any possible failure of the program. When the test suite for a program is adequate, testing can be stopped (Weyuker 1988).

A test suite consisting of the complete set of possible input values is trivially adequate, but complete testing of all but trivial software is infeasible. As we have already seen, the input domain is typically huge, so if a program is to be run with all possible

inputs and the results checked against the expected output, this could take arbitrarily long. Consider, for example, validating a compiler by compiling every possible string of characters.

On the other hand, a white box approach fares little better: to exercise every combination of paths in even a moderately complex program leads to a combinatorial explosion in the number of test cases. To take a very simple example, a program containing a single branch inside a loop that can execute at most 20 times has  $2^0 + 2^1 + \dots + 2^{20} = 2,097,151$  paths, each requiring a separate test case. Software is considerably more complex than this. Even if all paths are exhaustively tested, the software may still have errors due to unimplemented functions.

For efficiency, testers aim to build a test suite that is minimal, since test cases take time to run as well as time to develop. However, it is not easy to determine a minimally adequate test suite. In practice, a decision to stop testing must be made on the basis of the cost of developing additional test cases versus the risk of any remaining undetected faults. This will often depend on the software development context: the risk and cost of software failure, the quality requirements of the customer or developer, or simply the time and resources available for testing. This is a project management issue.

In the rest of this section, we briefly examine some techniques that have been used to determine test suite adequacy.

### 2.6.1 Coverage

The thoroughness with which control-flow and data-flow testing has been carried out can be measured by taking the percentage of constructs that were actually exercised by the test suite. For example, a test suite that exercises every statement in a program is said to have achieved 100% statement coverage. Coverage is probably the most commonly used technique for determining the adequacy of a test suite. It relates directly to several of the test techniques described above: branch, loop, and du-path coverage are often used, and tools exist for several programming languages for determining test suite coverage (for example, see (Marick 1992*b*, Horgan et al. 1994, Frakes et al. 1991)).

A typical target for testing is 100% statement coverage and 85% branch coverage (McDermid 1991). Weyuker (1988) states that an adequate test suite should, at a minimum, exercise 100% of the executable statements and Marick (1994) also sets 100% feasible coverage as target, but more for psychological reasons. Beizer (1984) is even more adamant:

If you still believe it's possible to test and integrate a system without meeting the minimum standard of 100% coverage, then there's no point in continuing with this book. There is a philosophical chasm between this writer and the reader that only the reader's future bitter and expensive experience will bridge.



---

Although coverage is widely used and often recommended, there have been few empirical studies of its effectiveness. It is still not clear whether high test coverage leads to a high rate of fault detection (Hutchins et al. 1994, Horgan et al. 1994).

### 2.6.2 Reliability Analysis

Another basis for a decision to stop testing is reliability analysis. This technique models the occurrence of failures as a statistical process, such as a Poisson process (Musa & Ackerman 1989), binomial distribution (Levendel 1991) or Markov chain (Whittaker & Thomason 1994), to derive a statistically based reliability measure such as mean time to failure (MTTF). When a predetermined reliability target is reached, testing is considered complete. Reliability analysis is (at least in theory) the only form of testing used in the cleanroom software development process (Lokan 1993).

In order to apply statistical techniques, reliability analysis makes a number of assumption about the random distribution of faults and failures. This has been much criticised since software faults are due to logical failures in the development process rather than random processes such as fatigue. Further, software faults cluster: it is a well established observation that 80% of faults occur in 20% of code (Levendel 1991, Walsh 1992). However, in large systems of high complexity, it may be reasonable to approximate failure distribution with a random process.

### 2.6.3 Error Seeding and Mutation Analysis

Error seeding is a method of testing the test suite by determining its effectiveness on a known set of faults. A representative set of known faults is inserted into the software, and the number of these faults found by the test is determined. This is then used to estimate the number of undiscovered faults in the system under test.

Mutation analysis is a more sophisticated version of this technique. It involves generating copies of the program with a fault inserted (known as *mutants*). Two forms of mutation are most commonly used: operator replacement (for example  $a := b + c$  could be mutated to  $a := b - c$ ) and variable replacement (in which original statement could be mutated to  $a := c + c$ ). In mutation analysis, a test suite is adequate if it can distinguish the program from all possible mutants.

Mutation analysis has been shown to be much more stringent than other testing techniques. However, generating and testing mutants is an expensive process, as the number of mutants is proportional to the square of the sum of the statements and variables. In practice it is generally not feasible, but a number of techniques are under development that may make the mutation analysis of larger commercial projects possible (see, for example, Duncan 1993).

Mutation analysis assumes that by the time programs are ready for testing, only such simple faults remain (in the literature on mutation this is known as the “com-

petent programmer hypothesis”). Mutation analysis seeks to establish the ability of a test suite to expose these kinds of faults. More recently, mutation coverage has been proposed as a means of designing test cases that are likely to expose common mistakes by programmers (Marick 1992*b*).

Proponents of error seeding and mutation analysis argue that, since it is based on an analysis of likely faults, it is more efficient at finding errors. However, the technique as described here concentrates on errors with very localised extent. It is interesting to compare Knuth’s detailed analysis of the errors found in T<sub>E</sub>X software (Knuth 1992). Only two of Knuth’s 15 categories of errors (constituting less than 8% total errors listed) would be exposed by mutation testing. Further, faults in these two categories tended to occur towards the earlier parts of development, rather than the end as would be expected under the “competent programmer hypothesis”.

#### 2.6.4 Process-based adequacy criteria

Myers advocates that a decision to stop testing be based on an analysis of the testing process (Myers 1979, Chap. 6). There are two core ideas. The first is to estimate the number of faults that are expected to be found, and then test until that many are uncovered. The second is that testing should not stop while the rate at which faults are being found is increasing.

The number of faults that are expected to be found is calculated from the expected number of faults in the software and the effectiveness of the test process and testers. The number of faults in the software under test can be estimated from historical data or by error seeding techniques (Myers 1976). If  $S$  faults are inserted into the software, and the test suite exposes  $s$  of them as well as  $n$  indigenous faults, the actual number of indigenous errors ( $N$ ) can be estimated by  $N = (S \times n)/s$ . The effectiveness of a tester is also determined from historical records. Thus, for a 2000 line software product developed by a team that had an error rate of 5 bugs per 100 lines of code in a previous project, a tester with a effectiveness of 80% should expect to find 80 bugs. So, testing should continue until that many bugs are found and the rate of finding bugs falls below a previously set threshold.

## 2.7 Summary

In Section 2.1 we said that a fault occurs in a program  $P$  when  $S(d) \neq P(d)$  for some  $d$  in domain  $D$  of the program. We base our confidence in the correctness of a program if  $S(d) = P(d)$  for all  $d$  in our test suite  $T \subset D$ . In this chapter we have described a number of heuristics for choosing a good  $T$ , that is, one which is likely to expose faults.

In general, the techniques described in this chapter assist in finding test criteria. It may be that a single test case may satisfy more than one test criteria. An efficient test

suite minimises the number of test cases. This can be important if testing is repeated, as in the iterative style recommended for developing object-oriented software.

Yet, despite some spectacular disasters, we do build software that does work and is, in the main, reliable. Experience shows that pay packets, electricity bills and bank statements are nearly always correct, telephones and electronic ignition usually work, and I have yet to meet a bug in my use of this word processor. So, like the dog on the mirror, software development and testing does “stand up”, although we don’t understand why and when we stop to examine it in detail, we have little reason for confidence in it. It is still the case that in the absence of a unifying theory, a tester’s best tool is experience. The test techniques described here are just a part of the tester’s armoury.

---

# Testing and Object-Oriented Programming

---

Most traditional testing techniques were developed in the context of testing software written in procedural programming languages. So naturally the question arises: to what extent are these techniques applicable to other programming paradigms, in particular object-oriented programming? In this chapter we look at the effect of object-oriented programming on testing.

Of the testing techniques described in Chapter 2, clearly Structural testing techniques need to be re-evaluated, since they are based on source code. In fact it is worth re-examining all testing techniques, since object-orientation affects so many parts of the software development process.

Much of what has been written to date on testing object-oriented software has focussed on aspects of object-oriented programming languages that affected testing. The main part of this chapter is a discussion of the ways in which aspects of object-oriented programming languages interact with testing, both from the point of view of testing objects and testing object interactions. The presentation aims to be language independent. In particular, memory management issues are not discussed as they hardly arise in languages with garbage collection such as Smalltalk, Eiffel, Java and Sather. Spruler (1994) and Hunt (1994) discuss testing issues for memory management in C++.

While the terminology of software testing has largely been settled, object-oriented technology is an area of active research, much debate and fluid terminology. The first section briefly covers those concepts and terms from object technology that will be needed in the rest of this thesis. In Section 3.2 we shall revisit the formal analysis from Section 2.1 of the previous chapter in the context of testing objects. Section 3.3 looks at encapsulation and the testing of individual objects or classes. The remaining sections examine in turn the effects on testing of inheritance, genericity, exceptions, programming by contract and testing the interactions of objects and classes.

### 3.1 Concepts in Object-Oriented Programming

The object-oriented approach to building software constructs a solution from objects and their interactions. Objects represent entities in the problem that the software is designed to solve. In this problem space, objects can be characterised by their *state* and *behaviour* (Booch 1994). The state of an object is determined by its static and dynamic properties, and its behaviour by its possible actions and reactions. An object's behaviour may change its state, and at any given time the state of an object is an accumulation of the effects of its behaviours since its creation.

So, objects are packages of data and operations that may act on that data, which we call its *attributes* and *methods*. We call this packaging *encapsulation*. Objects interact by sending a *message* to other objects requesting that they perform one or other of their methods. The sending object is called the *client*, and the *supplier* is the object receiving and acting on the message and producing some response. It is only possible to manipulate an object through its methods.

For an object, each of its methods has a name, a (perhaps empty) set of arguments and possibly returns an object. Its name and the types of its arguments and return value make up the *signature* of a method. The set of signatures of an object is its interface or *type*. The type of an object determines which messages it can receive, that is, only those which match a signature in its type. We say that one type is the *subtype* of another if it contains all the signatures in the other, which we call its *supertype*. Since an object of a particular type can accept the same messages as an object of a supertype, any object could be replaced by another that is of one its subtypes. This substitutability property is known as *polymorphism*.

Types, which can be thought of as existing in the 'design space', are implemented in an object-oriented programming language by *classes*. A class provides definitions for the attributes and methods of the type it implements. In the following, we shall use uppercase names for both classes and the types they represent. It will usually be clear which is meant from the context. By making the distinction between class and type we follow Leavens (1991) rather than Meyer (1988). We find this distinction useful because classes are constrained by the need to represent them in a programming language, whereas types can be represented in a 'design language' that also may not be as formally defined. This representation gap can be a source of defects. We will return to this issue in Section 5.2.

Simple types such as INTEGER, BOOLEAN or CHARACTER can be represented in by classes in some object-oriented programming languages such as Smalltalk, or as a distinct set of elementary types in others, for example C++, Java and Sather. We will assume the latter. Attributes can be simple values such as integers or booleans, or they can be other objects. The class ACCOUNT might include the attributes *balance*, simple value of type integer, and *owners*, itself a list object whose elements are of type CLIENT. The values of an object's attributes make up its internal

state.

Methods are implemented by functions or routines. *Information hiding* is the ability of a class to restrict the visibility of its attributes and methods and their implementation. Objects are created by *instantiating* a class and hence the object is then said to be an *instance* of the class. A class method that instantiates objects of that class we call a *constructor*. Methods that access an object's state without changing it are sometimes called *selectors* and those that do change an object's state are *modifiers*.

A class may *inherit* the attributes and methods of another class, becoming its *subclass* or *child*. The inherited class is referred to as a *superclass* or *parent*. In many object-oriented programming languages, inheritance is used both for subtyping and to reuse part or all of the implementation of the parent. A subclass may *redefine* an inherited method by providing a new implementation. Most languages do not distinguish between these two uses for inheritance, Java (Arnold & Gosling 1996) and Sather (Omohundro & Soutamire 1995) are perhaps the only exceptions.

An *abstract* class does not provide a complete implementation of its type, and so cannot be instantiated, that is objects cannot be created from it. Abstract classes are typically used to define a common interface that will be implemented by several subclasses. Classes that define a complete implementation, and hence can be instantiated, are said to be *concrete*.

A *generic* or *parameterised* class has one or more type parameters. A common use is to define data structure classes independent of the types that may be used in the data structure. For example, a `LIST{ELT}` class has a parameter `ELT` for the element type. Then an object could be declared to be a `LIST{INTEGER}` by supplying `INTEGER` for the type parameter, or a `LIST{POLYGON}` by substituting `POLYGON` instead. Both classes would have similar behaviour and interface except that one would work with objects of type `INTEGER` and the other `POLYGON`. Some object-oriented programming languages allow the type parameter to be *constrained* to be a subtype of some specified type. For example, a list that permits searching its elements could that require its element types have a method to make comparisons like this: `SEARCHABLE_LIST{ELT < COMPARABLE}`.

*Exceptions* are a mechanism used in many programming languages for handling error situations. The general model is that if a method cannot complete its task for some reason, instead of simply returning to its calling method it *raises* an exception. The exception may be *caught* in the calling method, if it has sufficient context to either retry the called method or clean up allocated resources and exit. If the exception is not caught by the immediate caller, it is passed on to the caller's calling method, and so on up the call stack. An uncaught exception usually causes the program to fail.

*Assertions* are boolean valued expressions that, when they evaluate to false, raise an exception, and otherwise have no effect. Assertions are used in a number of programming languages to check the state of a variable or a relationship between variables during the execution of a routine. Some object-oriented programming languages

provide the facility for a class to define an invariant and to define preconditions and postconditions for its methods.

A *precondition* is an assertion that is evaluated when a method is entered. They can express requirements of the input arguments or of the state of the supplier class. For example, before calling the debit method of an ACCOUNT object, it must have a positive balance and the amount to be debited must be positive.

A *postcondition* is an assertion that must be true when the method returns. Postconditions usually constrain the actions of methods or their returned objects. For example a debit message on an ACCOUNT object must reduce the account's balance by the amount withdrawn.

An *invariant* is an assertion that constrains the state of an object. For example, a SAVINGS\_ACCOUNT object must always have a non-negative balance. Normally, a class's invariants are evaluated upon entry to and exit from each of its methods.

These terms and definitions are by no means the only ones, but will suffice here. A common terminology for object-orientation has yet to evolve. Alternate terms and definitions can be found, for example in (Meyer 1988, Wirfs-Brock et al. 1990, Rumbaugh et al. 1991, Booch 1994, Berard 1993a, Henderson-Sellers & Edwards 1994, Palsberg & Schwartzbach 1994, Liskov & Guttag 1986).

## 3.2 Testing Redefined

In Section 2.1, we outlined a formal description of testing. There we saw that, given a program  $P : D \rightarrow R$  and a specification function  $S$  for  $P$ , we infer the correctness of  $P$  by selecting a (usually small) subset  $T \subset D$  and evaluating the boolean expression  $S(d) = P(d)$  for each  $d \in T$ . In this section we adapt this definition to testing classes.

Let  $C$  be a class with methods  $c_1, c_2, \dots, c_n$ . Each method has a signature of the form:

$$c_i(P_{i1}, P_{i2}, \dots, P_{im_i}) : R_i \quad (3.1)$$

where  $R_i$  is the return type of  $c_i$  and the  $P_{ij}$  are the types of its arguments. We allow that  $R_i$  is empty for methods that do not return an object and  $m_i = 0$  for methods with no arguments.

An object of class  $C$  can receive messages consisting of a method  $c_i$  and sequence of objects  $(p_{i1}, p_{i2}, \dots, p_{im_i})$  whose types are  $P_{i1}, P_{i2}, \dots, P_{im_i}$ . The method uses the current state of the object, that is the values of its attributes, and the state of each argument  $p_{ij}$ . The method may also change the object's state. So the behaviour of an object is determined by its state and the message received.

<p><b>Type:</b> SORTED_LIST (Parameter type: ELT)</p> <p>Maintains a list of objects of type ELT in sorted order.</p> <p><b>Signatures:</b></p> <p>size : INTEGER Returns the number of elements in the list.</p> <p>insert (ELT) Adds the argument as an element of the list, preserving the order.</p> <p>delete (ELT) Removes the argument from the list, if it is a member. If not the list is unchanged.</p> <p>member (ELT): BOOLEAN Returns 'true' if the argument occurs in the list, returns false if not.</p> <p>equals (SORTED_LIST) : BOOL Returns 'true' if each element of the argument is equal to the elements in the same position in the list object and vice versa. Returns false otherwise.</p>
---

**Figure 3.1:** A SORTED\_LIST type.

Let  $C'$  be the set of all possible states of objects of the class  $C$ . We will call  $C'$  the *state space* of the class  $C$ . Similarly  $P'_{ij}$  and  $R'_i$  are the state spaces of the argument types  $P_{ij}$  and return type  $R_i$ , respectively. We can write  $c_i$  as a (partial) function on state spaces:

$$c_i : C' \times P'_{i1} \times P'_{i2} \times \dots \times P'_{im_i} \rightarrow R'_i \times C' \quad (3.2)$$

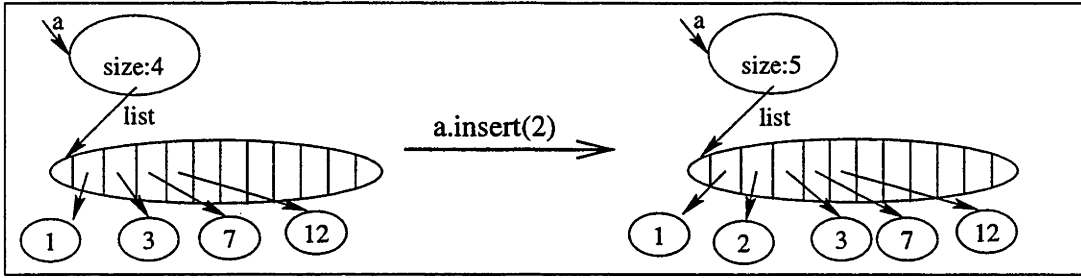
representing the output of the method and its effect on the object's state. This represents the most general form: in both the above expressions, the  $R$  could be removed if  $c_i$  does not return anything or similarly the  $P$ 's are removed if  $c_i$  lacks arguments.

A state of an object  $a$  that is an instance of class  $C$ , can be represented by a graph with nodes indicating objects and directed edges object references. The state space  $C'$  is then a set of these graphs. Messages sent to  $a$  can be represented as transformations from one state graph into another (Schmidt & Zimmermann 1994b, Schmidt & Zimmermann 1994a).

Figure 3.1 shows a description of a SORTED\_LIST type. A class implementing this type appears in Figure 3.4. The state of a SORTED\_LIST object is determined by the number and state of the elements it holds. Figure 3.2 shows two states of an object  $a$  which is an instance of SORTED\_LIST{INT}, and how the message  $a.insert(2)$  transforms the object from one state to another. In the figure, `list` is a reference to an array object containing references to integer objects (see the implementation in Figure 3.4).

It may be the case that  $c_i$  has some side-effect, such as writing to a file, committing a data base transaction, displaying a window in a GUI, or activating a valve in a process control system. All such situations could be specified and therefore should be tested.





**Figure 3.2:** Two states of a `SORTED_LIST{INT}` object, before and after the message `a.insert(2)`.

Let  $S$  represent the set of side-effects in which we are interested (it would all also include a “null side-effect” for the case when there is none). Then expression 3.2 becomes:

$$c_i : C' \times P'_{i1} \times P'_{i2} \times \dots \times P'_{im_i} \rightarrow C' \times R'_i \times S \quad (3.3)$$

Further, a method either returns normally or raises an exception. It may also have changed the state of the object or caused some side-effect before raising the exception. Let  $Exp$  be the set of known exceptions that could be raised, and expression 3.2 becomes:

$$c_i : C' \times P'_{i1} \times P'_{i2} \times \dots \times P'_{im_i} \rightarrow C' \times (R'_i \cup Exp) \times S \quad (3.4)$$

Now to test the method  $c_i$  of  $C$ , we set an object  $a$  of type  $C$  in the required state and select arguments  $b_{i1}, b_{i2}, \dots, b_{im_i}$  for  $c_i$  of the required types and in the required states. Then we evaluate the message  $a.c_i(b_{i1}, b_{i2}, \dots, b_{im_i})$ .

A test case for the method  $c_i$  of class  $C$  examines one aspect of the behaviour of that method. Given that the object  $a$  of type  $C$  (or a subtype of  $C$ ) is in a particular state (written as  $a'$ ), and the states of the arguments to  $c_i$ , the test case checks whether the returned object is correct and the resulting state of  $a$  is correct.

Let  $S_{c_i}$  be a function defining the requirements on the method  $c_i$ . Then, following the lines of Section 2.1, we say  $c_i$  is correct if, for all possible combinations of the states of  $a, b_{i1}, b_{i2}, \dots, b_{im_i}$ , the following is true:

$$S_{c_i}(a', b'_{i1}, b'_{i2}, \dots, b'_{im_i}) = a.c_i(b_{i1}, b_{i2}, \dots, b_{im_i})' \quad (3.5)$$

where  $a.c_i(b_{i1}, b_{i2}, \dots, b_{im_i})'$  is the state of the object returned from the method call when  $a$  is in the state  $a'$  and the arguments are in states  $b'_{i1}, b'_{i2}, \dots, b'_{im_i}$ . Of course the specification function  $S_{c_i}$  must also consider side-effects and exceptions.

In practice, this is complicated by the interaction of state and behaviour. The state of an object at any moment is the result of the messages it has received and acted upon up to that moment. So, before the test case can be run there will be some sequence of

---

methods to set  $a$  in the required state. This will include a constructor and perhaps some modifiers. Further, the only way to find out an object's state is to send it a message. To determine the correct behaviour of the test case, we must evaluate the state of  $a$ , the state of the returned object when there is one, as well as any side-effects or exceptions.

It should be noted that a very similar line of reasoning could be applied to the case where  $c_i$  is simply a function in a procedural programming language (and  $a$  would be just another of its arguments). In that case, we would need to consider the states of the whole system rather than just those of the class  $C$ . Also note that the states of a class show a structure which can be taken advantage of in the integration testing of classes.

### 3.3 Testing Objects

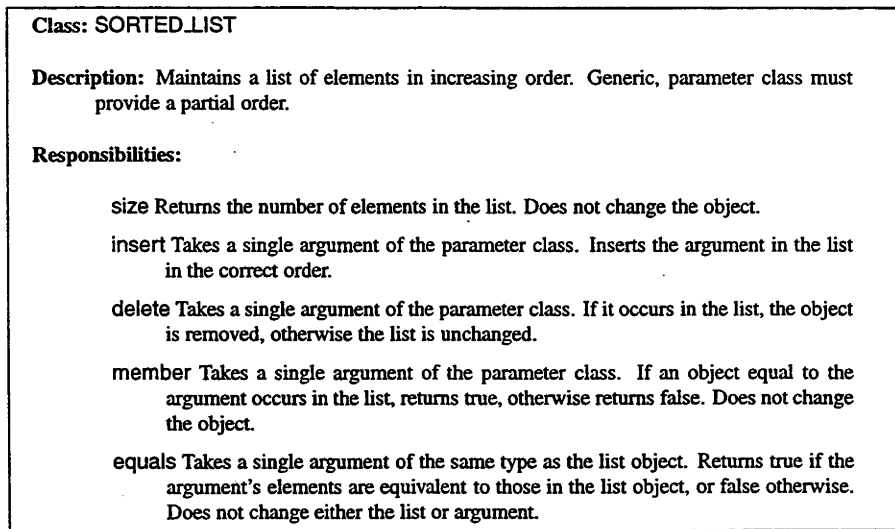
Chao & Smith (1993) note that the organisation of code in object-oriented programs is different from procedural programs in the following sense. In structured programming, procedures are the essential organisational unit while object-oriented programming is organised around classes. The effect of localisation on testing is that instead of organising test plans around procedures and modules as in traditional programs, in object-oriented programs they are organised around classes (Berard 1993a).

In the formal definition of testing presented in the previous section, we have already seen the important role of an object's state with regard to testing that object. Also implicit in the definition is the effect of encapsulation: the encapsulated behaviours of an object must be tested together. The following subsection looks more closely at these aspects of testing of individual objects and classes in isolation.

#### 3.3.1 Encapsulation

A class such as the `SORTED_LIST` class, whose specification appears in Figure 3.3, encapsulates methods for accessing and manipulating a sorted list data structure. The methods `size`, `insert`, `delete`, `member` and `equals` make up the interface for this class. As we have seen, the state of objects of this class is represented by the elements that are in the list, and we will see different behaviour from the methods depending upon the state of the object.

For the tester, encapsulation can mean that it may not be possible to test methods of the class individually because of the level of interaction between them. In our `SORTED_LIST` class, it would not make sense to test one of the methods without using the others. Equivalence partitioning (see Section 2.3) suggests the tests for `delete`, for instance, should include removing a member from the list and removing a non-member from a non-empty and an empty list. However, without breaking the class interface, the only way to create a non-empty `SORTED_LIST` object is to use the `insert` method.



**Figure 3.3:** Specification for a SORTED\_LIST class.

This has led many authors to state that in object-oriented software, the class is the “unit of testing” (Berard 1993*a*, Fiedler 1989, Cheatham & Mellinger 1990, Turner & Robson 1992*b*). By this they mean that rather than testing the methods of a class separately, test cases should be selected to test a class as a whole. As Berard (1993*b*) puts it:

It makes as much sense to individually add methods to a class and test them as it does to individually add statements to a procedure and test them.

This does not mean that we do not test individual methods of the class, only that we do so in the context of the whole class. More precisely, we should take account of the internal state and the ways in which it can affect the methods.

Some authors see classes as integrating a collection of methods, and thus see a need to separately test methods (D’Souza & LeBlanc 1994). It should be clear from the preceding discussion that this is only possible if either the class has no internal state, or there is some means to access and manipulate the class’s internal state directly. If such a test interface is not available, this requires that the tester modify the class either directly, by adding methods to manipulate the state, or indirectly, by subclassing, adding the state manipulator methods to the subclass and testing the subclass. The latter is the approach suggested by Turner & Robson (1993). In either case, the class tested is not the original class, which raises the question “what are we really testing?”. If a failure is uncovered, is it due to the additions made by the tester?

On the other hand, when the methods of a class do not communicate through the class’s internal data, they can be tested and integrated in the traditional way.

### 3.3.2 Information Hiding

In object-oriented programming, information hiding is the means for controlling access to the attributes and methods of a class. Sather and Eiffel use the keyword `private` to remove attributes and methods from the class interface, in C++ this is handled by the `public:`, `private:` and `protected:` parts of the class declaration.

The most common use for information hiding is to deny clients access to details of the class's implementation. In the implementation of the `SORTED_LIST` type in Figure 3.4, the attribute `list` is hidden and can only be accessed indirectly through other methods.

Information hiding affects the testability of classes. In order to determine that the result of a message is correct, we may need to check whether some internal data structure has been updated correctly. If the data structure cannot be accessed directly, the (black-box) behaviour of a class must be tested in a "stimulus-response" way.

In structural testing of the `insert` method in the `SORTED_LIST` implementation, we would ideally like to check the change in `list` directly to see that it was entered in the list in the correct position. In practice, we rely on the other methods: that `size` increased by one, and `member` returns `true`.

Berard (1993a) makes the simple suggestion to first establish a level of confidence in the state reporting methods (`member` and `size` in the example) and then use these to evaluate the other methods. This either requires careful test design or, as in the case of the example, other means of verifying those methods.

### 3.3.3 State

As we have seen above, there is a close link between an object's state and behaviour: the state of an object can affect its response to the messages sent to it, and this response may involve a change of the object's state. In effect, an object's methods communicate through its state. So to test the methods of an object, we must take into account its state as well as the arguments to the method.

Among writers and practitioners of object technology, the word "state" is used in two different ways. In object-oriented programming, "state" refers to the values of the attributes of an object at a particular point in time. In modelling systems with objects, state is an abstraction of this: an object's state qualitatively determines the response of an object to messages. We shall use the term *internal state* for the first sense and *design state* for the latter (Jacobson et al. (1992) calls this *computational state*). The *internal state space* of an object is the set of all possible combinations of values of its attributes (that is the cross-product of the domains of the attributes). For many classes this will be very large.

These two meanings for "state" are commonly confused in the literature on object-oriented programming, design and analysis. For the tester however, we shall see that the distinction is important.

```

class SORTED_LIST{T < $IS_LT{T}} is
  private attr list:ARRAY{T};          -- Array holding list elements.
  readonly attr size:INT;              -- Number of elements in list.
  private const initial_size:INT:=5;   -- Starting size of the list array.

  create:SAME post ~void(result) and result.size = 0 is
    -- A new sorted list.
    res ::= new; res.list := #ARRAY{T}(initial_size);
    return res
  end; -- create

  private elt_eq(e1,e2:T):BOOL is
    -- True if 'e1' is equal to 'e2' for the semantics of this list.
    -- If T descends from $IS_EQ{T}, use its 'is_eq', otherwise use
    -- object equality.
    typecase e1
      when $IS_EQ{T} then return e1.is_eq(e2)
      else return SYS::ob_eq(e1,e2) end
  end; -- elt_eq

  private position(e:T): INT is
    -- Return the index of 'e', or '-1' if not found.
    i:INT;
    loop i := size.times!; while!(list[i] < e); end;
    if (i < size) then return i else return -1 end
  end; -- is_mem

  insert(e:T) post member(e) >= 0 is
    -- Put the element 'e'.
    if size >= list.size then          -- Expand list if full.
      list := list.resize(2 * list.size) -- This copies over the old elts.
    end;
    -- Search to find where it goes (index held in 'i').
    i:INT;
    loop i := size.times!; while!(list[i] < e); end;
    -- Copy bigger elements on one place.
    loop list.set!(i+1, list.elt!(i)) end;
    list[i] := e; size := size + 1;
  end; -- insert

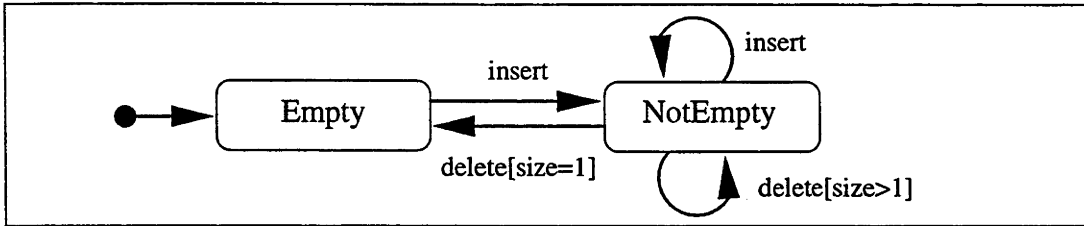
  delete(i:INT) pre i >= 0 and i < size is
    -- Remove 'i'th element from the list.
    loop list.set!(i, list.elt!(i+1)) end;
    -- Just copy later elts back one place.
    size := size -1; list[size] := void;
  end; -- delete

  member(e:T): BOOL is
    -- Return true iff 'e' is an element of the list.
    return position(e) /= -1;
  end; -- is_mem

  is_eq(other: SAME): BOOL is
    -- Return 'true' iff 'other' has the same elements as 'self'.
    if size /= other.size then return false end;
    loop if ~elt_eq(list.elt!, other.list.elt!) then return false end; end;
    return true
  end; -- is_eq
end; -- class SORTED_LIST{T}

```

Figure 3.4: A Sather implementation of the SORTED\_LIST class.



**Figure 3.5:** A state-transition diagram showing design states for `SORTED_LIST`.

McGregor & Dyer (1993) regard the design states of an object as a partitioning of the internal states according to some classification rule or rules that group internal states which share some observable behavioural attribute. Design states can be modelled by statecharts (Harel 1987) or state-transition diagrams (Booch 1994, Rumbaugh et al. 1991, Shlaer & Mellor 1988). An example appears in Figure 3.5, and further examples will be given in Chapters 5 and 6, where we use state-transition diagrams to generate test suites.

Our `SORTED_LIST` object has design states that are also determined by the number and state of its elements. One state corresponds to the empty list, which is also the *initial state* or the state of the object when it is created. We would expect a `SORTED_LIST` object to respond quite differently to a `delete` message if was in the empty state than otherwise.

The internal state of a `SORTED_LIST` object, as implemented in Figure 3.4, is determined by the values of `list`, `size` and `initial_size`. This is effectively the number and value of the elements of the list. The design state is illustrated by the state diagram in Figure 3.5. The diagram shows two states, one representing the list with no elements and in the other the list has at least one element. The initial state is indicated by an arrow with no source. State transitions are caused by the methods `insert` and `delete`. The other methods (`member`, `size` and `equals`) are not shown since they do not lead to state transitions and would clutter the diagram.

A more formal way to relate internal state to design states is to define the design states by predicates on the attributes. The empty and not empty states for the `SORTED_LIST` class can be defined by the two predicates `size = 0` and `size > 0` respectively. A `SORTED_LIST` object is in the *Empty* state if it satisfies the first, or the *NotEmpty* state if it satisfies the latter predicate. Substates can be created by refining a predicate and concurrent states by independent predicates (having no variable in common). Notice that since the class invariant is a predicate that captures the valid states for an object of this class, it is the logical disjunction of these design state predicates. This idea is developed further in chapter 5

If the object has a finite state machine specification or state-transition diagram, or the tester can deduce such a specification, then functional testing can follow estab-

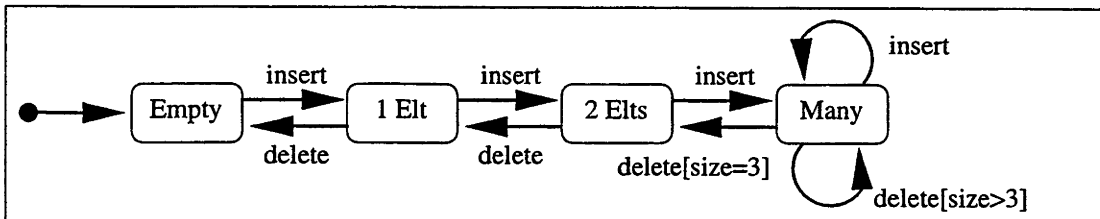


Figure 3.6: An alternative state-transition diagram for SORTED\_LIST.

lished techniques for testing finite state machines described in Section 2.4 (McGregor 1994, Marick 1994). For example, the set of test cases derived from a transition tree for the state-transition diagram in Figure 3.5 would be

event sequence: create;	final state: <i>Empty</i>
event sequence: create, insert;	final state: <i>Not empty</i>
event sequence: create, insert, insert, delete[size > 1];	final state: <i>Not empty</i>
event sequence: create, insert, delete[size = 1];	final state: <i>Empty</i>

While these test requirements cover the behaviour described by the state-transition diagram, they will not make an adequate test suite, even when we add test requirements for testing the other methods in each state. Binder (1996) suggests that a transition tree can be combined with boundary-value analysis, depending on the class's kind of state-based behaviour. Structural testing techniques from Section 2.3 such as data-flow testing would be required to further test internal state.

Note that there are many possible representations of the design states of a class. The more refined state-transition diagram for SORTED\_LIST shown in Figure 3.6 shows a more detailed description of the class's behaviour and would lead to a more detailed set of test requirements. Refining state-transition diagrams can be a useful technique, especially when the test cases can be automatically generated from the diagrams, as we shall see in Chapter 6.

### 3.4 Inheritance

Inheritance is probably the most discussed feature of object-oriented programming languages and there are many different views. One outcome is that every object-oriented programming language provides its own unique mechanisms and semantics for inheritance. It is in the context of testing, where we compare requirements, design model and implementation, that we can bring these differences into a new, sharp focus.

In testing classes and objects, one key question is to what extent does a subclass need to be tested given that its parent has been sufficiently tested. As we shall see

---

there are several aspects to be addressed in answering this question. We start this section with an examination of different views of inheritance and the considerations for testing, then discuss issues raised by multiple inheritance and briefly look at testing abstract classes. To finish we shall return to state-based testing, but this time the state-based testing subclasses.

### 3.4.1 Inheritance is a Many Splendour'd Thing

Inheritance implements the “is a” relationship between objects in object-oriented design. During design we might say a `OVERDRAFT_ACCOUNT` “is a” `ACCOUNT`, or a `BOUNDED_STACK` “is a” `STACK`. But there is an essential difference between these two examples. `OVERDRAFT_ACCOUNT` has all the behaviour of `ACCOUNT` (as well as additional functions), but a `BOUNDED_STACK` does not provide the same behaviour as `STACK`. This is because a `BOUNDED_STACK` has some maximum size and once that size is reached, after a maximum number of push messages, additional pushes would not change it. A `STACK`, on the other hand, does not have this limit and would continue to grow with every push. The relationship between the `ACCOUNT`s is an example of *classification* and the other is an example of *specialisation* (Henderson-Sellers & Edwards 1994). In this case, the `BOUNDED_STACK` specialisation allows a more efficient implementation to be used.

The distinction is important for the tester who tries to validate the intent of the designer in the code. Classification implies substitutability while specialisation does not. It should be completely safe to use an `OVERDRAFT_ACCOUNT` where ever an `ACCOUNT` object is required. But this is not the case for `BOUNDED_STACK`: when it replaces a `STACK` object, its use needs to be validated to ensure that behavioural differences do not cause failures in that context. Specialisations require retesting each time they are used.

More formally in a specialisation, method preconditions can be strengthened and postconditions weakened, while in a classification preconditions can only be weakened and postconditions strengthened (Frick et al. 1994). The precondition for push is strengthened to `size < max_size` in `BOUNDED_STACK`, whereas it has no precondition in `STACK` (or more accurately is has the precondition `true`).

When testing a specialisation we should include test cases that violate its preconditions or its parent’s postconditions in the context where it is used. A test suite for any class that uses `STACK` should try to do more than `max_size` pushes, followed by as many pops. The intention is to find faults that might arise when `BOUNDED_STACK` is substituted for `STACK`. Further information on pre- and postconditions and testing can be found in Section 3.7.

Unfortunately, inheritance in object-oriented programming adds further complications. The central issue for testing is the distinction we drew in Section 3.1 between two types of inheritance: implementation inheritance or subclassing, and specification



```

type $ACCOUNT is
    balance: MONEY;
    open;
    close;
    credit(amount: MONEY);
    debit(amount: MONEY);
end; -- class $ACCOUNT
-----

class ACCOUNT < $ACCOUNT is
    readonly attr balance: MONEY;
    private attr isOpen: BOOL;

    create: SAME is
        res: SAME := new;
        res.isOpen := false;
        res.balance := #MONEY(0);
        return res;
    end;

    open is ... end;
    close is ... end;

    credit(amount: MONEY) is
        if isOpen and amount >= 0 then
            balance := balance + amount;
        end;
    end;

    debit(amount: MONEY) is
        if isOpen and amount >= 0 and balance >= 0 then
            balance := balance - amount;
        end;
    end;
end; -- class ACCOUNT
-----

class OVERDRAFT_ACCOUNT < $ACCOUNT is
    include ACCOUNT debit->;

    readonly attr limit: MONEY;

    debit(amount: MONEY) is
        if isOpen and amount >= 0 and balance - amount >= limit then
            balance := balance - amount;
        end;
    end;

    set_limit(amount: MONEY) is
        if isOpen then
            limit := amount;
        end;
    end;
end; -- class OVERDRAFT_ACCOUNT

```

**Figure 3.7:** Part of a Sather implementation of the ACCOUNT - OVERDRAFT\_ACCOUNT inheritance hierarchy.

---

or interface inheritance which we called subtyping.

Specification inheritance is a relationship between interfaces or types. A subtype inherits all the attributes and methods of its parent, and may add more. We say that the child *conforms* to its parent's interface. This means that every signature that appears in a type will also be found in its subtypes and hence a subtype can receive at least the same messages as any of its parents (Leavens 1991).

Specification inheritance is the basis for polymorphism: a child can be substituted for its parent because for any message that the parent could receive the child could also receive since it has that signature. Note that this means that the child should not remove any of its inherited methods, or equivalently, hide them by making them private, even though this is permitted in a number of object-oriented programming languages. It is nearly always trivial to find a test case that causes a failure when an inherited method is removed or hidden.

Implementation inheritance, on the other hand, is a form of code reuse. A subclass includes the implementation of all the methods of its parent classes. It can rename or remove methods, usually in order to redefine them, as well as define new ones. Since implementation inheritance occurs behind the wall of a class's interface it is not constrained to conform to the parent class. An alternative technique for reusing a method implementation from another class is *delegation* (Lieberman 1986).

In almost all object-oriented programming languages, specification and implementation inheritance are combined through a single mechanism. In C++ implementation inheritance can be achieved through `private` inheritance since it hides the inheritance relation, but `public` inheritance combines both specification, because it means the child substituted for the parent, and implementation inheritance, since the parent's methods and attributes are included in the child. In Java, `implements` is pure specification inheritance, but `extends` also combines specification and implementation inheritance.

Sather is perhaps the only object-oriented programming language that distinguishes between specification and implementation inheritance. An example of the two forms of inheritance in Sather is shown in Figure 3.7. The abstract class `$ACCOUNT` defines an interface and both `ACCOUNT` and `OVERDRAFT_ACCOUNT` are subtypes, as indicated by the clause `< $ACCOUNT` in their declarations. Implementation inheritance occurs through the statement

```
include ACCOUNT;
```

in the body of `OVERDRAFT_ACCOUNT`. Semantically, this is as if the implementation of `ACCOUNT` were copied in at that point, except that `debit` is left out. Inheritance in Sather is described in the Appendix, Section B.4.1.

Specification inheritance implies substitutability. Wherever an object of a particular type is required, it is possible to substitute an object of any subtype because the

interface of the subtype is a superset of the interface of required type. An immediate consequence of substitutability is that test cases for objects of one type are applicable to all its subtypes. Marick (1994, Chap. 22) sees a hierarchy of test requirements that parallels each (specification) inheritance hierarchy, in which the test requirements for each class includes the test requirements of its parents.

We can draw a similar distinction in test design for classes: functional test requirements are derived from the interface, and structural test requirements from its implementation. The functional test requirements for a class should include those of its supertype and the structural test requirements include those of any class whose implementation is reused. To these are added the functional test requirements from additional methods for the interface of the class under test, and structural test requirements from new and replaced method implementations.

This, however, is not enough. In the implementation of a class, methods may interact with each other and the class's attributes. A change in one part of the implementation, such as when an inherited method is redefined, potentially creates new interactions and therefore new test requirements. It cannot be assumed that tested methods remain correct when inherited. They must be retested in their new context, the subclass. Harrold et al. (1992) have developed an algorithm that applies this to testing subclasses in C++.

Perry & Kaiser (1990) provide a more detailed answer to this question when they re-evaluated Weyuker's axioms of test suite adequacy (Weyuker 1986, Weyuker 1988) in the context of testing object-oriented software. They found that two of Weyuker's axioms have particular relevance to the situation we have just described.

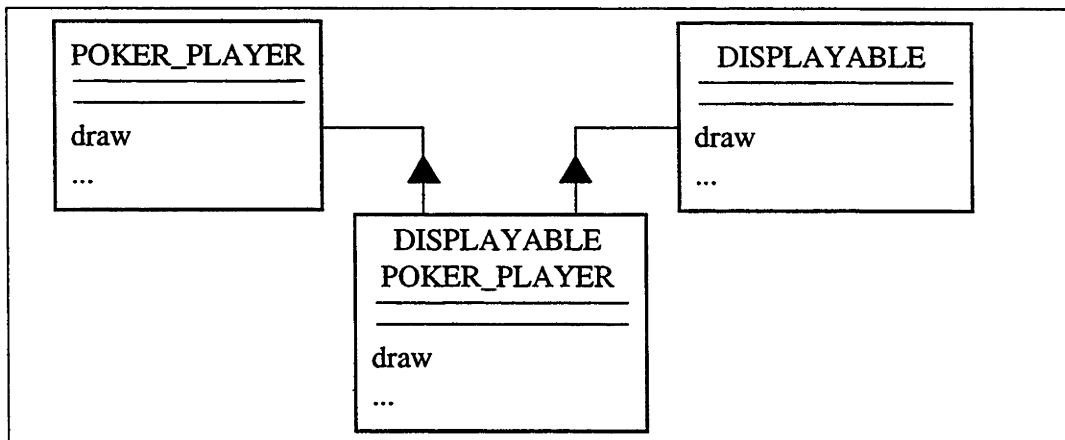
Weyuker's axiom of *antiextensionality* states that if two programs compute the same function, an adequate test suite for one is not necessarily adequate for the other. If, in a child class, an inherited method is replaced by a locally defined one, antiextensionality warns that it is not necessarily adequately tested by the test suite of the inherited method.

Weyuker's axiom of *antidecomposition* states that given an adequate test suite for a program, the subset applicable to a component of that program is not necessarily an adequate test suite for that component. We have already seen that redefined methods in a child class may need new tests. A consequence of antidecomposition is that this is also true of inherited methods, since the child class may give them a different context.

### 3.4.2 Multiple Inheritance

If we allow a type to have more than one parent (multiple inheritance), a subtype inherits the attributes and methods of all its parents. A problem arises when two parents have the same signature, since the subtype can not have two signatures that are the same.

Consider a scenario in which a software developer is asked to add a graphical

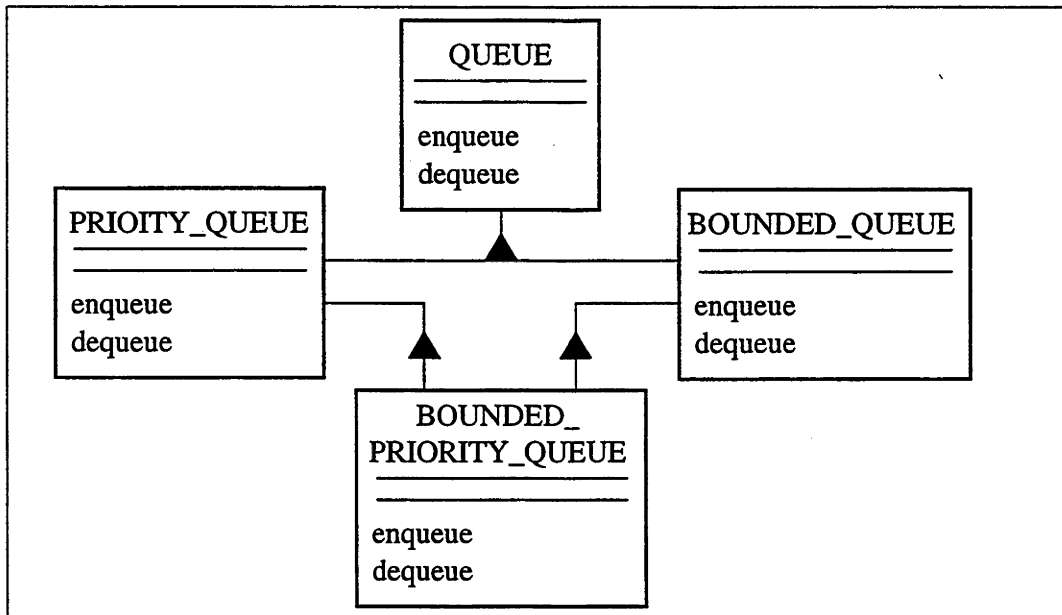


**Figure 3.8:** A signature clash resulting from multiple inheritance.

user interface (GUI) to a simulation that plays the card game “Poker”. One way to create a visual representation of a simulated poker player might be to combine the `POKER_PLAYER` class with the services provided by the `DISPLAYABLE` class from a GUI framework. Figure 3.8 shows a new class, `DISPLAYABLE_POKER_PLAYER`, that combines the required services by inheriting them from both classes. Unfortunately, through lack of prescience on the part of designers of the GUI framework or the Poker simulation, both these classes use the same method name, `draw`, for entirely different behaviours. When a class inherits the same signature from two different parents, we call this a *name clash*.

More commonly, name clashes occur from *convergent inheritance* (sometimes called repeated inheritance). This occurs when a class has multiple parents which themselves have a common parent. In effect, the class repeatedly inherits the interface of this common “grandparent” through each of its parents. For example in Figure 3.9, the class `QUEUE` could be specialised to the classes `PRIORITY_QUEUE` and `BOUNDED_QUEUE`. The new class `BOUNDED_PRIORITY_QUEUE` would inherit the methods `enqueue` and `dequeue` from both subclasses of `QUEUE`.

A name clash is usually not a problem when it results from convergent inheritance, because both inherited methods will have the same behaviour, having themselves inherited it from a common parent. The behaviour of `enqueue` and `dequeue` in `BOUNDED_PRIORITY_QUEUE` is determined in `QUEUE` since both `PRIORITY_QUEUE` and `BOUNDED_QUEUE` objects can be substituted where a `QUEUE` is required. But in the example in Figure 3.8 it cannot be resolved. One way to avoid the name clash is to rename the method `draw` inherited from `POKER_PLAYER` to say `draw_a_card`, but in that case, when the poker simulation sends a `draw` message to each of its `POKER_PLAYER` objects, any whose actual type is `DISPLAYABLE_POKER_PLAYER` will instead redraw itself on the screen. On the



**Figure 3.9:** A signature clash resulting from convergent inheritance.

other hand, the alternative, renaming the draw inherited from `DISPLAYABLE`, will cause unexpected behaviour when a collection of `DISPLAYABLE` objects that include a `DISPLAYABLE_POKER_PLAYER` object is drawn on the display.

Object-oriented programming languages have a variety of mechanisms for resolving name clashes. Some languages, such as Smalltalk and Objective C, avoid this problem altogether by only allowing single inheritance: a class may have no more than one parent so a name clash cannot occur. Many object-oriented programming languages that permit multiple inheritance have some facility for renaming inherited attributes and methods. When two parents have a common signature, changing the name of either or both of the inherited attributes or methods prevents a name clash.

As Berard (1993*b*) points out, different languages have different mechanisms for resolving these conflicts and this will determine test requirements. Sather has renaming associated with implementation inheritance (the `include` clause), but signature conflicts need be resolved by changing the parents. C++ does not have a means of renaming attributes and methods, instead the required method is selected at each call by explicitly indicating the parent from which it was inherited. In some object-oriented programming languages (including earlier releases of Sather (Omohundro et al. 1993)) a name conflict is resolved by taking the last declaration, so later declarations hide earlier ones with the same signature. In this case changing the inheritance order is a small change that can have a significant effect on test design (Perry & Kaiser 1990). Resolving signature conflicts is more properly a modelling and design issue. See, for

---

example, Henderson-Sellers & Edwards (1994, pages 62–65) for a discussion of issues that arise in modelling with multiple inheritance.

Subclasses need to be retested to show their behaviour is not at odds with each parent. So the test requirements for a class should include those of all its parents. With convergent inheritance this is usually straight forward: the tests for `BOUNDED_PRIORITY_QUEUE` include those for both `PRIORITY_QUEUE` and `BOUNDED_QUEUE`, and both these include the tests for `QUEUE`. For `DISPLAYABLE_POKER_PLAYER`, substitutability for one or other of the parents will be broken and the combining tests from `POKER_PLAYER` and `DISPLAYABLE` can be used to discover if this causes a failure.

### 3.4.3 Abstract classes

To test a concrete class, one or more objects are instantiated from that class and the test cases applied to those objects. But abstract classes cannot be instantiated, no objects can be created from them. Obviously abstract classes cannot be tested in this way. So what is sufficient testing for abstract classes?

Abstract classes often define a common interface abstraction for several classes. They may hide various trade-offs that need to be made in the implementation, such as memory usage versus computation speed.

In a typical application there will already be concrete subclasses of an abstract class. So creating new subclasses specifically for testing is unnecessary. Test requirements are derived from the abstract class and then added to those of each concrete subclass. If there are no concrete subclasses then the abstract class is not used, which may also indicate an error in the design or the implementation. In short, abstract classes are tested by testing their (concrete) subclasses.

### 3.4.4 Inheritance and State

In Section 3.3.3 we made the distinction between design state and internal state. Under implementation inheritance, a subclass acquires the internal state space of its parent. This may then be modified in the subclass by adding, renaming, replacing or removing attributes. Since a class's internal state is hidden and not part of its interface, it is unaffected by specification inheritance.

Design states, however, are related to the object's visible behaviour. In object-oriented design, we require that an object conform to the behaviour of its parent(s). Thus a finite state machine specification or state-transition diagram for one type will also apply to subtypes. So, as we have seen previously, state-based test cases derived from a parent's state-transition diagram should be applied to all child classes.

Under specification inheritance, a state-transition diagram for a subclass may only vary from its parent's state-transition diagram in a limited number of ways. States

may be modified, but can only be added as substates of one of the parent's states or as concurrent states. Concurrent states often result from new independent attributes such as occurs in multiple inheritance. Transitions can be added between existing and new states or changed. Neither states nor transitions may be deleted (Coleman et al. 1992, McGregor & Dyer 1993). Guard conditions on transition can only be weakened. These are necessary, but not sufficient conditions to ensure substitutability, allowing an object of any subtype to act in the role of its parent type. When these rules are followed, tests based on a transition tree of the parent's state-transition diagram should not find any new faults when applied to subtypes.

Figure 3.10 shows an example of the inheritance of design states among members of the `QUEUE` inheritance hierarchy in Figure 3.9. Figure 3.10(a) shows a state-transition diagram for `QUEUE` with states *Empty* and *NotEmpty*. In Figure 3.10(b), a `BOUNDED_QUEUE` adds its two new states, *Full* and *NotFull* as substates of *NotEmpty*. The transition

*NotEmpty*, `dequeue[size>1]`, *NotEmpty*

has been replaced by the transitions

*Full*, `dequeue`, *NotFull*

*NotFull*, `dequeue[size>1]`, *NotFull*

That `BOUNDED_QUEUE` is a specialisation rather than a proper subtype is apparent when we look at what happens to the `enqueue` transition on *NotEmpty*: it is not accepted in substate *Full*.

The state-transition diagram for a `PRIORITY_QUEUE`, in Figure 3.10(c), also introduces two substates to the *NotEmpty* state. Here they represent the situations where either a new element has the highest priority and can be just pushed onto the queue (the *NewTop* state) or the queue needs to be restructured (the *NewShape* state). In this case both the `enqueue` and `dequeue` messages have modified behaviour that does conform to that of `QUEUE`.

Figure 3.10(d) shows concurrent states (or "AND states" in Harel 1987) introduced by multiple inheritance. A `BOUNDED_PRIORITY_QUEUE` can be in one of the pairs of states *NotFull* and *NewTop*; *NotFull* and *NewShape*; *Full* and *NewTop*; or *NotFull* and *NewShape*. `BOUNDED_PRIORITY_QUEUE` is a proper subtype of both `PRIORITY_QUEUE` and `BOUNDED_QUEUE`.

### 3.5 Genericity

Genericity, like inheritance, is a form of abstraction. A generic or parameterised type abstracts parametrically identical signatures of a group of behaviourally related

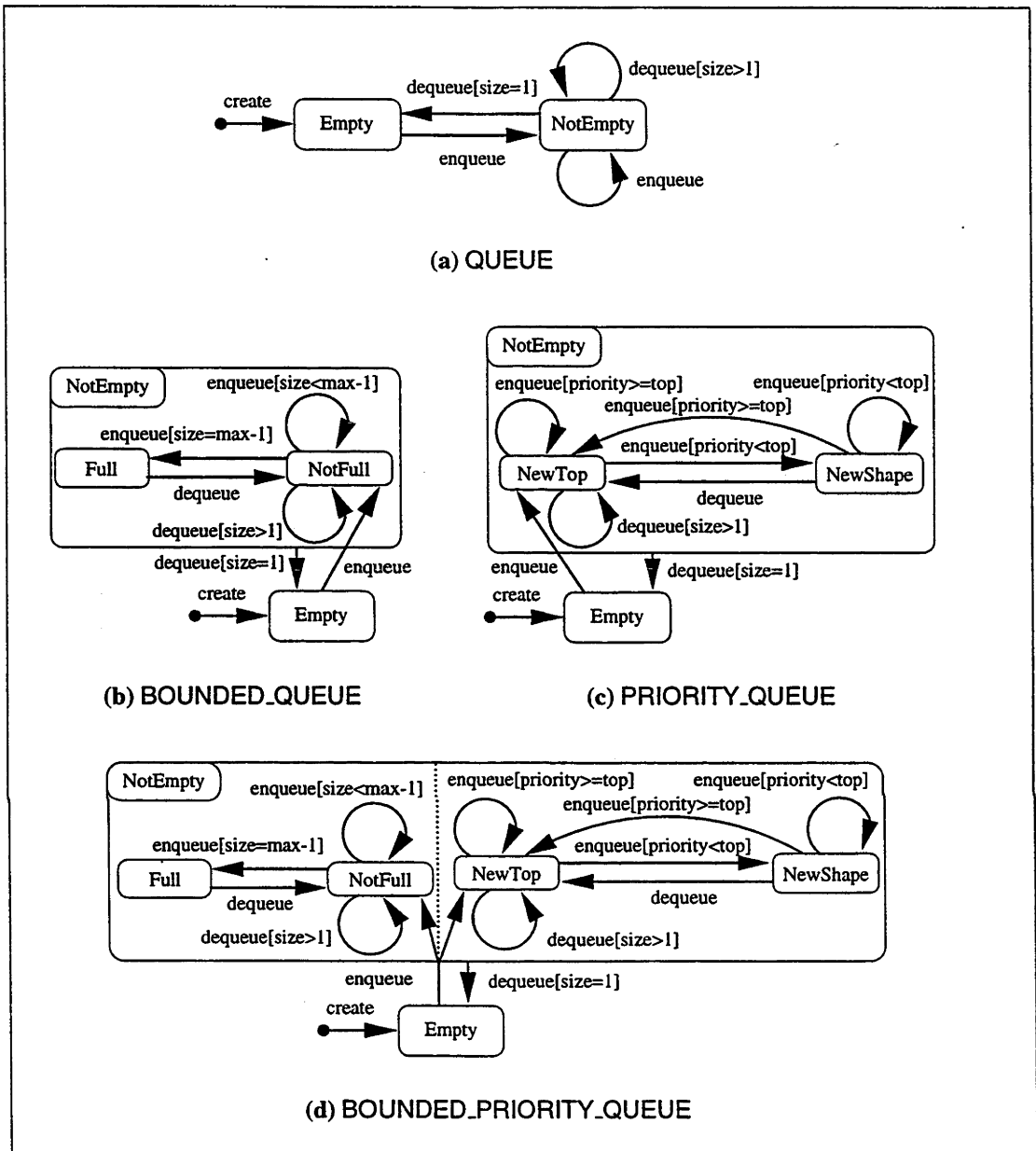


Figure 3.10: Inheriting design states in the QUEUE hierarchy.

classes. For example, all of the classes `LIST{INTEGER}`, `LIST{STRING}` and `LIST{TEMPERATURE_SENSOR}` have similar list behaviour, such as inserting and deleting elements, they differ only in the type of list element. The common list behaviour can be extracted by parametrising the list element type (`INTEGER`, `STRING` or `TEMPERATURE_SENSOR`). A generic class is then a template or pattern for making classes with similar behaviour. Concrete classes are made by substituting for



the type parameter. Generic classes are often used for collection types with the element types as parameters, such as the `SORTED_LIST{ELT}` example from Section 3.3.

Generic classes, like abstract classes, can not be tested directly. Classes must be chosen to substitute for the parameters, and test cases applied to objects of the resulting class.

Of particular concern with generic classes are possible interactions between a generic class and its parameter class. Sometimes these are hidden in implicit assumptions about methods or behaviour the parameter class should provide. For example, many generic classes that model data structures provide some means of searching for an element, such as `member` in the `SORTED_LIST` example above, and this requires that element objects are compared. If the generic class assumes that element types all define the same equivalence relation, it will fail with those that don't.

Ideally all possible instantiations of parameters should be tested. In an application it may be possible to test each of the parameters that is actually used in the application. For generic classes that are intended for reuse, it is not in general known with what combinations of parameters they will be used or how they may interact. One guideline is to instantiate the generic class with at least a small class, such as `INT`, and a more complex class (Smillie & Strooper 1995), but where the generic class makes assumptions about its parameters this is clearly not enough.

### 3.6 Exceptions and Exception Handling

Exceptions may have many sources and correspondingly are made to serve many roles. In most programming languages that have exception handling, programmers may define exceptions to suit their own purposes, but they may also originate in the hardware or operating system, for example interrupts, file access failures and divide-by-zero errors. They may also be built into the language or its runtime system, such as a failure to allocate memory or accessing attributes through a variable that does not reference any object (a "void" access). In languages that have them, violation of an assertion, method precondition, postcondition or class invariant causes an exception to be raised. In these examples we have seen exceptions used as a reporting mechanism for system interrupts, resource acquisition failures, and programming errors.

While these are all examples of unexpected, exceptional conditions, they can mean different things in different situations. This can make it difficult to set a policy on the handling of exceptions, and without a policy it can be very difficult to decide on an appropriate approach to testing exceptions and exception handling. Further, exceptions caused by program errors, such as void accesses and contract violations, should not be caught during testing and debugging as catching them hides the errors, but in operation it may be required that no exception is uncaught.

Good examples of exception recovery are rare in the literature (Meyer 1992a). Part of the problem is defining an appropriate strategy for dealing with what may

---

be an internal problem in a supplier. For example a postcondition failure in Eiffel or Sather triggers an exception which, if caught, must be caught in a client. So in effect, the client is asked to deal with the supplier's failure to satisfy its contract. It is possible that the client will have some alternate strategy for performing the task normally performed by the supplier but in general this is unlikely to be the case.

With regard to testing, the advice most often given is that test cases should raise all possible exceptions (Liskov & Guttag 1986, Marick 1994, Berard 1993a). However, in many object-oriented languages, the exceptions that can be raised by a class are buried in the implementation of the class's methods. Worse still, exceptions may also be raised in a called method of some further supplier class. If the supplier class is from a class library or under the control of a different development group, there may be no information other than the class interface. If not caught, these exceptions will also be suffered by the class's clients. Exception handling code in a caller creates dependencies between the client and the implementation of the supplier. Subclasses of the supplier must not add new exceptions for fear of breaking client code.

In a few object-oriented programming languages, Java and CLU are examples, exceptions are specified in the method interface. Failure to handle *at the method call* all the possible exceptions raised by the method is considered a syntax error and rejected by the compiler (Liskov & Guttag 1986).

There are two aspects to validating the use of exceptions in software: raising and catching. In the first case testing seeks to ensure that the right exceptions are raised in the right place in the right circumstances. The second is to check that exception handling code responds appropriately to the correct exception. To some extent this can be checked statically, for example Eiffel's "exception correctness" (Meyer 1992b) tries to ensure that the handling of exceptions does not violate the method postcondition or the class invariant.

Ideally, a complete class design should describe the exceptions raised and caught by each method, although some of this information could be summarised in an exception handling policy (testers should be so lucky). But few object-oriented design methods specify exceptions in the class interface (Berard 1993a, Chap. 8) (the OODLE notation (Shlaer & Mellor 1992) is one exception).

The above requirements amount to raising all exceptions in each context they affect. This may be difficult to achieve if some exceptions are caused by the operating system, hardware or some other means beyond programmer control. An important example of this is exceptions due to failed memory allocation. It is more important to check that caught exceptions are handled correctly (Marick 1994, p. 226).

### 3.7 Assertions and Software Contracts

A central concern of verification and validation is whether supplier classes have been used correctly. The correct syntax can be determined by compilers, but the correct

semantics of an interaction between objects, in the end, needs to be tested.

Software contracts are a means of specifying the semantics of correct usage. The “programming by contract” paradigm was developed in the context of object-oriented programming by Meyer (1988) and Wirfs-Brock et al. (1990), from much earlier work on formal semantics and program proving. Preconditions and postconditions are cast as a “contract” between client and supplier: if the client abides by the precondition, providing arguments as required then the supplier is bound to perform such actions as necessary to meet the postcondition.

Preconditions and postconditions specify the external behaviour of objects, so they are properly part of their interface. Since subtypes conform to the behaviour of their parents (this is the substitutability property described above) they should also fulfil their parent’s contracts. So preconditions and postconditions preserve semantics under inheritance (subtyping).

Invariants can be used in two quite distinct ways. Some invariants specify constraints on the behaviour of objects of a type, while others ensure the implementation remains valid. The latter is called a *representation invariant* (Liskov & Guttag 1986). An example of the first kind is that elements of a doubly-linked list are strongly connected: `next = void` or `next.prev = self`. A representation invariant describes a consistency relationship between attributes, for example, in the implementation of `SORTED_LIST` in Figure 3.4, `size` should be the index of the first void entry in `list`. Marick (1994, Chap. 18) describes a detailed strategy for testing consistency relationships.

Preconditions, postconditions and invariants can be checked with assertion statements in languages that have them, but few object-oriented programming languages, Eiffel and Sather among them, provide specific support for programming by contract. Meyer proposed the assertion sub-language for software *verification* in Eiffel (Meyer 1994). A more practical approach is ADL (Viswanadha & Sankar 1996), where invariants, preconditions and postconditions are used to generate test cases.

Adding assertions for checking preconditions, postconditions and invariants is a useful technique for testing if a method of a class is used incorrectly (in the case of a precondition violation) or if it is implemented incorrectly (in the case that a postcondition or invariant is violated) (Marick 1994, page 348). As Meyer (1992a) points out: “any runtime violation of an assertion is [...] always the manifestation of a software bug.” So assertions are an aid in exposing errors (or propagation as we called it in Section 2.1). In any case, preconditions, postconditions and invariants are an important source of test requirements.

Given their important role in ensuring correctness, it is vital to validate all assertions. It is not uncommon for functions to appear in assertions, since if functions are not used, this limits the programmer to the expressibility of propositional logic. Of particular concern for testing is that functions in assertions “should be of unimpeachable quality, avoiding change to the current state and any operation that could result in

---

abnormal situations” (Meyer 1992a). Such functions could, of course, be redefined in subclasses and need to be retested as described previously in Section 3.4.

Invariants can constrain the relations between objects in a cluster or subsystem (Duke 1994). Unless it can be applied to a single class in the cluster, there is currently no way to capture such invariants in code. In that case they can only be validated by testing and are then a further source of test requirements (Marick 1994, Chap. 21).

## 3.8 Integrating Classes and Subsystems

Objects rarely exist in isolation. Useful object-oriented programs consist of many interacting objects. In well designed object-oriented software, objects are grouped into *clusters* or *subsystems*, which in turn are combined into larger subsystems until they form an application. Many design patterns (Gamma et al. 1995) are such clusters. Objects typically interact more closely with other objects in the same subsystem than those outside it. Subsystems may have one or more interfaces and objects outside the subsystem interact with those inside through one of the interfaces.

Given that objects are accessed only through their interfaces, if their individual classes are correct, why, it might be asked, do we need integration testing of subsystems? Part of the answer is that class interfaces only define the syntax of interaction, not how they should work together. Groups of classes that are designed to work together need to be tested together (Marick 1994, Chap. 21) and there can be errors in the interfaces between them (Harrold & Soffa 1991).

Perry & Kaiser (1990) apply another of Weyuker’s axioms of test set adequacy to this question. *Anticomposition* states that given an adequate test suite for a program, and the outputs of applying that test suite to the program forms an adequate test suite for a second program, then the original test suite is not necessarily an adequate test suite for the composition of the two programs (Weyuker 1986, Weyuker 1988). Because of encapsulation, intuition may suggest that it is sufficient to test each class in isolation. However anticomposition warns that classes must be tested in the context in which they are used. It is not enough to test subclasses according to their parent’s test requirements and test each use of the parent; the subclasses should also be tested in those contexts where it is substituted for its parent(s).

The rest of this section discusses specific forms of interaction between objects and examines the implications for testing those interactions.

### 3.8.1 Associations and Aggregations

When one object sends a message to another, an *association* exists between them. Association is the most common form of relationship between objects. A special kind of association is *aggregation*, which refers to the case where one object is a component of another. For example, MEETING\_ROOM and RECEPTION objects might be

parts of an `OFFICE_BUILDING` object, and `CHAIR`, `TABLE` and `WHITE_BOARD` objects components of the `MEETING_ROOM` object. The objects in this example form a “parts-of” hierarchy, characteristic of aggregation.

The client-supplier relationship is a dependency between objects. A client object requires the existence and correct operation of the supplier object for its own correct operation. This implies a similar relationship, and dependency between classes. The objects of a supplier class provide services for objects of a client class, and the client class depends on the supplier class.

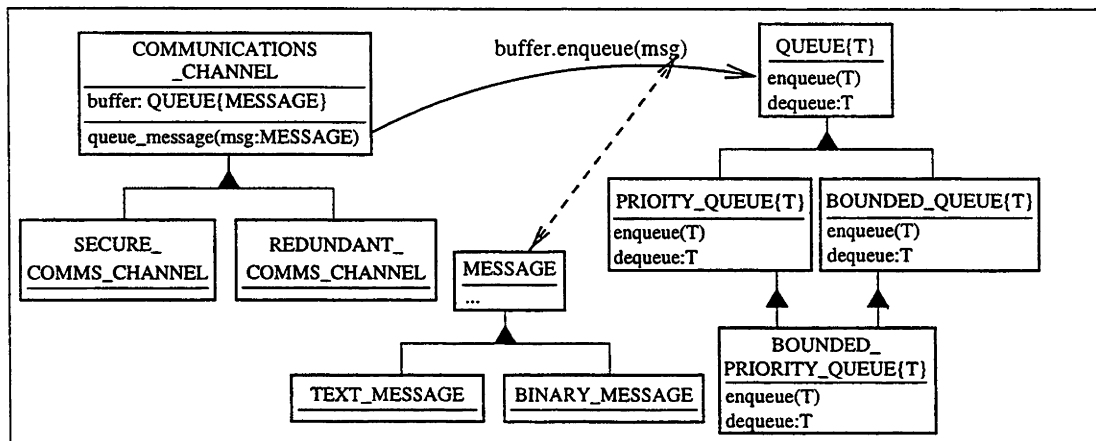
The most natural strategy to apply to testing groups of interacting classes is the “bottom-up” approach, borrowed from integration testing. This requires the construction of a graph of the dependencies between classes in the cluster or subsystem. Good design indicates that this graph will have no cycles, and redesign to remove cyclic dependencies is straight forward (Lakos 1992). Classes with no dependencies are tested first. Then each remaining class can be tested when those classes that it depends upon have been tested.

However, circular dependencies are not uncommon in low-level design, for example the “model” and “view” classes in the model-view-controller design pattern. Co-dependant classes appear in some of the design patterns (Gamma et al. 1995) as recursive data structures such as Composite, Decorator and Interpreter, and also where an aspect of the behaviour of an object is captured in a separate object such as Iterator, Mediator and Observer. In these cases, the same argument used in Section 3.3.1 can be applied and the “unit of testing” becomes a small cluster of tightly bound, cyclically dependant classes. It makes no sense to test an iterator, for instance, without the container it iterates over, and at least some of the container class’s behaviour will require the iterator. Such cyclically intra-dependant clusters can then be treated as a single node in a class dependency graph and integrated as described.

This technique assumes all the classes to be tested are complete. This is not always the case. A “top-down” or “middle-out” approach may be more appropriate if the required “stub” classes and methods can be added at minimal cost or are part of the development anyway. As the stub classes are replaced the classes which relied upon them will have to be re-tested. These approaches still make use of a class dependency graph, which will need to be updated as class interactions are redesigned during development. An approach to integrating testing into this sort of iterative and incremental software development is proposed in chapter 4.

Some classes assume that their methods will only be applied in certain sequences. For example, an `ACCOUNT` object will not accept debit and credit messages before it has received an `open` or after a `close`. These built-in assumptions can be explicitly tested using the method sequence specification (MtSS) technique of Kirani & Tsai (1994).

In fact these correct message sequences are equivalent to valid sequences of transitions of design states discussed in Section 3.3.3. As we saw there, the receiver of a



**Figure 3.11:** The classes involved testing a polymorphic method call enqueue.

message may respond differently according to its state. So not only is it necessary to test every different message but also to test each message with the different states of the receiver.

### 3.8.2 Inheritance and Dynamic-Binding

In Section 3.4, we have seen that specification inheritance implies that objects of a subtype are substitutable for objects of the parent type. In other words, if an object can receive a message, so can objects of each of its subtypes. Dynamic binding is the mechanism that defers until run-time the decision as to which actual object will receive a message, and thus which subclass's method is invoked. To ensure the substitutability of subtypes, they should be tested in the same situations as the parent is used.

Testing all the combinations of subtypes that could be involved in each interaction can lead to a combinatorial explosion in the number of test cases. Take, for example, the situation in Figure 3.11. Here we see that during the execution of its `queue_message` method, a `COMMUNICATIONS_CHANNEL` object sends a message `enqueue(msg)` to its `buffer` attribute. Both the subtypes `SECURE_COMMS_CHANNEL` and `REDUNDANT_COMMS_CHANNEL` inherit `queue_message` and `buffer`, but could redefine or add other attributes and methods that interact with them, and so they should be retested as was explained in Section 3.4. The attribute `buffer` has type `QUEUE{MESSAGE}` and so could also be any of the three subtypes we have seen previously, and the actual method argument `msg` could be a `TEXT_MESSAGE` or a `BINARY_MESSAGE` object instead of a `MESSAGE` object. Complete testing of all combinations of subtypes requires  $3 \times 4 \times 3 = 36$  versions of the tests for `queue_message`. In fact, we need to also consider the design states of `COMMUNICATIONS_CHANNEL`, `QUEUE` and `MESSAGE`, according to Sec-

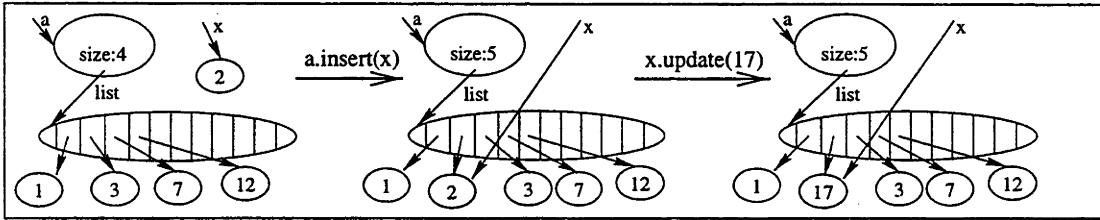


Figure 3.12: Changing the internal state of a class through an alias.

tion 3.3.3. So, the number of tests required is the product of the numbers of subtypes and states involved in the message.

McDaniel & McGregor (1994) suggest a strategy that selects from all these combinations, a set of test cases that includes at least one of each subtype and state. This is a similar idea to configuration testing (see, for example, Kaner et al. 1993, Chapter 8). This technique ensures that each known subclass and state is exercised at least once, which is a fairly weak notion of coverage. They report that it produces a test suite that is substantially smaller than exhaustively testing all combinations but still has quite good error detecting ability.

### 3.8.3 Aliases

An alias occurs in a program whenever two different objects possess the same references to another object. An alias makes it possible for a value in the referenced object to be changed through one reference, with unexpected consequences for the other referencing object. Figure 3.12 shows how this might occur, and the potential consequences, for the `SORTED_LIST` list class from Figure 3.4. When the value referenced by `x` is inserted on the list, `x`'s reference to it remains resulting in an alias. If this value is changed through `x`'s reference, the list may no longer be sorted. This would violate the class invariant of `SORTED_LIST` without executing any of its methods! Testing `SORTED_LIST` on its own will not find this kind of error, it only appears in its interaction with other classes.

It needs to be remembered that aliasing is useful and even unavoidable in object-oriented programs. Most object-oriented programmers have had the experience of writing a class that performs some manipulation with objects of the same type as itself, such as an `append` method in a `STRING` class. The first time an object of this class is given itself as an argument, as in `s.append(s)`, it fails.

Sometimes the aliasing comes from the problem domain. Usually this results from many-to-many relationships, such as multiple transactions in a data base system, or joint bank accounts in which two or more customers can operate the same account.

Like exceptions, aliasing is not unique to object-oriented programming, but also

---

like exceptions, they have a special prominence. There are essentially two reasons for this. One is that object-oriented programming makes heavy use of references: objects are accessed through pointers, so there are many more opportunities for aliasing to arise. The other is that objects have state that persists between accesses. For many reasons, an object may make its attributes accessible to other objects, which leaves it susceptible to an unexpected change of its internal state.

Several techniques for limiting aliasing in object-oriented programs have been discussed in the literature (Hogg et al. 1992). One is the use of so-called “copy” or value semantics: whenever an object is to be returned by a function, a copy is made first and that is returned instead. In many object-oriented languages, basic types like CHAR and INT have value semantics. If this were the way SORTED\_LIST operated, in Figure 3.12 *x*’s reference would not point into the list. However this can be expensive if the objects are large, and is not always the most suitable behaviour: any other references to the SORTED\_LIST object would also be broken. “Islands” are a more sophisticated method of isolating groups of classes, so that aliasing can only occur within an “island” (Hogg 1991). In some situations, correct behaviour in the presence of aliases can be described by a predicate on the classes in a subsystem or cluster (Duke 1994). For example, it is perfectly acceptable for other objects to access elements in a SORTED\_LIST as long as the elements are not changed. However, this too requires special language support.

Checking programs for aliasing is very difficult. Detecting the possibility of aliasing (“may-alias”) is undecidable in most programming languages, and determining whether two variables must be aliased at some point in the program (“must-alias”) is not even recursively enumerable (Ramalingam 1994, Landi 1992).

For the same reason, it is very difficult to design a test strategy for detecting faults due to aliasing. In reviewing a class design for test requirements, each attribute should be checked to see if it can be aliased and what could be the effect of changing it. In particular, class invariants that constrain attributes that could be aliased should be checked. Test requirements for methods with two arguments of the same type should include the case that the same object is the actual argument for both. In particular, for methods with an argument of the same type as the class, the object receiving the message (“self”) could be passed as the actual argument.

### 3.9 Testing and Reuse

Reuse is widely promoted as the mechanism whereby object technology will drastically improve the efficiency of developing software. Object-oriented software development creates components which can be reused in developing other applications. In this view, “everything is reusable”: not only class implementations but designs, architectures, plans and of course, test requirements and test cases. To be reusable test cases



and test requirements must be traceable to the thing they validate, so they should be archived with their associated product.

We expect software components archived in a reuse repository to be of a higher level of quality than other software. When test cases for those components are available we have a means of assessing their quality. But test requirements should also be archived with the reusable components they test. As Marick (1994) points out, when we use one of these components we will need to test that it is used correctly. The test requirements for exposing a usage error will be the same for all users of the component and properly belongs with it in the repository.

Classes can be reused in two quite different ways: as suppliers or by creating a subclass and inheriting attributes and methods. The test requirements for the two cases will be different, as we have seen in see Sections 3.4 and 3.8.

The requirements for testing frameworks and class libraries are somewhat different than for applications. Classes used in a specific application need to work correctly in that application, whereas those in frameworks or libraries are intended to be reused and must work in yet unwritten applications. Of particular concern here is specialisation inheritance, discussed in Section 3.4. A subclass that is a specialisation can be substituted for its parent provided the strengthened method preconditions are not violated. While a specific application may be able to guarantee this, it can be a potential source for errors in the use of a class library. In an ideal world, developers of class libraries and frameworks would provide their customers with test requirements for the correct use of their software (Marick 1992a), and this would include test requirements that make their specialised subclasses fail substitutability.

### 3.10 Other Object Testing Issues

*Reflection* is the ability to manipulate, as data, some aspect of the program during its own execution (Paepcke 1993, Kiczales et al. 1993). This feature has been used to build test drivers that can detect the interface of known classes and automatically create and execute test cases for them (Rettig 1991).

In object-oriented design, “patterns” describe, at an abstract level, recurring patterns of interaction between classes. Design patterns have been documented so that they can be reused in other object-oriented designs (Gamma et al. 1995). It seems reasonable that such recurring patterns should also have recurring test requirements, which could be described at the same level of abstraction. Current techniques for describing design patterns do not specifically record these test requirements.

The patterns technique has also been applied to testing. “Pattern languages” have been proposed for specific object-oriented programming languages (Firesmith 1996), component testing (McGregor & Kare 1996) and system testing (DeLano & Rising 1996). It remains to be seen whether this is a useful technique.

## 3.11 Summary

A broad survey of the issues concerning the testing of object-oriented programs was presented. While object-oriented programming doesn't invalidate traditional testing techniques, those techniques do need to be adapted to testing classes and their interactions. In particular, the testing of object-oriented software needs to address object state and its interaction in inheritance hierarchies.

Object-oriented techniques provide the means to design and build larger and more complex systems. However, the underlying complexity of object-oriented systems is reflected in their testing. Discovering techniques for reducing the complexity of testing remains an opportunity for further research.

---

# Testing and Object-Oriented Software Development

---

The previous chapter examined testing issues in object-oriented software. This chapter looks at the process issues involving testing and its place in object-oriented software development. We develop a framework for incorporating software testing into the kind of incremental and iterative process that is common in object-oriented software development. We then illustrate the framework by applying it to the popular Booch method (Booch 1994).

Our current model of the software testing process has been developed in the context of traditional models of software development. So, before we examine testing in the object-oriented software development process, we need to review those traditional process models and how object-oriented development differs.

In the next section we review three well-known models of the software development process and, in Section 4.2, the part played by testing. We look at object-oriented software development processes in Section 4.3. Section 4.4 describes the proposed framework for incorporating testing into an object-oriented software development process and then applies it to the Booch process. We finish with a brief look at testing in a few other object-oriented software development methods.

This chapter is derived from a paper given at the International Conference on Testing Computer Software, Washington D.C., June 1996 (Bosman 1996).

## 4.1 The Software Development Process

Before we begin, it will help to clarify a few terms that are essential to this chapter. A software development *process model* or *process* specifies a set of tasks or activities that must be carried out in order to manufacture a software article, their entry and exit criteria, their inputs and outputs, and interrelationships between them. A software development *method* (or *methodology*) prescribes how one or perhaps several of the activities should be carried out and the type of deliverables to be produced.

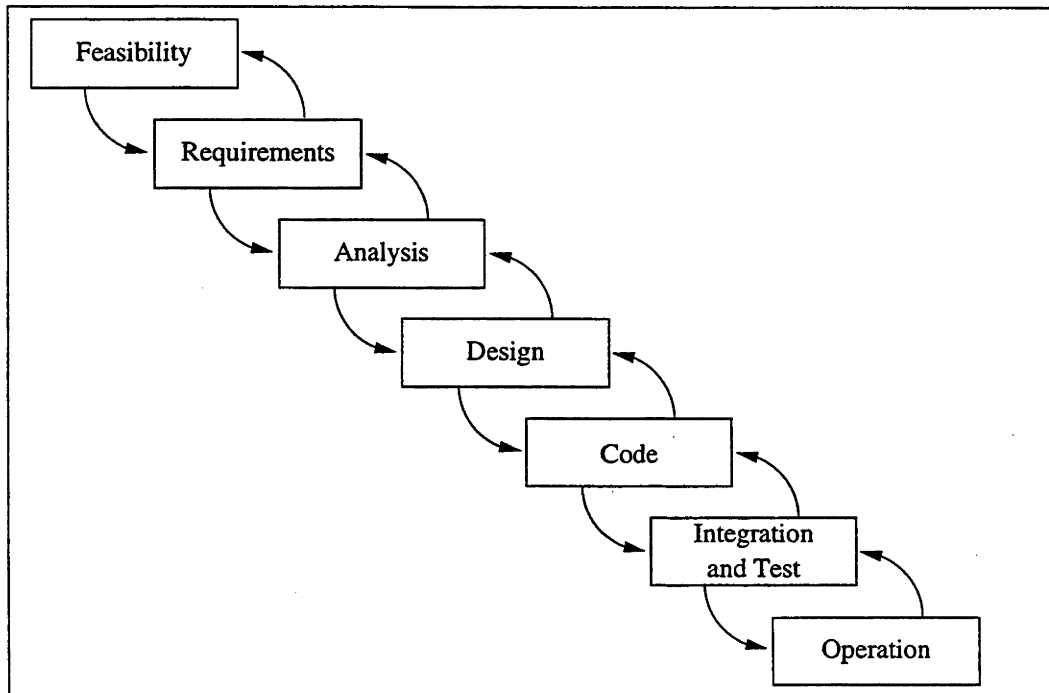


Figure 4.1: The waterfall process model

### 4.1.1 Waterfall Process Model

The most widely recognised process for software development is the “waterfall” process model, one form of which is illustrated in Figure 4.1. This model presents software development as a sequence of phases or stages. The first three stages, namely project feasibility, software requirements and analysis, address the clients, clarifying their requirements. A solution is developed in the design stage. Programmers construct the software to implement the solution in the implementation phase. After integration and validation the software is released to the clients. At the completion of development the software product enters the operation or maintenance phase, in which new requirements are accommodated and bugs fixed.

The way this model is most commonly read, requires that one stage should be completed before the next is begun, hence the name “waterfall,” indicating that returning to a previously completed process is like pushing water back uphill. The waterfall model is suited to the management of the process, since there are well defined criteria for entry to and exit from each stage. It assumes that the problem is sufficiently well understood beforehand so that all requirements can be specified before commencing analysis, all analysis completed before commencing design and all design completed before coding starts. In practice, faults introduced in one stage may not become visible until later stages are entered, thus it becomes necessary to return to earlier stages to

---

solve the introduced fault. Hence, as indicated in Figure 4.1, it is permissible to iterate between successive stages (Royce 1970).

Actually, it is often the case that larger iterations are required. For example integration testing may uncover a problem that should properly be fixed in the design, or later stages of design and implementation may uncover missing requirements. In fact, typically over 30% of errors occur in requirements gathering, and errors in requirements are also the most significant technical risk factor in software development projects. In a strictly enforced waterfall process, the fault is fixed in the phase in which it is found. If the design cannot be changed this can result in an inferior solution and potentially, a dissatisfied client. On the other hand, returning to previously completed tasks can have an impact on all the dependant tasks. This makes the process more difficult to manage, potentially resulting in delayed delivery, increased defects and a dissatisfied client.

In reality there is often a great deal of overlap between stages (Berard 1993a, Chap. 5). It is possible for analysis to commence on one aspect of a proposed system for which requirements exist, while work on requirements for other parts continues. Further, it is not uncommon for requirements to be specified at different levels of detail, with the intention that the higher level ones will be refined “when we get to that part”. The same considerations can be applied to the analysis and design activities and the design and coding activities. Testing is an activity which properly extends across almost the whole process—as we shall see in Section 4.2.

### 4.1.2 Evolutionary Process Models

There are many alternative models for software development and the field is subject to lively debate, particularly in the context of alternative implementation techniques, such as “fourth generation” languages, logic programming, expert systems, applicative programming and, of course object-oriented programming. For our purposes it is more important to recognise the subprocesses or tasks that make up the software development process. These subprocesses are more or less common to all models (McDermid & Rook 1991), with the models differing rather in the ordering of tasks and their entry and exit criteria. A number of different process models are described by McDermid & Rook (1991) and Berard (1993a, Chap. 4). The two most influential alternative process models are *evolutionary delivery* (Gilb 1988) and the *spiral model* (Boehm 1988). For our understanding of object-oriented software development process models, it will be useful to outline them here.

Project milestones in the waterfall process occur at the end of each stage, and as a result the users do not see the software until the project nears completion. Gilb (1988) suggests that it is unrealistic to wait until acceptance testing at the end of development to involve the customer. So instead, he proposed an evolutionary process that delivers the software to the users in small increments of functionality, as early in the project

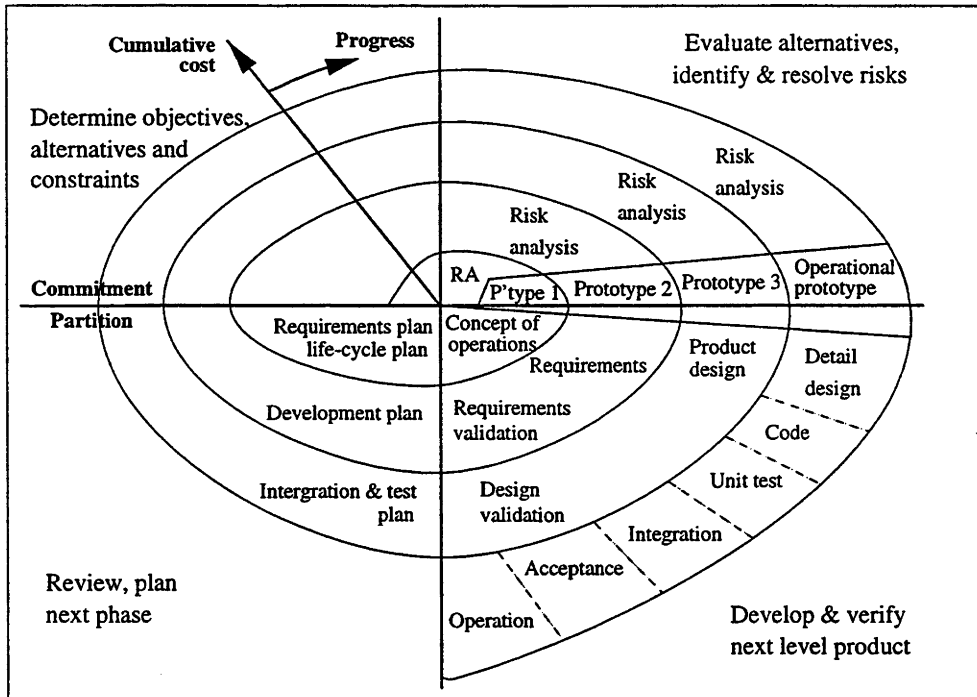


Figure 4.2: The spiral process model

as possible. From a process view, this requires many small frequent iterations through the whole set of development activities, once for each delivered increment.

Evolutionary development is much better at handling changes to requirements and early delivery gives the customers (and therefore project managers) confidence that the project is “on the right track”. It also gives management flexibility in dealing with schedule overrun because there is the additional option of reducing the functionality of the final product (Kaner et al. 1993).

In the spiral model (see Figure 4.2), software development proceeds along an outward spiral curve, iterating through the tasks for each quadrant and building incrementally more detailed prototypes. Progress to the next cycle is based on risk assessment. Each new iteration progresses through the tasks in increasing detail, with consequently increasing cost. Boehm claims that his spiral model extends most other process models.

Different software development process models may be more or less applicable in particular situations. A waterfall process is suited to a domain where the problems are well understood or where a similar system has already been built. Where user requirements are unclear or user-computer interaction is an important facet, an evolutionary delivery process may be more appropriate. The choice of development process is an important early project management decision.

---

The concepts of prototyping and iterating through all the development tasks are central to many views of the object-oriented software development process, as we shall see in Section 4.3. But first we must examine in more detail the part played by testing in the software development process, which is the subject of the next section.

## 4.2 Testing and the Software Development Process

In the waterfall model of software development, testing is generally carried out in the reverse order to software construction. This is represented in the “V” model of software verification and validation from the STARTS Guide (DTI 1987). Figure 4.3 illustrates one form of this model (McDermid & Rook 1991). In this diagram, processes are represented by boxes and the products of those processes by ellipses. Notice that verification tasks (inspections, walk-through and reviews) tend to appear in the right side of this model and testing in the left. The horizontal dotted lines are intended to indicate that the product on the right is expected to meet the specification on the left.

This model suffers from the same problems as the waterfall model. In fact the need to return to earlier phases is more apparent because the V model makes the dependencies between them visible. But as before our purpose here is to identify the components of the testing process and their dependencies with the other processes and products of software development.

At the lowest level, *unit testing*, the smallest testable components of the software are tested. These units of test will have been identified during design. In procedural programming the units of testing are typically procedures, functions and modules. In object-oriented programming they will typically be classes, as we have seen in Section 3.3.1. The techniques described in the previous two chapters apply in the main to unit testing.

At the next level, modules or classes are combined according to the design and then *integration tested* to detect incorrect interactions between the components. Integration testing focusses on interfaces and thus uses mainly black-box testing techniques. The method of integration has a major effect on how integration testing is performed, as we have seen in Section 2.5.

Incremental integration is largely a matter of how the dependencies between components are handled. For the purposes of integration testing, we say that one module *depends* upon another if it uses the services of that module, that is, it calls a function or procedure or references some data defined in that module. The same can be said of classes but in this case dependency may occur through inheritance as well as association since the services may be provided by a parent class or a supplier class (see section 3.8 for a discussion of the integration testing of classes). A *module dependency graph* or *class dependency graph* illustrates this relationship within subsystems or across the whole system (see, for example, Hetzel 1988). Its nodes are modules or classes and a directed edge indicates a dependency relationship between modules or

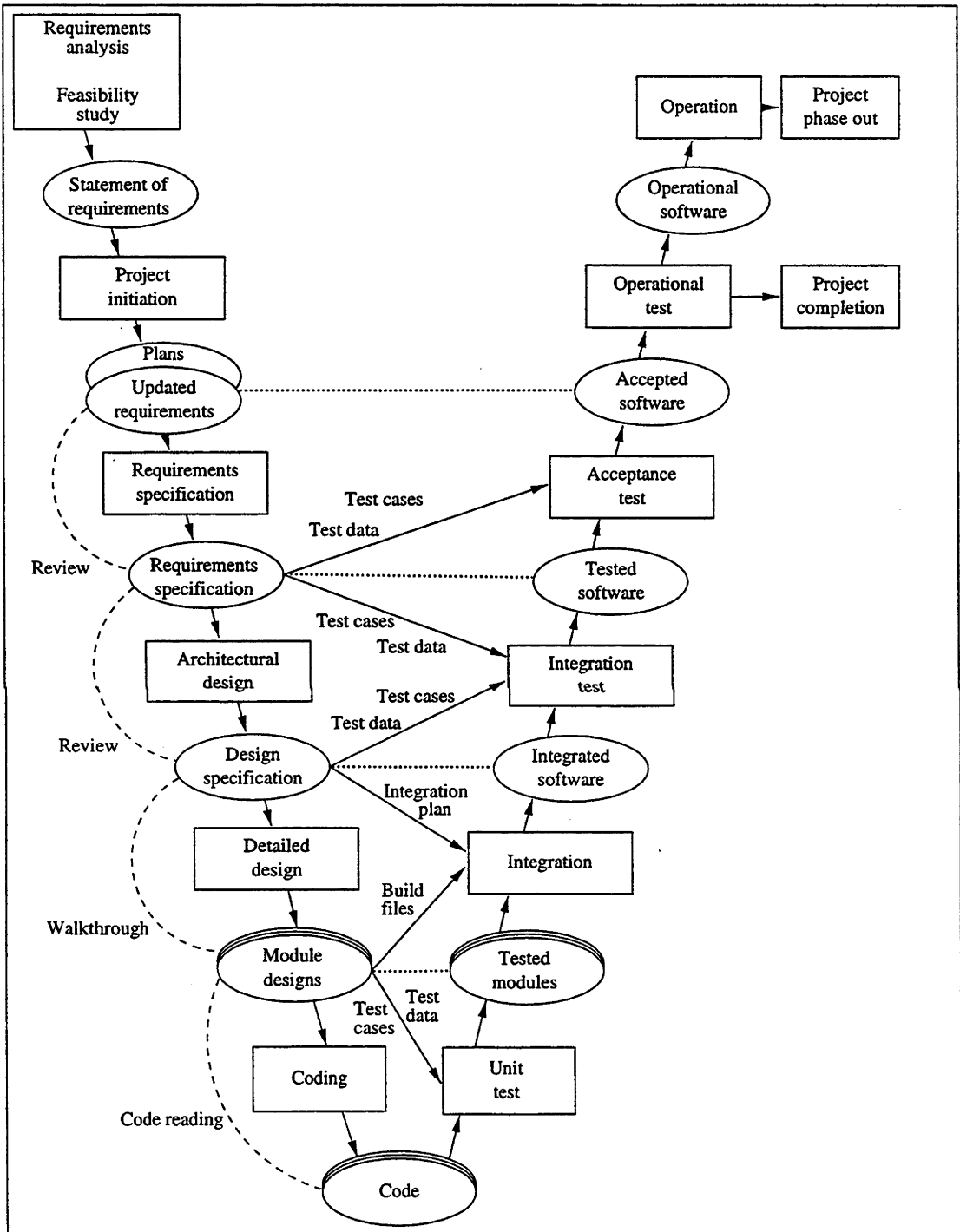


Figure 4.3: The “V” model of software development (after McDermid & Rook 1991)



---

classes. A dependency graph is typically a product of the architectural design. Good design will generally avoid large cycles in the dependency graph resulting in a tree-like structure. One common architecture is a sequence of layers in which the top level is closest to the application domain and at the bottom closest to the hardware and operating system. In this case modules in one layer can only depend upon those in the layer immediately below. Another common architecture is a hierarchy of subsystems where modules are grouped into subsystems which are grouped into larger subsystems and so forth, and interdependencies are restricted to occur only within a subsystem.

*System testing*, in which the system as a whole is tested, occurs when all the subsystems are integrated. This may include other forms of test such as performance testing, stress testing, usability testing, security testing, and reliability testing.

*Acceptance testing* is carried out by or on behalf of the customer to decide whether the software meets the customer's original requirements, as modified and agreed during development, and *operational testing* deals with the customer's initial experience in operating the software.

An important test activity not shown in Figure 4.3 is *Regression testing*. This involves re-executing an existing test suite to check that changes to the software don't introduce new faults. Regression testing does not appear in the figure since it is associated with iterations, which are not shown in the "V" model. During the maintenance phase, regression testing ensures that bug fixes and new features do not adversely affect the correctness and existing functions of the software. As we shall see, regression testing is also important in an iterative software development process. Test sets from earlier cycles are re-executed to check that work done in the current cycle does not change existing correct functionality.

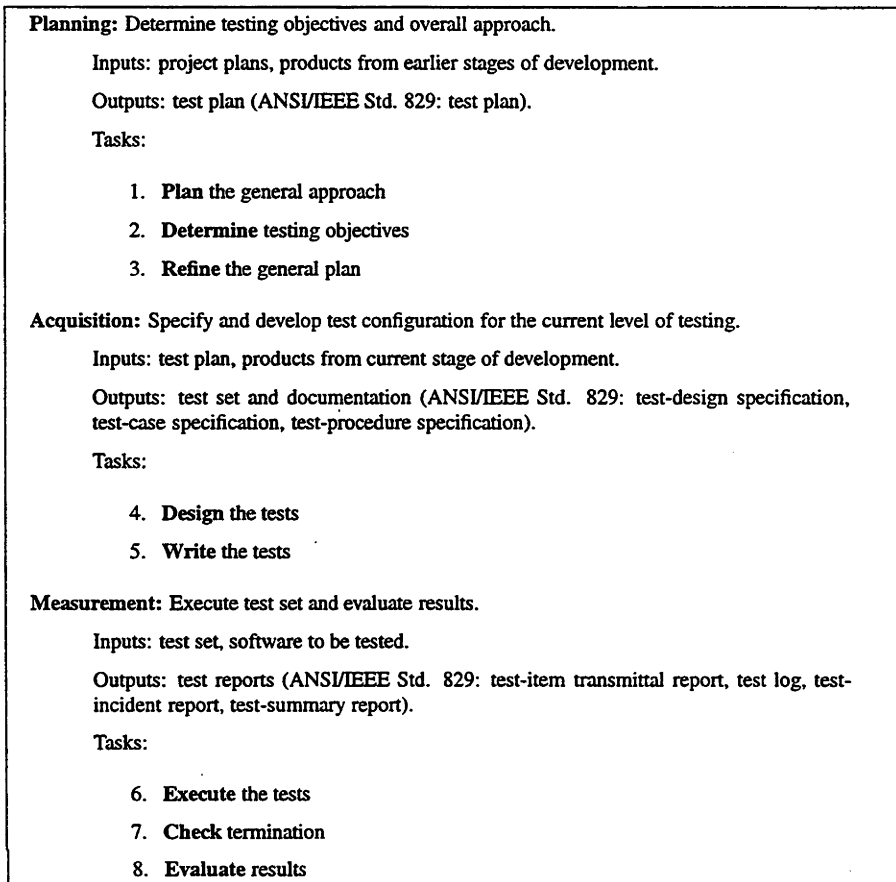
### 4.2.1 A Process Model for Software Testing

In the last section we identified four main test activities: acceptance, system, integration and unit testing. Each of these activities can be broken down into the following phases: planning, acquisition and measurement (Hetzel 1988). These phases are described in Figure 4.4. In the Figure 4.4, the test phases have also been linked to the relevant ANSI/IEEE standard 829 test documents (IEEE 1983).

Tasks in the planning phase determine such things as the level of testing required, parts of the software that will or will not be tested and test techniques to be used. Test planning must of course be done in the context of the overall project plan and the quality assurance plan.

In the acquisition phase, tasks extract test requirements from products of the current stage of development, for example, requirements specification documents for acceptance testing, design documents for integration testing, and source code and detail designs for unit testing. Then test data is designed and test cases implemented.

Tasks in the measurement phase run the test cases to establish that the software

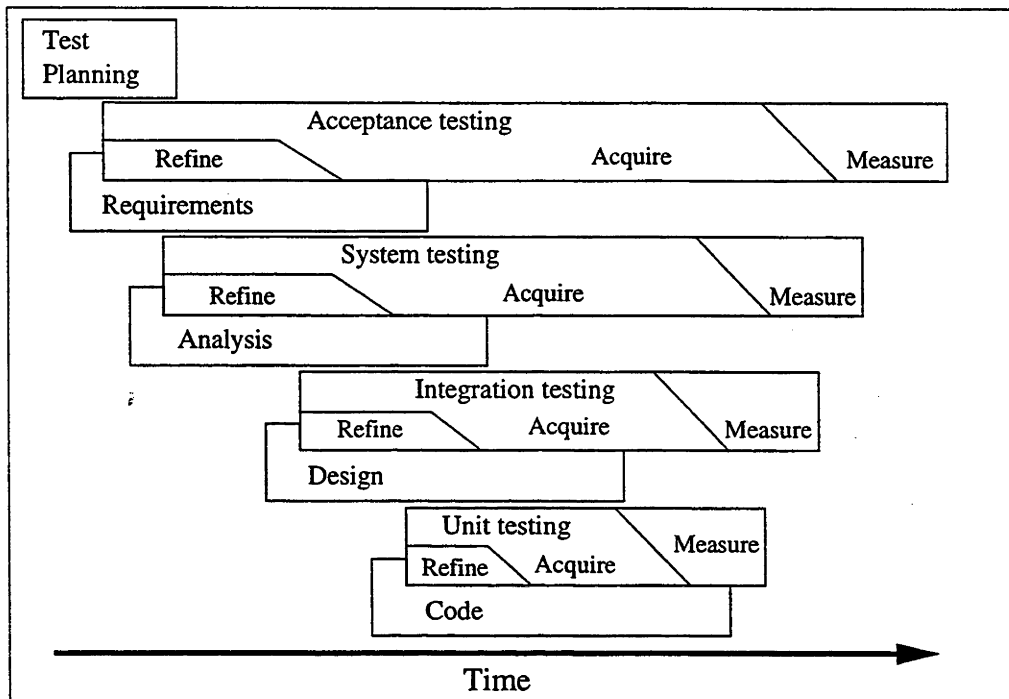


**Figure 4.4:** The major activities in each testing process (after Hetzel 1988)

achieves the level of quality required by the test cases, and report the results and any defects found.

When we recognise the role of planning and acquisition in the test process, we must come to the conclusion that testing and software construction are properly *parallel* activities (Hetzel 1988). Acceptance testing parallels requirements specification, integration testing is done in parallel with design, as are unit testing and implementation. This is because test case design and test data collection, the acquisition tasks, for, say, acceptance tests can commence as soon as even partial documents from requirements specification are available. Delaying these tasks means delaying test execution in the measurement phase, and it is this task that is on the critical path for any software project. The same applies to system, integration and unit testing and their matching development activities.

Figure 4.5 illustrates the synchronisation of test phases and their associated development activities. Here we see the acquisition phase occurring during the associated development phase. If we were to draw the “V” diagram (Figure 4.3) over the top of



**Figure 4.5:** Coordinating testing and software development phases (after Hetzel 1988)

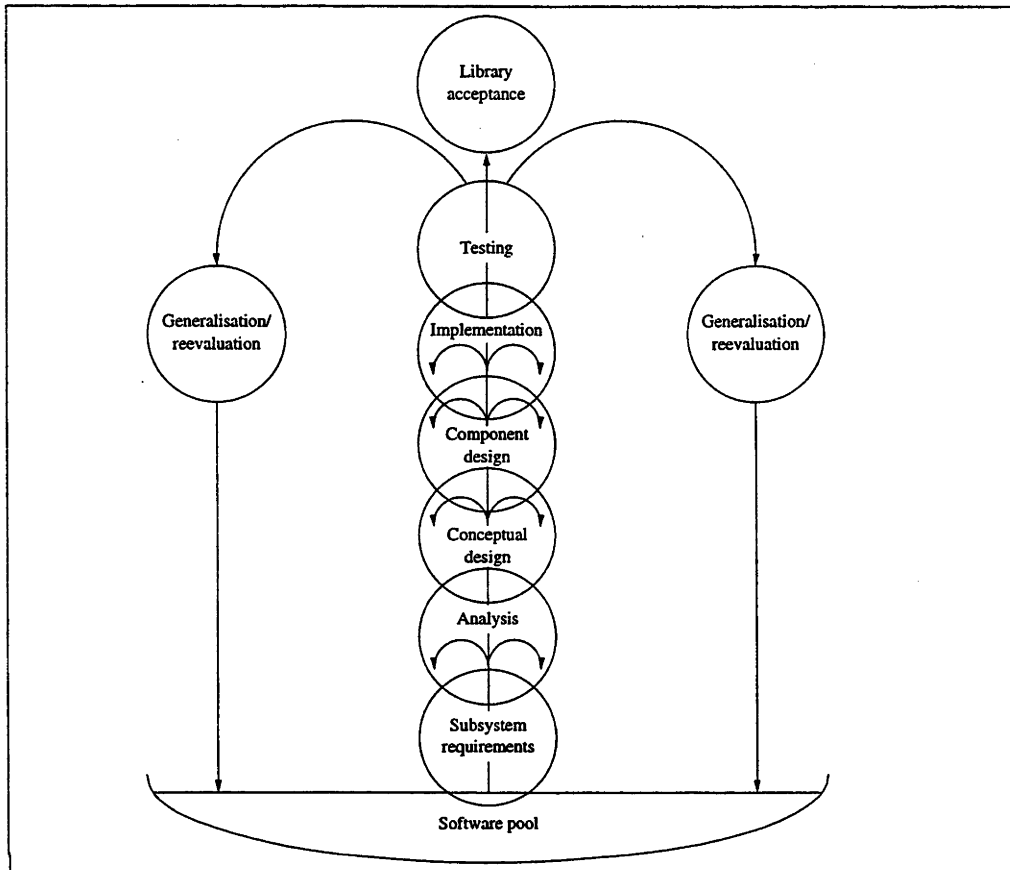
this figure, we would find that only the measurement phase lies on its “up-stroke”. Test planning occurs alongside other project planning activities, although some refinement of the test plan is inevitable once its execution has started, as indicated by the “refine” tasks in the figure.

Next, we leave testing for a moment to look at process models for object-oriented software development. In Section 4.4 we will introduce into those processes, the test phases and tasks described here.

### 4.3 Object-Oriented Software Development Processes

While object-oriented software could conceivably be developed in any of the software development process models we have mentioned in Section 4.1 above, the conceptual closeness of object-oriented analysis, design and programming encourages a highly iterative development style in which these phases are closely integrated. This has lead several authors of object-oriented methods to rethink the process (for example, Booch 1994, Henderson-Sellers & Edwards 1994, Jacobson et al. 1992)

Software development process models are, essentially, idealisations (Parnas & Clements 1986). As we have already seen, even in older process models such as the “waterfall” model, earlier phases are inevitably revisited because the implications

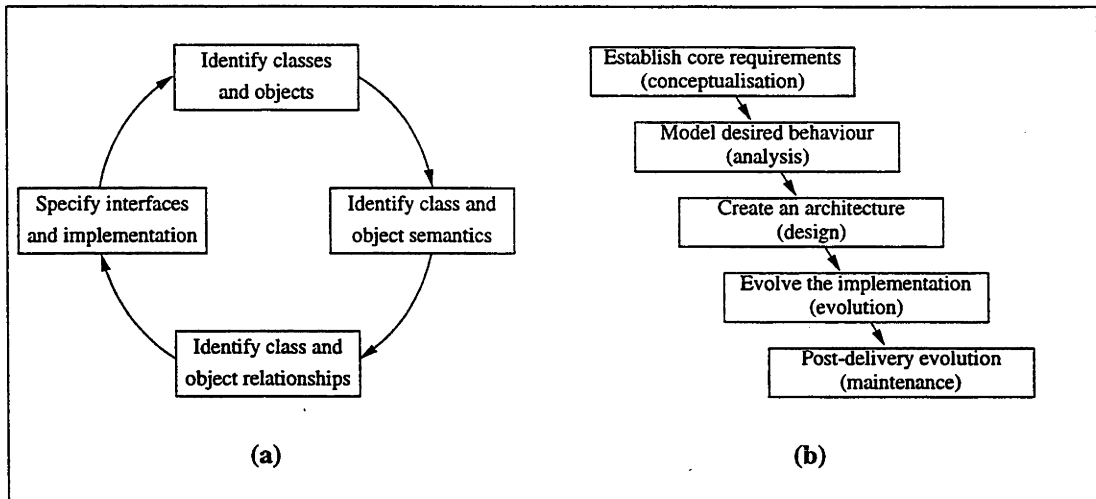


**Figure 4.6:** The fountain model of software development

of decisions made in those phases cannot be completely understood until it is worked through in later phases. There is a folk wisdom that in software, we need to build something three times to get the design right (Love 1993, Lorenz 1993).

This experience has, to some extent, been absorbed into the practice of object-oriented software development. When authors of object-oriented methods describe a software development process, they typically suggest one that incorporates incremental and iterative elements from Gilb's evolutionary development and Boehm's spiral, described above.

A recent survey of sixteen object-oriented software development methods (Hutt 1994) found that of those that specified a development process, all supported some form of iterative development strategy, while less than half additionally supported a once through, waterfall-style strategy. Further, all surveyed methods used an additive progression strategy, where each phase contributes new objects and details to an existing model, but less than half also transform a model from one phase to the model that will be the basis of the next phase. Transformational progression, which is typical



**Figure 4.7:** The Booch micro process (a) and macro process (b)

of structured analysis/structured design processes, implies a sequential, waterfall style development. By contrast, in additive progression the current phase of development is less clear and it is possible to swap rapidly between analysis, design and implementation activities.

One of the earliest representations of the iterative nature of object-oriented software development is the “fountain” model (Henderson-Sellers & Edwards 1990). In this model the main thrust of development is upward (see Figure 4.6). However, individual software items and other products will, like droplets of water in a fountain, fall back to earlier phases only to be caught in the upward thrust again. Eventually they fall back to be collected in a repository where they can be used again in other projects

Once again this model is an idealisation. By turning the waterfall model on its head, the fountain model has given to the software development process the flexibility to design a better solution, but at the cost of management control. Where there was an ordered waterfall there is now a chaotic fountain. A practical object-oriented software development process must address entry and exit criteria for tasks as well as the progression of tasks.

For the purpose of illustrating the object-oriented software development process and the part that is (or could be) played by testing, we will use as an example the Booch process. Booch (1994) provides an overall idealised structure and general descriptions of the tasks making up the process.

### 4.3.1 The Booch Process

Grady Booch is credited with the aphorism “analyse a little; design a little; code a little; test a little,” emphasising fine-grained iteration and incremental delivery, which has become a catch-cry of object-oriented software development<sup>1</sup>. This idea is embodied in his micro development process (Figure 4.7a). The micro process is directed by a macro development process (Figure 4.7b) that guides development to completion and manages risk. The Booch process is described in (Booch 1991, Booch 1994). Isseult White in her tutorial on the Booch method (White 1994) describes the earlier phases of the macro process in more detail.

The micro process iterates through the four steps in the figure to build an “executable release”<sup>2</sup> which could be delivered to the customer. Each iteration adds a new increment of functionality to the executable release. At the same time, these steps form a sequence of tasks for producing and refining a set of documents that form the architectural description of the system. We can deduce from Booch’s reference to Parnas & Clements’s (1986) article, that their order is less important than their existence.

The first phase of the macro process, conceptualisation, seeks to establish the core requirements of the proposed system (White calls this phase “requirements analysis”). Booch suggests that the products of this phase are prototypes that establish vision and validate assumptions. White recommends producing a system function statement and a system charter. The first captures the key use cases<sup>3</sup> and sample outputs. The latter is a statement of the system’s responsibilities and scope. These form a contract with the customer, although they will change as development proceeds.

In the second phase of the macro process, analysis, Booch advises that we focus on outwardly observable and testable behaviours. According to White, who calls this phase “domain analysis,” this includes identifying all major objects, data and operations need to carry out the system’s function and the result should be a precise and concise object-oriented model of the problem domain. Products of this phase include object, class and inheritance diagrams, object scenarios and a data dictionary or CRC cards<sup>4</sup>.

---

<sup>1</sup>Ed Berard also claims partial credit for this expression (Berard 1993a).

<sup>2</sup>White’s book has an interesting bug: a short extract from the text is printed on the cover, but where the words “executable release” have been replaced by “prototype”. Perhaps it was taken from an earlier executable release/prototype of the text, which leaves room for speculation as to why the change was made.

<sup>3</sup>Use cases are a high-level modelling technique developed by Jacobson et al. (1992). A use case is a scenario of a user initiated interaction with the system that is an exemplar or shows a pattern of usage, for example a customer withdrawal in a banking system.

<sup>4</sup>CRC cards were popularised by Wirfs-Brock et al. (1990) and are used in a number of object-oriented design methods. CRC stands for Class-Responsibilities-Collaborators. Each card describes a class in terms of its inheritance hierarchy, the operations it provides for other classes (responsibilities), and the classes providing services that support those operations (collaborators). The cards were originally 3”×5” file cards on which this information was laid out, but could also be entries in an automated

---

During the design phase we define an initial architecture for the proposed system leading to the first executable release (a version of the product that is released to the customer). It would appear that White sees this as the first iteration of the evolution phase (she calls it “system design”), in which a series of executable releases are produced.

In the process of creating executable releases (the design, evolution and maintenance phases), a number of documents are generated and revised. One is an architecture description which may include class, object, cluster and subsystem diagrams, class specifications, object scenario diagrams and state-transition diagrams, among others.

## 4.4 Testing and Object-Oriented Software Development

This section will first motivate and outline a general framework for integrating testing into an object-oriented software development process, before examining testing in the Booch process.

When testing waits until coding is complete, development in effect reverts to a waterfall process—with its accompanying problems. In particular, testing becomes limited by the schedule, with any over-runs in other phases pushing testing against the deadline for the whole project. This is the problem which was discussed in Section 4.2. Just as in other process models, testing effort in object-oriented software development must be distributed throughout the other development phases, so an iterative development model must include testing in the iterations.

The question then is how should testing be integrated. The answer lies in an understanding of the test process, as outlined in Section 4.2.1. There we saw that testing consisted of three phases: a planning phase, an acquisition phase, and a measurement phase. Following the distribution of effort recommended in Figure 4.5, we should see development and test case acquisition as concurrent activities, and measurement occurring at the end of each iteration.

Each phase of development produces a specification of some aspect of the final software product in some form and at some level of detail. Whether it be a requirements document, a class interaction diagram or source code, we can view them as specifications. Each item specified also creates a test requirement. Each functional requirement, class interaction or algorithm is also a criterion that the software should satisfy and each such criterion can be evaluated by one or more test cases.

The approach suggested here is to associate with each specification or design document a set of test requirements. These test requirements are developed along with their associated specification document. During each iteration these documents may be revisited, in which case associated test requirements should be updated. Test cases are

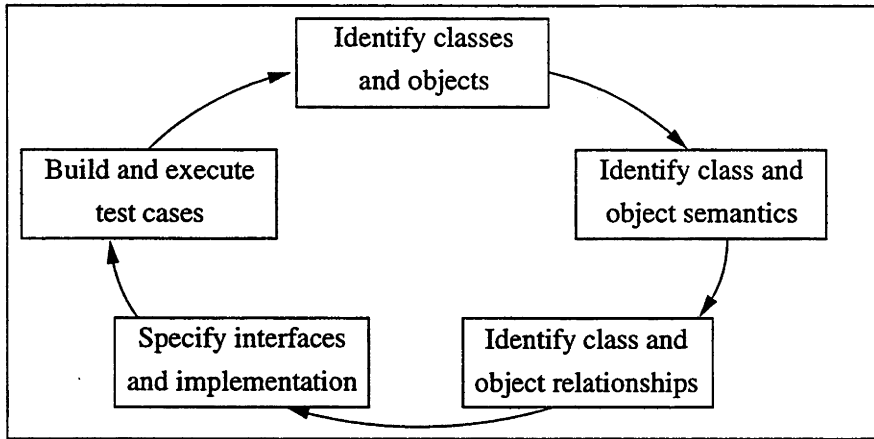


Figure 4.8: Adding testing to the Booch micro process

developed or updated from the test requirements and executed in each iteration. Test planning determines the acceptance criteria for each release which, in turn, determines the criteria used for selecting test requirements.

Let us now see how this approach can be applied in the Booch software development process.

#### 4.4.1 Incorporating Testing in the Booch Process

The iterations in the Booch process are captured by the micro process. So the acquisition, execution and revision of the test suite has to be integrated into the micro process. There are two aspects to this. One is to add an extra step to the micro process, “build and execute test cases,” to look for defects in the executable release currently under development. This is illustrated in Figure 4.8. The other aspect is to incorporate test requirement acquisition into the other steps of the micro process.

The purpose of the added step is to find defects in the system in its current state of development. To do this we update and execute the current test suite, report defects found and collect and report the test results. The test results contribute information to the risk analysis that precedes decisions to release the current product and on the focus of the next iteration. This step also marks the closure of each micro process iteration.

In order to execute the test cases in this new step, they must first exist. Test requirement acquisition must occur in the other four existing steps the micro process. To each of the documents that are generated and revisited in the micro process, we should add a new section containing the test requirements that are derived from it. The kinds of test requirements to be collected in each document can be specified in the test plan.

Turning now to the macro process view, the products of the conceptualisation phase address the customer, so the test effort should focus on acquiring acceptance tests. The



---

use cases in the system function statement describe a user's interaction with the system to produce some result. Each use case will be the basis of one or more acceptance test requirements and the possible interactions between them will provide more. The use case's variables and their domains should be identified and equivalence partitioning (see section 2.2) used to expand the set of test requirements (Binder 1995). The sequence of inputs and operations required to produce the sample outputs can also be used to find acceptance test requirements. Prototypes are not very useful as a source of test requirements but can be used as "oracles": either to derive the expected results of test cases, or to compare the results of executing test cases with the system under development.

During the analysis phase we identify system test requirements. These will be derived from the requirements and behaviours of "system level" or "problem domain" objects and classes. In particular, each object scenario defines at least one test requirement, as do its alternate paths. Equivalence partitioning on the scenario's input data can be used to select further test requirements.

The process of creating executable releases, that is the design, evolution and maintenance phases, generates and revises a number of documents. These constitute an architecture description may which include class, object, cluster and subsystem diagrams, class specifications, object scenario diagrams and state-transition diagrams. These are all sources of unit, integration and system test requirements. Those documents relating to an individual class will provide unit test requirements. There will be several levels of integration test requirements each testing the interactions between classes, clusters and subsystems. System test requirements are those that examine the behaviour and performance of the system as a whole. Taken together, these form a test suite for the executable release.

Many of the techniques described in Chapter 3 can be applied to derive test requirements from the documents comprising the architecture description. For example a class specification is a source of functional unit test requirements for the class. At a minimum we would expect each method to exercised and equivalence partitioning on the method arguments could be used to further refine the test requirements. State-transition machine diagrams permit testing strategies described in Section 2.4. Every path through object interaction diagrams and scenarios can be expressed as an integration test requirement.

The added "build and execute test cases" step ensures that test cases reflect the current state of development of the product they test. The documents constituting the architectural plan will be revisited many times during iterations of the evolutionary phase. Since the test requirements are associated with the document from which they are derived, they are revisited together with that document. Test cases in turn, are linked to the test requirements they were developed from. In the added step the test cases are updated from the changes to these test requirements. In this way a test suite for the system is built up and maintained in a way that tracks the system's development.

Part of the assessment of the current state of the product is an assessment of the adequacy of the test suite. The added step should include a review of the test cases for this purpose. The test suite should be reviewed against adequacy criteria specified in the test plan.

In a process based on incremental delivery, as the Booch process is, the customer will receive many versions of the system, each providing some new functionality or features. It is important that functionality that was added in one version is not lost in a later version. That is the role of regression testing. The step added in Figure 4.8 builds regression testing into the Booch micro process, ensuring that functionality and quality are not lost between executable releases.

## **4.5 Other Object-Oriented Process Models**

The approach just described for incorporating testing into object-oriented software development can be applied to other object-oriented software development processes. It requires that we

- identify which of the documents that are generated during development are potential sources of test requirements,
- ensure that as part of each iteration test cases are built from the test requirements, collected into a test suite and the test suite exercised,
- ensure that the test requirements are updated as their associated documents are revisited during iterations of the development process,
- ensure that test cases are updated when the test requirements from which they were derived are updated, and
- ensure that the planning process determines what level of testing is required for each of the identified documents in the context of the quality goals for the proposed releases of the product.

In a development process with many iterations, the documents and their test suites will be revisited many times and the same test case may be run many times. It is useful to automate as much of these repetitive and clerical tasks as possible. There is a range of existing tools to support activities such as regression testing. Tools for identifying test cases is an area of continuing research.

In the rest of this section we look briefly at applying this approach to testing to a few other object-oriented software development processes.

---

## MOSES

Software development in the MOSES method (Henderson-Sellers & Edwards 1994) has five phases: planning, investigation, specification, implementation and review; and some twenty activities that occur during one, some or all phases. In addition to iterating through a sequence of tasks as in the Booch micro process, MOSES allows the possibility of falling back to an earlier phase as suggested by the fountain model (Figure 4.6).

Each phase of the MOSES process is concluded by an inspection or review of the products of that phase. They recommend unit testing of classes by the developer at the end of implementation, and the review phase includes class integration testing, subsystem integration testing and system testing. Thus each iteration is somewhat like a miniature “V” model.

In fact MOSES already contains the necessary test steps. What is needed is to move test requirement acquisition into activities where products are created and revised.

## Classworks

Classworks (Ratjens & Steele 1993) is in a similar vein to MOSES. Tasks are identified on a three-dimensional grid whose dimensions are the application framework which is an architecture in which systems are built; a “technical cycle” whose steps are design, construct and review; and a “management cycle” consisting of the cycle of tasks: plan, initiate or do, monitor and review.

Those tasks in the plane of the cube marked by the “review” step of the technical cycle provide the right place to test planning, where it intersects with “plan” in the management cycle, and test execution. Test requirement acquisition needs to be incorporated into the other steps of the technical cycle.

## Objectory

In Objectory (Jacobson et al. 1992), object-oriented software development is based around identifying, describing and implementing use cases. The system is delivered in increments of added use cases. They suggest that testing is a parallel activity that should begin early in the process. They do not, however, integrate it into development as is described in the previous section. On the other hand they also say that testing can not begin until there is an implementation, a view that is contradicted by the test process described in Section 4.2.1. Integration testing and system testing is based on executing use cases.

## **4.6 Summary**

In this chapter we presented a framework for integrating testing into an object-oriented software development process. This is, of course, an idealised view of the testing process and should to be viewed in the context of software process improvement.

In an iterative and incremental software development process, typical of object-oriented software development, the development activities can no longer be readily identified by their position in a life-cycle but rather by their inputs, deliverables and purpose. So it should be with testing. Testing is more efficient and effective, and has less impact on the release date, when test activities are scheduled into the iterations.

---

# Testing Objects as Finite State Machines

---

In this chapter, we focus on the behaviour of objects in terms of states and state transitions, which we introduced in Section 3.3.3. We develop a technique for testing object state that both reconciles the design view of object state with that of object-oriented programming, and also builds upon the established techniques for testing finite state machines first mentioned in Chapter 2.

The first section discusses finite state machines and object-oriented design. Section 5.2 describes our method for relating the finite state machine view of object behaviour to an implementation in an object-oriented programming language. The applications to testing are addressed in Section 5.3 and our approach to generating test cases is described in Section 5.4. In Section 5.5, we relate the concepts presented here to the work of others.

This chapter is derived from a joint paper with Prof. Heinz Schmidt, given at the TOOLS Pacific '95 conference (Bosman & Schmidt 1995a). An earlier version is also available as a technical report (Bosman & Schmidt 1995b).

## 5.1 State-Based Models for Class Design

As we have seen in Section 3.3.3, objects have state and behaviour, and these two aspects of objects are closely related: the state of an object affects its behaviour in response to the messages sent to it, and this behaviour may, in turn, change the object's state. In Section 3.3.3 we introduced the concept of design states. In this section we show how the Abstract behaviour of many kinds of objects can be represented by finite state machines based on design states.

In Section 2.4, we defined a (deterministic) finite state machine as a sextuple,  $M = \langle I, O, S, s_o, f, g \rangle$ . For an object, the inputs  $I$  represent messages accepted by the object, that is, a method call and actual parameters, and the outputs  $O$  are messages sent to other objects. The initial state  $s_o$  is the initial state of the object when constructed. The transition function  $f$  and output function  $g$  can be derived from the

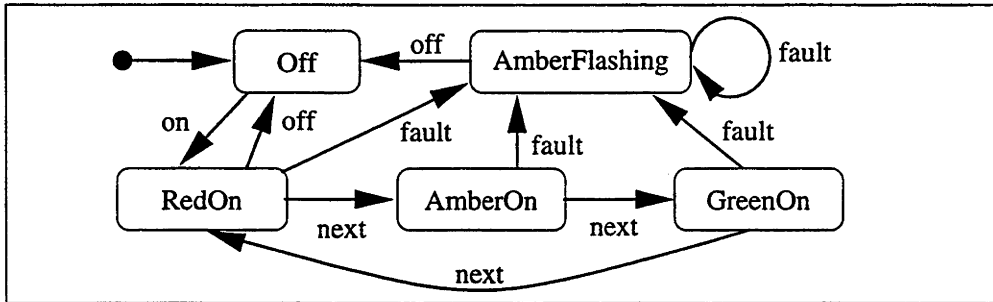


Figure 5.1: State-transition diagram for a TRAFFIC\_LIGHT.

actions of the object's methods.

As we have seen in Section 3.3.3 and in Chapter 4, many object-oriented design methods use statecharts (Harel 1987) or some derivative of them such as state-transition diagrams (Booch 1994, Rumbaugh et al. 1991, Shlaer & Mellor 1988), to specify the dynamic behaviour of objects. In state-transition diagrams, states are indicated by rounded boxes and transitions between them by arrows. Transitions are labelled by the event (message) that causes them, and may have a "guard condition" in square brackets. The initial state is indicated by an arrow with no source, corresponding to a creation message (a call on the constructor method). We call these finite state machine models *design* finite state machines, their states *design states* and their transitions *design transitions*.

Binder (1996) classifies the state-based behaviour of objects into four kinds, which he calls nonmodal, unimodal, quasimodal and modal.

**Nonmodal** classes place no constraints on the sequence of messages they will accept.

Nonmodal behaviour is typical of simple data types, for example a `TIME` class. Any sequence of calls to the methods `set`, `get`, `is_equal` of a `TIME` object is accepted. When partitioned into design states, nonmodal classes do not exhibit obvious state-based behaviour: all messages cause transitions in all states.

**Unimodal** classes accept messages based only on the sequence of previous messages, not the message content. Application control objects are typically unimodal. When partitioned into design states, unimodal classes can be represented by state-transition diagrams without guards. Figure 5.1 shows a finite state machine representation of the behaviour of a `TRAFFIC_LIGHT` object.

**Quasimodal** classes accept messages depending only upon the internal state of the object. For example, a `pop` on a `STACK` object is not accepted when it has no elements. A state-transition diagram for a quasimodal class will have guard conditions, but they are only on attributes of the object. This behaviour is common

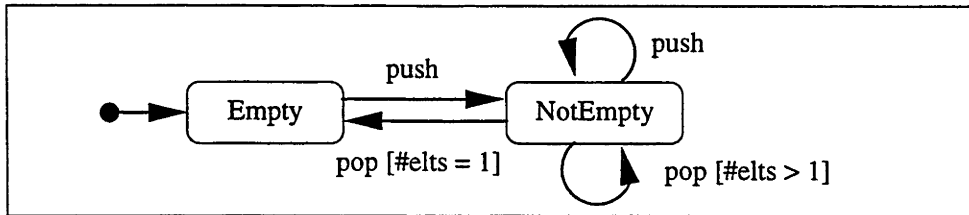


Figure 5.2: State-transition diagram for a STACK.

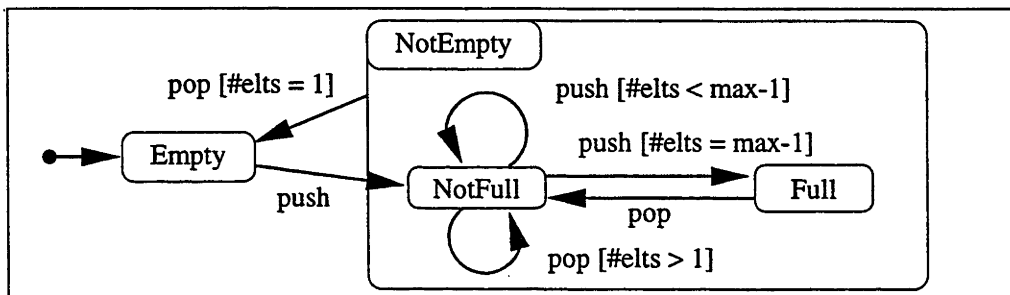


Figure 5.3: State-transition diagram for a BOUNDED\_STACK.

to many container or collection classes. Figure 5.2 shows the state-transition diagram for a STACK. To avoid cluttering the diagram, some methods that do not change the state of the object, such as `top`, are not shown.

**Modal** classes may also reject messages according to their content, that is, the actual arguments. For example, an ACCOUNT object will only accept a `debit` message if the amount argument is less than the balance (see Figure 6.1 on page 86). Notice that subsets of a class's methods may show behaviour in different categories. If we ignore all but ACCOUNT's `open` and `close` methods, it is unimodal and restricting to its `balance` is nonmodal. Both modal and unimodal classes have distinct finite state machine behaviour, which can be tested by transition trees (Binder 1996).

In Section 3.4.4 we discussed the relationship between a state-transition diagram for a subclass and that for its parent. Figure 5.3 shows the state-transition diagram for a specialisation of STACK, the BOUNDED\_STACK. Notice that the new states have been added as substates and the finite state machine of the parent has been preserved. In this case, however, not all sequences of inputs to a STACK are legitimate for a BOUNDED\_STACK, for example a BOUNDED\_STACK of length 10 will not support a sequence of 11 consecutive pushes.

## 5.2 Representation and Implementation

In Section 3.2 we introduced a graph representation for the internal states of objects. The nodes of this graph are objects in an executing program. Each (directed) arc is a reference from its source object to its destination object. The source of a reference can be a local variable of a currently executing method, an actual argument of a currently executing method, or an attribute of an object. Each arc is labelled with the name of the variable, formal argument or attribute. The current value of attributes of basic type (for example, INT or BOOL) can be kept with their object's node. These state graphs have been used to reason about the behaviour of object-oriented programs (Schmidt & Zimmermann 1994a, Schmidt & Zimmermann 1994b, Schmidt & Chen 1995).

In practice, the mapping from design states to internal states is often not explicit and state transitions are usually buried in the method implementations. All these factors can make it difficult to validate that an implementation is correct with respect to its design.

We suggest an abstract representation of implementation decisions as a *representation* finite state machine which captures the most important internal states as *representation states* or predicates together with an abstraction of implementation transitions. The corresponding finite state machine invariant is called *representation invariant* following the terminology of CLU (Liskov & Guttag 1986) and Alphard (Wulf et al. 1976).

With this distinction, each design state corresponds to one or more representation states, that is, vectors of values for the object's attributes. So, the design states of an object can be represented by partitioning its representation states. The representation state predicates are defined on attribute values of the object. Furthermore, we assume that there is a similar mapping from representation transitions to design transitions. Representation finite state machines make the mapping between design and implementation levels explicit, and simplify the validation of the dynamic behaviour of classes.

In methodologies such as RSML (Leveson et al. 1994) and Shlaer & Mellor (1992), finite state machines are developed in requirements and analysis, and transformed into implementation FSMs. In this approach there is a one-to-one mapping from design to representation states that is constructed by the method. Our proposal extends this concept to other methodologies that do not use finite state machines as a basis for requirements and analysis.

Figure 5.4 summarises the proposed testing strategy. Our aim is to validate the implementation of the class against its specification. At the specification level, a class design has structural elements which are captured in the class interface, namely methods and attributes, and their types. The class design also describes the class's behaviour which we can capture in its design finite state machine. At the implementation level, that the class body conforms to its interface can be checked statically by a compiler.



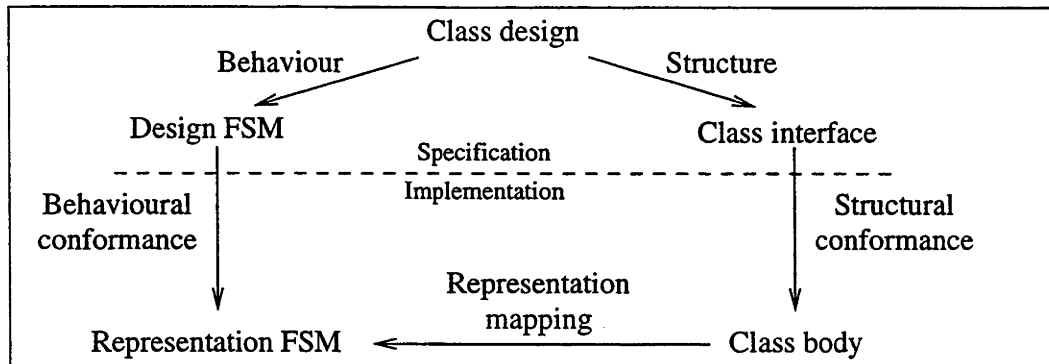


Figure 5.4: Different views of object/class behaviour.

Checking behavioural conformance requires testing, and the representation finite state machine provides a natural vehicle for comparing the behaviour of the design finite state machine to that of the class body. The representation mapping we describe here projects the implementation onto a representation finite state machine. Then, as we shall see in Section 5.3, we can determine the behavioural conformance of the two finite state machines using the state-based testing techniques described in Section 2.4.

A simple example of the use of representation state predicates can be seen in Figure 5.5, which shows an implementation in Sather of the `STACK` class whose finite state machine specification is illustrated in Figure 5.2. Two representation states are defined by the predicates *Empty* and *NotEmpty*, corresponding to the similarly named the design states from the state-transition diagram.

While the design finite state machines of those classes related by inheritance are similar or identical, a corresponding relation is not present at the level of code. The implementation states are the values of an object's attributes. When these are hidden, as suggested by good programming practice, inheritance makes no restriction on their addition or removal. In the absence of language support, preserving the correct finite state machine behaviour requires a programming discipline.

A Sather implementation of the `BOUNDED_STACK` specialisation of `STACK` can be found in Figure 5.6. The new states in the design finite state machine (see Figure 5.3) partition the *NotEmpty* state of the parent. In the implementation, this appears in the redefined predicate `notempty`, which has become the disjunction of the representation state predicates for the added substates. Thus the invariant of the parent still holds and is inherited unchanged.

Clearly, since design states are abstractions of attribute values (both basic types and referenced objects), any design state can be represented by a predicate on those attributes. Furthermore, if every valid representation state of the object makes a predicate true, the disjunction of all these predicates is an invariant for the object. For

```

class STACK{T} is
  -- Implementation of stack using arrays
  stack: ARRAY{T};
  ssize: INT;
  constant initsize := 5;

  create: SAME is
    res := STACK{T}::new(ssize := 0);
    -- create an empty array for storing the elements
    res.stack := ARRAY{T}::new(size := initsize);
  end; -- create

  push(x:T) is
    if stack.size = ssize then -- extend the stack
      h: ARRAY{T} := ARRAY{T}::new(size := 2*ssize);
      -- copy all the elements across to the new array.
      i: INT := 0;
      while not i = ssize do
        h[i] := stack[i];
        i := i+1;
      end; -- while
      stack := h;
    end; -- if
    stack[ssize] := x;
    ssize := ssize + 1;
  end; -- push

  pop:T is if notempty then ssize:=ssize-1; end; end;

  top:T is if notempty then res := stack[ssize-1] end; end;

  -- Representation state predicates
  empty:BOOL is res := ssize = 0 end;
  notempty:BOOL is res := ssize > 0 end;

  -- Representation invariant
  invariant welldefined = empty or notempty end;
end; -- class STACK

```

Figure 5.5: A STACK class implemented in Sather

each method, the finite state machine design specifies the possible transitions between design states. In the implementation, this can be expressed in a post condition for the method using the representation state predicates. In languages that support invariants and pre- and postconditions, such as Sather, explicitly describing the mapping to the finite state machine design enhances the robustness of the implementation (Meyer 1992a).

```

class BOUNDED_STACK(T) is
  -- Implementation of bounded stack
  include STACK(T); -- reuse implementation of STACK.
  constant max_size := 1000;

  create: SAME is
    res := BOUNDED_STACK(T)::new(ssize := 0);
    -- create an empty array for storing the elements
    res.stack := ARRAY(T)::new(size := max_size);
  end; -- create

  push(x:T) is
    if notfull then
      stack[ssize] := x;
      ssize := ssize + 1;
    end; -- if
  end; -- push

  -- Representation state predicates.
  full:BOOL is res := ssize = max_size end;
  notfull:BOOL is res := ssize < max_size end;
  notempty:BOOL is
    -- Redefine parent's 'notempty' state (assume max_size>0).
    res := full or notfull
  end;
end; -- class BOUNDED_STACK

```

Figure 5.6: A BOUNDED\_STACK class implemented in Sather

## 5.3 Testing Finite State Machines

To test a representation finite state machine, we use the test sequences  $T$  derived from its transition tree by Chow's W-method, which we first met in Section 2.4. Here we adapt this method to state-transition diagrams with substates and concurrent states. The nodes of the tree represent states of a state-transition diagram and the arcs represent transitions.

In practice the restriction to a minimal finite state machine presents little difficulty, since the specification represented by the design finite state machine will typically be minimal or can readily be minimised (Hopcroft & Ullman 1979). In fact, we are less concerned to show the equivalence of the design and representation finite state machines for a class than to generate test cases that are likely to expose differences in their behaviour. The value of Chow's result is that  $T$  is sufficient: it "covers" the behaviour described by the design finite state machine. We do not claim that  $T$  is a complete test of a class, since a design finite state machine does not completely describe its behaviour, and as we have seen, there are classes whose behaviour is not well captured by a design finite state machine. Using the W-method we can automatically generate a test suite that efficiently and effectively validates the behaviour described by a design finite state machine.

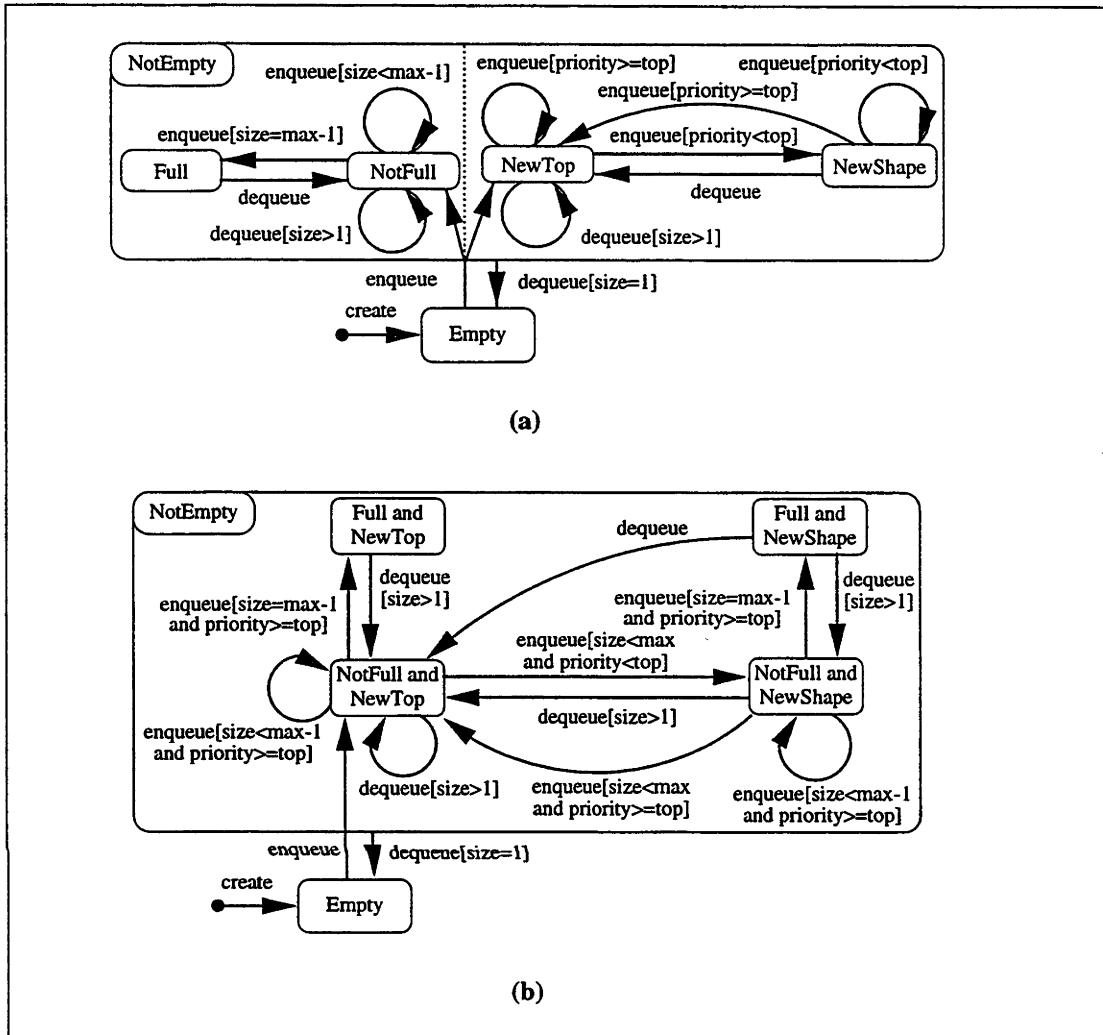


Figure 5.7: Eliminating concurrent states.

The first step is to eliminate any concurrent states by replacing those sub-states with an equivalent, non-concurrent state-transition diagram (Harel 1987). Figure 5.7(a) shows the state-transition diagram for **BOUNDED\_PRIORITY\_QUEUE** from Section 3.4.4. Its *NotEmpty* state has two concurrent substates, so a **BOUNDED\_PRIORITY\_QUEUE** will be simultaneously in both, that is, it will be in both *NotFull* and *NewTop*, *NotFull* and *NewShape*, *Full* and *NewTop* or *Full* and *NewShape*.

Figure 5.7(b) shows a state-transition diagram equivalent to Figure 5.7(a) that is non-concurrent. The states in the new state-transition diagram are all pairs of possible concurrent states. Transitions are derived from those in the concurrent version in the obvious way, conjoining guard conditions as necessary. For example, in the concurrent

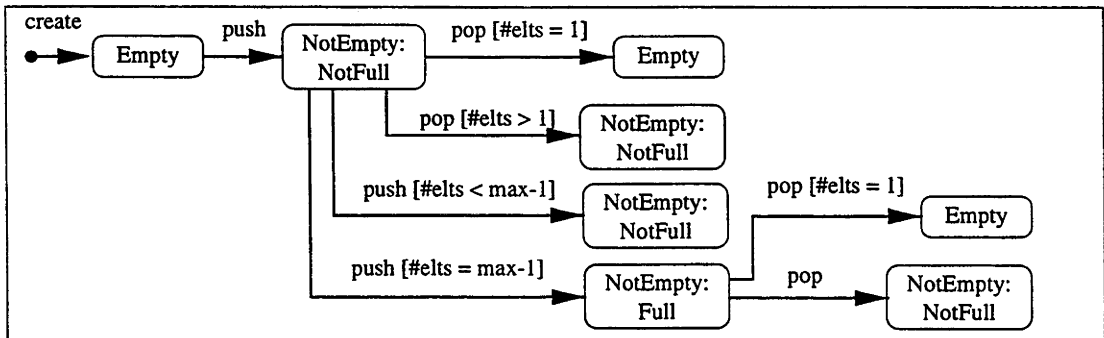


Figure 5.8: A transition tree for BOUNDED\_STACK.

version a *dequeue* message in the *Full* state transitions to the *NotFull* state and also from *NewShape* to *NewTop*, so in the non-concurrent version *dequeue* transitions from the new *Full* and *NewShape* state to *NotFull* and *NewTop*. Another example is the transition *enqueue[size=max-1 and priority>=top]* from the new *NotFull* and *NewTop* state to *Full* and *NewTop*, which results from combining the *enqueue[size=max-1]* transition from *NotFull* to *Full* and the *enqueue[priority>=top]* transition on *NewTop*. Notice how the derived guard condition on the new transition is the conjunction of the guard conditions of the transitions in the version with concurrent states. If there are three or more concurrent substates they can be replaced in this manner pair-wise.

To construct a transition tree for a state-transition diagram without concurrent states we proceed exactly as in Section 2.4 except that in each node we include the names of all containing states. Figure 5.8 shows a transition tree derived from the state-transition diagram for a BOUNDED\_STACK in Figure 5.3.

Thus we can derive a set of traces or sequences of transitions that cover the representation finite state machine. Each trace is a test case of the implementation relative to its representation finite state machine. Each trace is implemented by a sequence of calls to the appropriate methods in the code guarded by tests of the preconditions. If the test cases find no errors, then the implementation is complete and correct with respect to its representation finite state machine (Fujiwara et al. 1991, Chow 1978). Since we can map the states and transitions of the representation finite state machine to those of the design finite state machine, this is also a cover of the design finite state machine.

While representation invariants could be related by subtyping, we would gain an advantage from this situation only in a case where the design finite state machine, *D* say, stays fixed and the representation finite state machine (*R* say) is changed to *R'*. If we generate a coverage of *R* by *R'* we could reuse the coverage of *D* by *R*. We do not expect the benefits of such an approach to outweigh the restrictions introduced by sub-

typing, viz. the requirements of not weakening representation invariants. Therefore, in the context of this project we prefer that representation invariants of subclasses are not related at all under subtyping.

## 5.4 Test Case Generator

Chapter 6 describes the design of a test case generator using the principles described here. Constructing the representation finite state machine requires adding the predicates defining representation states to the class. However we believe that this is a small portion of the implementation effort. Testing is typically in excess of 40% of total development effort (Pressman 1992). By basing our testing on finite state machine models, test data generation is straight forward using well established techniques.

A further advantage is the clear semantic relation between finite state machines and the class invariants. This has the additional benefit of documenting design (from the design finite state machine) and implementation decisions (in terms of the representation finite state machine). So the understandability and hence maintainability of software is increased.

The representation finite state machine drives the tests of the class's code. The design finite state machine defines the expected outcomes. The level of testing, or how much of a class's behaviour is actually tested, is defined by the granularity of the representation finite state machine. How much of this behaviour is observed or required is defined by the granularity of the design finite state machine. The more detailed these two aspects of the behavioural model, the more coverage of the interface during testing and the greater the confidence in the correctness of its workings. By selecting the appropriate level of detail for the design and representation finite state machine, it is possible to strike a balance between the cost of validation effort, and targeted reliability and robustness.

Ideally we would also like to have some notion of coverage for the code itself rather than just the behavioural model as described by the design finite state machine, which is typically a coarse abstraction of the actual behaviour. Such a coverage analysis would have to compare the control flow graph of the program with the representation finite state machine. Once this is established, it will be possible to design guidelines for combining the test coverage system with runtime instrumentation.

## 5.5 Related Work

State-defining predicates are used in (Chambers 1993), where the class of an object is determined dynamically according to whether its attributes satisfy that class's predicate. This is a powerful design technique and simplifies class implementation, but

---

it requires multi-methods and so its applicability is limited to the few object-oriented programming languages that support them.

Over the last two decades there has been much work in generating test cases from various kinds of specifications. Some of the more significant include (Gannon et al. 1981, Maurer 1990, Korel 1990, Ostrand & Balcer 1988, Balcer et al. 1989, DeMillo & Offutt 1993). More recently Barbey et al. (1996) have developed a tool that generates tests from a specification in an object-oriented variation of Petri nets. Some work has also been done on adapting ADL specification language and test generation tools to C++ (Viswanadha & Sankar 1996).

Hoffman & Strooper (1995) use an approach to class testing that is in some ways similar to ours. They also partition a class's implementation state, from which they derive appropriate transitions, creating in effect a finite state machine. Rather than deriving finite state machine-based test cases from the class design specification, an implementation of this finite state machine is used as a test oracle, and a test set should cover nodes and arcs of this "test-graph." In this case finite state machine testing techniques could also have been used to show the equivalence of the class and the test-graph, as we propose.

The ACE tool (Murphy et al. 1994), like the testgraph method just mentioned, is derived from the PGMGEN tool for testing C code (Hoffman & Brealey 1989), but uses a very different strategy. The tester is required to develop their own test cases in a script file. ACE generates the test drivers and runs the test cases.

The StP/T tool (Poston 1994) also uses design information and input data values to generate test case inputs, but it does not generate expected outputs for the test cases. In practice there are too many test cases for a tester to manually enter the expected results, so the program under test is run as an "oracle" and the test suite is used for regression testing of changes to the system. This means that original bugs have to be found and removed from the test suite as well as the system. And as the product evolves, the tester must determine which failures are due to introduced bugs and which are due to the correct implementation of changes to the design or specification.

A more immediate approach to testing object state in C++ classes is described in (Turner & Robson 1992b, Turner & Robson 1992a). Object state, in this case the vector of attribute values, is seen as additional method arguments, so adequacy is addressed by covering a partitioning of the state values in the same way as the other inputs. To use their tool, the tester must develop a subclass of the class under test which identifies the representation states to be tested, and a test script. The tool generates a test driver which runs the test case in the script against the subclass defined by the tester.

Doong & Frankl (1993) derive a test suite for a class that consists of pairs of message sequences that should put an object in the same state. This in essence implements trace equivalence (Wang & Parnas 1994) for the class when viewed as a finite state machine. A further method for testing class behaviour based on Method Sequence

Specifications and Message Sequence Specifications (Kirani & Tsai 1994) can be derived from a complete finite state machine model.

## **5.6 Summary**

We have described a technique for validating the dynamic behaviour of objects, based on refinement of finite state machines, for which coverage becomes equivalence. The method is adaptable to software development priorities since it allows users to define granularity of testing by means of the representation finite state machine against which the implementation is tested. In the next chapter we describe a tool that applies this technique to generate test case from state-transition diagrams.



---

# STAT: Generating Test Cases from Object-Oriented Design

---

This chapter describes the result of a practical application of the ideas developed in Chapters 4 and 5. It describes STAT, a tool we propose for generating executable test cases from state-transition diagrams. STAT is an abbreviation for State-Transition Automatic Tester.

In Chapter 4, we described a framework for incorporating testing into the iterative and incremental development process recommended by most object-oriented development methods. In the approach suggested there, the gathering of test requirements, test design and implementation of test cases are conducted concurrently with the other analysis, design and implementation phases. A natural extension of these ideas is to include support for test analysis and test design in the CASE tools used to facilitate software design. By generating test cases from design documents, this tool demonstrates how part of that test design support could be achieved.

STAT implements the technique described in Chapter 5. In order for STAT to access the representation finite state machine of the class under test, it makes use of an interface, `STAT_TESTABLE`. When it implements this interface, a class describes the representation mapping from its implementation states to the design states in its state-transition diagram. The `STAT_TESTABLE` interface provides a standard means of accessing the testable behaviour of the class, much as “test pins” do on a hardware integrated circuit.

It is intended that STAT be integrated into a CASE tool to enable concurrent development of test design and software design. The aim is to allow the user of the CASE tool to move easily between designing the software and developing test cases from the design information. STAT makes use of information in the class interface specifications and state-transition diagrams, which are a normal part of object-oriented design and supported by most object-oriented CASE tools. The user is also required to supply data values for method arguments. These may either be the values the designer uses as examples in the design process, or the result of a more thorough test strategy such as domain testing.

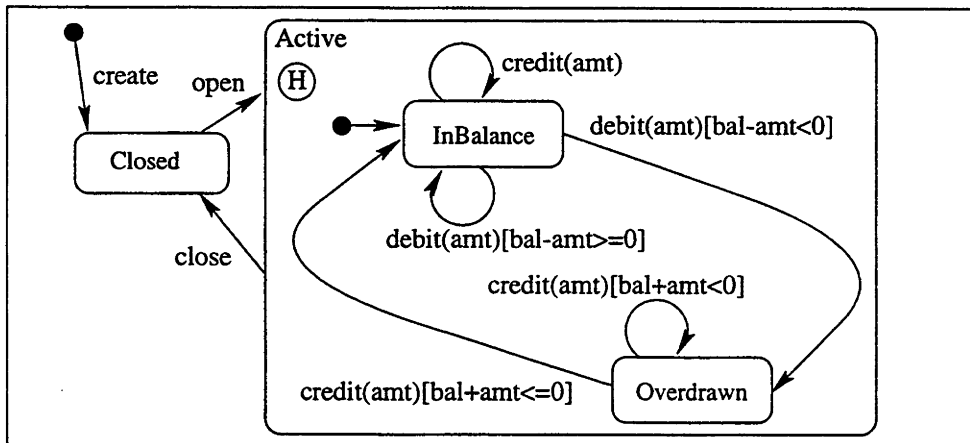


Figure 6.1: A state-transition diagram for an ACCOUNT object.

The next section describes informally the semantics of state-transition diagrams as they are used in object-oriented software development, and standard approaches to testing them. Section 6.2 describes the kinds of test cases generated by STAT. Some initial results are discussed in Section 6.3, then further work is proposed in Section 6.4.

## 6.1 State-Transition Diagrams

The example we shall use for explaining the operation of STAT appears in Figure 6.1. The complete input and generated code for this example appears in Appendix A.

The figure shows an abstraction of the dynamic behaviour of an ACCOUNT object such as might occur in a financial system. Upon creation the object is in the state *Closed* and the event *open* brings it into the *Active* state. The *Active* state is in fact a cluster of two substates: *InBalance* and *Overdrawn*. The first time it enters the *Active* state, the object is in the *InBalance* substate. In either substate, a *close* event takes the ACCOUNT object to the *Closed* state and a further *open* event returns it to its previous substate (this is indicated by the “history” marker  $\textcircled{H}$  on the *Active* state). While in the *Active* state, *credit(amt)* and *debit(amt)* events move the object between the *InBalance* and *Overdrawn* states. In the *InBalance* state, a *debit(amt)* event triggers either a transition to *Overdrawn* or back to *InBalance* depending upon the difference between *bal* and *amt*, as specified in the guard conditions. While in the *Overdrawn* state no further *debit(amt)* events will trigger transitions.

In Chapter 5, we described the use of transition trees to derive test cases from state-transition diagrams. Transitions leaving a superstate correspond to that transition occurring on each substate, but where the superstate has “history”, the transition tree for the subgraph outside the superstate must be traversed for each substate. Handling

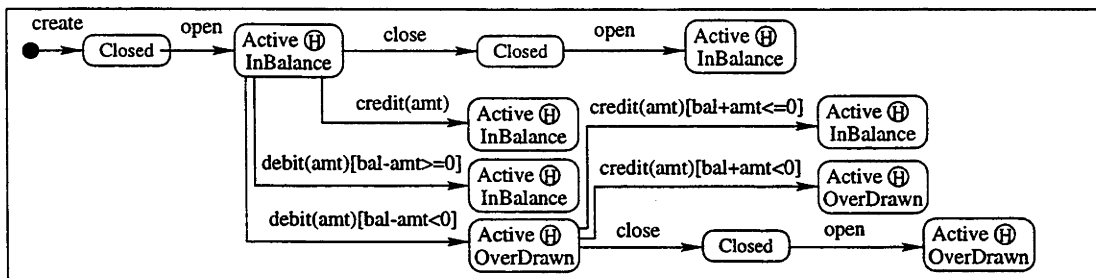


Figure 6.2: Transition tree for the ACCOUNT state-transition diagram.

Event sequence	Final state
1. create	Closed
2. create, open	InBalance
3. create, open, close	Closed
4. create, open, close, open	InBalance
5. create, open, credit(amt)	InBalance
6. create, open, debit(amt)[bal-amt ≥ 0]	InBalance
7. create, open, debit(amt)[bal-amt < 0]	Overdrawn
8. create, open, debit(amt)[bal-amt < 0], close	Closed
9. create, open, debit(amt)[bal-amt < 0], close, open	Overdrawn
10. create, open, debit(amt)[bal-amt < 0], credit(amt)[bal+amt < 0]	Overdrawn
11. create, open, debit(amt)[bal-amt < 0], credit(amt)[bal+amt ≥ 0]	InBalance

Figure 6.3: Test cases derived from the transition tree for ACCOUNT.

states with history requires a small extension to the algorithm presented in Chapter 5. Firstly, nodes whose state (or a superstate of which) has history are marked as such. Along paths leaving nodes marked with history, no nodes are marked as terminal nodes until that state is re-entered. An example appears in Figure 6.2, which shows the transition tree derived from the state-transition diagram for ACCOUNT in Figure 6.1.

The transition tree approach also assumes that the state-transition diagram is *connected*, that is, there is a sequence of transitions from the initial state to every state. However guards may make some transitions impossible or require a complex conjunction of conditions to trigger the transition.

Figure 6.3 lists the sequences of events and final states derived by the method from the transition tree for the state-transition diagram of the ACCOUNT object in Figure 6.2. Note the two sets of sequences that close and open the object from each of the substates *InBalance* (3 and 4) and *Overdrawn* (8 and 9) which test the “history” behaviour of the *Active* state.

```

abstract class $STAT_TESTABLE is
  -- An interface that is used by test code generated by STAT testing
  -- tool. Classes to be tested with STAT should implement this class.

  in_state($STR): BOOL;
  -- Returns true if 'self' is in a design state named in the argument,
  -- or false otherwise. This must match the name of a state in the
  -- state-transition diagram from which test cases are generated.

end; -- abstract class $STAT_TESTABLE

```

**Figure 6.4:** The Sather interface for querying an object's state.

The tool generates test cases from event sequences derived from the transition tree. Each event is assumed to be a method call on the object under test. Some of the events have arguments, and suitable values need to be supplied. The next section describes in more detail the input and output to the tool.

## 6.2 Generating Test Cases

STAT takes as input a signature (interface) for the class under test, a state-transition diagram describing the behaviour of that class and test data for types that are used in method arguments by the class. It generates a class containing test cases in the form of compilable Sather code. A method is created for each of the events sequences derived from a transition tree. The generated test class also has a main method that runs all the test cases and reports the results.

During design, state-transition diagrams are usually used somewhat informally. Booch (1996), for example, suggests that design documents should only have enough information to communicate the design to engineers. In order to generate test cases, however, the diagrams and the classes under test need to be “test ready.” This section describes assumptions made by the tool about the semantics of state-transition diagrams and the class under test, and the resulting test code produced.

STAT assumes that events in the state-transition diagram are calls on the methods of the class under test. These are checked against the class signature for name and number of arguments. The class signature also supplies the types for the arguments. STAT also assumes that guard conditions are fully specified as boolean-valued Sather expressions. Any identifiers either appear as formal arguments to the method or are themselves methods of the class under test. In Figure 6.1, the guard condition for the transition

```
debit(amt) [balance-amt>=0]
```

has an event argument named `amt` and `balance` is a method of `ACCOUNT`.

We also require some means of querying the current state of the object and inter-

```
event_seq_003 is
  test_label:= "event seq. 003: open, close";
  ob ::= #ACCOUNT;
  ob.open;
  ob.close;
  test(test_label, true.str, ob.in_state("Closed").str);
end;
```

**Figure 6.5:** Code generated for event sequence no. 3 in Figure 6.3.

preting the result as one of the design states represented in the state-transition diagram. The code generated by STAT assumes this is provided by a method `in_state` in the class under test. This method provides the mapping from the internal states of the object to representation states discussed in Chapter 5. The code generated by STAT assumes that this method returns true when passed the name of design state matching the current representation state. The abstract class `$STAT_TESTABLE`, shown in Figure 6.4, specifies the required interface. It can be inherited by the class under test which must supply an appropriate implementation. As we saw in Section 5.2, the implementation of this method is a matter of finding the predicates defining the representation states, and is usually straight forward.

For each event sequence, a separate method is generated. An object of the class under test is created and each of the events sent to it as method calls, then the final state is checked. Figure 6.5 is an example: it shows the code generated for the third event sequence in Figure 6.3. This code makes use of the `test` method of the Sather system class `TEST`, which records the result of the comparison of its second and third argument (the first is a label used for reporting the test case). The `TEST` accumulates these results and reports them at the end of the test run. The Sather `TEST` class is described in the Appendix, Section B.4.7.

Not many of the transitions for which we wish to generate test cases are quite so straight forward as this example: they may also have guards, and events may have arguments. Arguments on events correspond to method arguments under our assumption that events in a state-transition diagram are messages. STAT requires a set of one or more input data values for each argument type used in the state-transition diagram. In the `ACCOUNT` class example, the `credit` and `debit` methods both have a single argument of type `MONEY`. The tool determines the argument type from the class signature. Possible values for this argument are in the range 0 – 99 cents and \$0 – \$2147483647 (the largest signed integer representable in 32 bits<sup>1</sup>). A suitable set of input values can be found using the equivalence partitioning and boundary value anal-

---

<sup>1</sup>“Real testers”, whether or not they eat quiche, withdraw \$2147483648 from an ATM to see if it adds \$1 to their account’s balance!

```
#MONEY (-0.01),  
#MONEY ( 0),  
#MONEY ( 0.01),  
#MONEY ( 0.17),  
#MONEY ( 0.99),  
#MONEY ( 1.00),  
#MONEY ( 1.01),  
#MONEY (52.30),  
#MONEY (21474836.47),  
#MONEY (21474836.48),  
#MONEY (2147483646.99),  
#MONEY (2147483647.99),  
#MONEY (2147483648.99)
```

**Figure 6.6:** Test data for the class MONEY.

ysis testing strategy, which we described in Section 2.2. This strategy uses values at, immediately above and immediately below the range limits, as well as a typical value. The set of data used for testing the ACCOUNT class is shown in Figure 6.6. The syntax in the figure is that for creating an object with each of the required values (in Sather #MONEY is a constructor for the class MONEY, see the Appendix, Section B.1.2). STAT leaves it up to the user to select a suitable set of input data values.

In the generated methods of the test class, the guard conditions are checked in an *if*-statement before calling the method corresponding to the event. If there is more than one guard in an event sequence then the checks are nested. STAT assumes each guard in the state-transition diagram contains a boolean-valued Sather expression, and that any identifiers either match one of the event's arguments or are methods of the class under test. If the latter it is converted to a call on the object under test. Figure 6.7 shows the code generated for the last (11th) event sequence in Figure 6.3.

Methods that check guard conditions return the value true if all the guard conditions evaluate to true, and false otherwise. This is used to check that the event sequence in this method was achieved during execution of the test cases, as can be seen in the example in Figure 6.7. This example also show the handling of an event arguments, which become arguments to the method.

The main method runs event sequence methods with all combinations of input data for the types of its arguments. If none of the combinations exercise the transition it prints a warning. This should be considered a failure of the test data set. Where the guard conditions check event arguments this usually requires adding values to the input data set. The user can use the guard conditions to reason backwards about boundaries in the input domains that have been missed. Where the guard conditions depend only upon the internal state of the class under test, then this is a quasimodal class (see Section 5.1). Quasimodal classes should be validated using other techniques (Binder 1996).

```

event_seq_011(amt1, amt2: MONEY): BOOL is
  test_label:= "event seq. 011: open, debit("+amt1+")[balance - "
    +amt1+" < 0], credit("+amt2+")[balance + "+amt2+" >= 0]";
  ob:= #ACCOUNT;
  ob.open;
  if (ob.balance - amt1 < 0) then
    ob.debit(amt1);
    if (ob.balance + amt2 >= 0) then
      ob.credit(amt2);
      test(test_label, true.str, ob.in_state("InBalance").str);
      return true;
    else return false;
  end;
  else return false;
end;
end;

```

Figure 6.7: Code generated for event sequence no. 11 in Figure 6.3.

## 6.3 Experience To Date

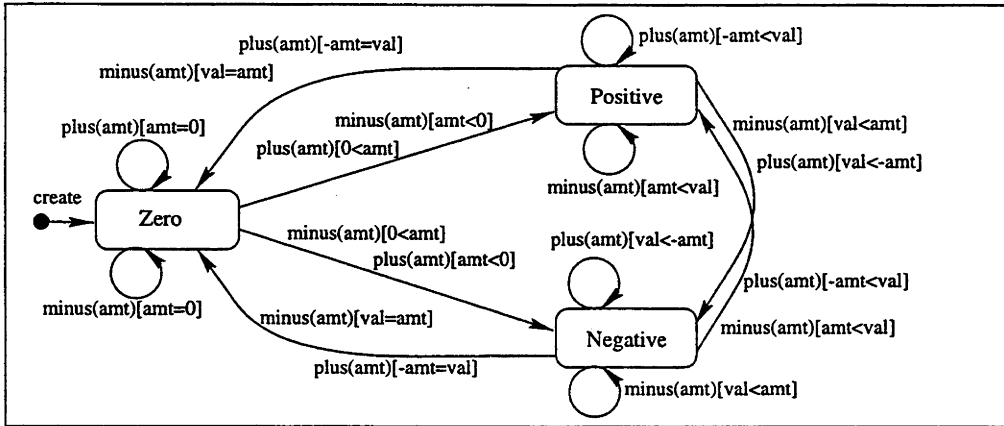
In Section 5.1, we described Binder's (1996) classification of four kinds of classes for the purpose of selecting an appropriate test strategy. The suggested test strategies are based on different combinations of domain testing and transition trees according to the kind of class. STAT also combines these two strategies for testing classes, but it generates the test cases. Thus the user need only specify the behaviour of the class in a state-transition diagram and a set of input values. In effect, STAT allows a tester to concentrate more effort on test design rather than the implementation of test cases.

STAT has been used to generate test cases for examples of three of the four kinds of classes in Binder's (1996) classification. The ACCOUNT example in Figure 6.1 is an example of Binder's "modal" class. A "unimodal" class such as TRAFFIC\_SIGNAL from Section 5.1, is simpler because there are no event arguments. Binder's testing strategy for the other two kinds, "quasimodal" and "nonmodal," does not use transition trees.

Nevertheless, STAT has been used to generate test cases for nonmodal classes. To generate test cases for the class TIME, which performs arithmetic with hours, minutes and seconds, we created the state-transition diagram in Figure 6.8, and a boundary-value analysis testing strategy was used to develop the input data in Figure 6.9.

An important consideration is the tool's ease of use in evolutionary development. For example, in a later iteration of the development of a system, an over-draft facility could be added to the ACCOUNT class. This would make it possible to debit an overdrawn account provided the balance remained above a specified limit. The state-transition diagram used to test this class is shown in Figure 6.10.

According to Perry & Kaiser (1990), not only does the new set\_limit method need to be tested, but both the methods credit and debit need to be retested, since they



**Figure 6.8:** A state-transition diagram for testing the class TIME.

```
#TIME( 0, 0, -1),
#TIME( 0, 0, 0),
#TIME( 0, 0, 1),
#TIME( 0, 0, 59),
#TIME( 0, 1, 0),
#TIME( 0, 1, 1),
#TIME( 0, 59, 59),
#TIME( 0, 0, 3600),
#TIME( 1, 0, 1),
#TIME( 0, -59, -59),
#TIME(-1, 0, 0),
#TIME(-1, 0, -1),
#TIME(52, 30, 22),
#TIME(2147483647, 59, 59),
#TIME(-2147483647, 59, 59)
```

**Figure 6.9:** Test data for the class TIME.

would have to be reimplemented to handle the over-draft limit, (see Section 3.4). Note that the `OVERDRAFT_ACCOUNT` class has behaviour that differentiates it from `ACCOUNT` (except when the over-draft limit is \$0) in the sense that the same sequence of method calls would produce different results in objects of each class. So although the interface of `OVERDRAFT_ACCOUNT` conforms to that of `ACCOUNT` it is not substitutable for it.

During software design, state-transition diagrams along with all other design documentation can be incomplete: “obvious” information is left out in order to avoid cluttering the diagram, and they can quickly become out of date. This can be a problem when using state-transition diagrams to generate test cases. For example in testing the `ACCOUNT` class using the state-transition diagram in Figure 6.1, the domain testing strategy led to the inclusion of a negative value in the set of input data values (see



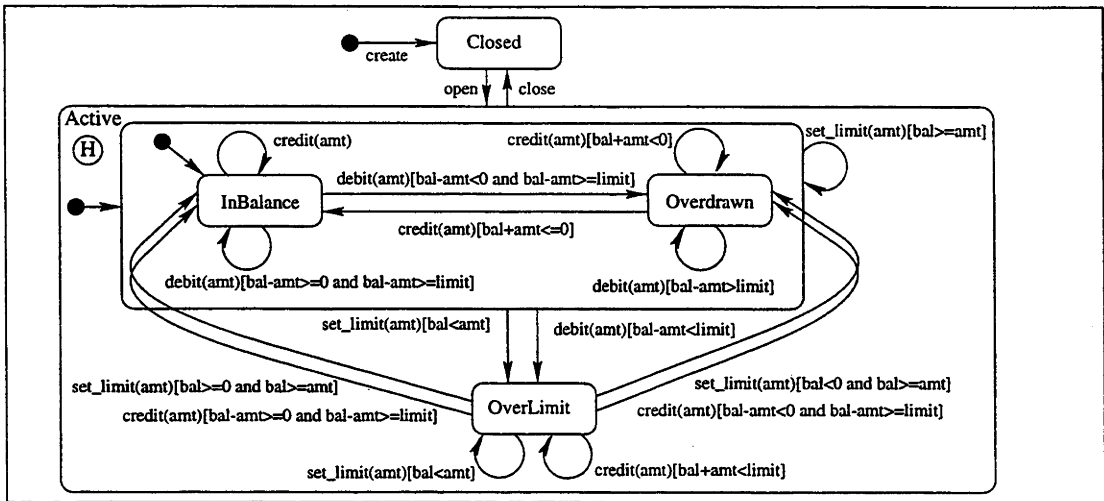


Figure 6.10: A state-transition diagram for an OVERDRAFT\_ACCOUNT object.

Figure 6.6). But a negative argument to the `credit` or `debit` methods was “obviously” an error and an exception should be raised. In executing the generated test cases there were a number of failures when the object under test raised an exception instead of changing state as apparently indicated in the state-transition diagram. Usually this problem is fixed by making the “obvious” details explicit in the diagram before generating the test cases. In this case, the way to correct the generated test suite was to include a non-negative condition in the guards of the affected transitions, that is, the transitions whose events are either `credit(amt)` or `debit(amt)` should have the condition `and amt>0` included in their guards.

## 6.4 Possible Extensions

In Chapter 5 we discussed the scope and limitations of our transition tree based approach. In particular the event sequences generated by a transition tree provide a minimal complete coverage of the state-transition diagram, that is, it produces the shortest sequences that reach all states. Occasionally this leads to inadequate testing of some transitions. A case in point occurs in the OVERDRAFT\_ACCOUNT example in Figure 6.10: the only event sequences that include `set_limit` are

```
create, open, set_limit(amt)[bal ≥ amt]           InBalance
create, open, debit(amt)[bal - amt < 0], set_limit(amt)[bal ≥ amt] Overdrawn
```

Most testers would like to include test cases that check the effect of `credit` and `debit` after a `set_limit`.

One way to generate longer sequences is to develop more elaborate state-transition diagrams, but this is not always easy. A future version of STAT could include the option to generate additional transition sequences that add a specified number of transitions to each of those in the transition tree.

Another reason for incorporating such an option is to enable the testing of “sneak paths”, discussed in Section 3.3.3. These are incorrect transitions in the implementation of the class under test that do not appear in the state-transition diagram. The suggested strategy to test for sneak paths is for each state to try events that are not permitted in that state. The main issue for sneak path testing in STAT is how should the correct handling of error conditions be represented? Error handling of illegal transitions may cause the class to enter an error state, raise an exception or simply ignore them. This information is sometimes represented in comments in the class specification, but is overlooked in most object-oriented design methods. A consistent error handling policy is necessary in order to automatically generate test cases for error situations.

It is possible with state-transition diagrams to describe the behaviour of a system of several interacting objects. However, STAT assumes that a single object is the receiver of all events. One strategy for testing such a system using STAT would be to add a test harness class to the system, for example using the Facade design pattern (Gamma et al. 1995). The state-transition diagram would have to be altered to route all events through the test facade class. The test facade class would also have to construct the system under test and perform any other necessary initialisations.

## 6.5 Summary

The goal of STAT is to simplify the generation of test cases in an iterative and incremental development environment typical of object-oriented software development. It makes use of information generated during object-oriented design and so is intended to be easily integrated into a CASE tool used to generate those design documents.

Our experience with STAT has suggested that it can also be used by testers as a tool to simplify test design and automatically generate a covering test suite. It reduces the effort in the mechanical and error prone activity of developing test cases and allows the user to concentrate on defining the behaviour of the class under test.

The applicability of STAT to testing object-oriented software is constrained by the applicability, expressiveness and use of state-transition diagrams. We have seen in Section 3.4.4 that the state-transition diagram for a class should apply to all subclasses, so the test suite generated by STAT can also test the subclasses, but this is true of any test suite for a class. The syntax used to specify test data for parameters to events, as in Figure 6.6, allows any subclass of the parameter type. One strategy which could be used to select data across multiple subtypes is described by McDaniel & McGregor (1994).

---

The future of a software component industry depends first on the wide acceptance of standard interfaces such as CORBA and JVM that provide the “sockets” for the components to plug into. But secondly, such an industry depends upon consumer trust in the reliability of those components. One way to build this trust is to use a standard component test interface such as the `STAT_TESTABLE` class. This would become a software component industry’s equivalent of the “test pins” in hardware IC’s.

---

## Conclusion

---

In this thesis we have explored the effect of object-oriented technology on software testing. We broke the discussion down into three broad themes: namely testing and object-oriented programming, testing and object-oriented design, and testing process issues in object-oriented software development. We began in Chapter 2 with an introduction to the established terminology and concepts of software testing.

In Chapter 3, we looked at a number of aspects of object-oriented programming and how they affected testing. These were encapsulation and object state, inheritance, exception handling, the use of assertions, and object interactions. It is possible to say that issues in testing classes: encapsulation, state, inheritance and polymorphism have been well covered in the research literature. What constitutes adequate testing for higher level constructs, including parameterised classes, clusters and subsystems of interacting classes, and frameworks, is still to be determined.

In Chapter 4, we considered the object-oriented software development process and the role of testing. The central issues to the discussion there were iteration and incremental delivery. We developed a framework that ensures test design should be carried out in parallel with other development activities. This chapter described an adaption of the Booch process (Booch 1994) that explicitly includes testing in its iterations.

The final two chapters focussed on testing a particular aspect of object-oriented design: object behaviour described in terms of state-transition diagrams. In Chapter 5, we presented the concept of a representation finite state machine to capture the relationship between an object's design states, as might be described in a state-transition diagram, and the implementation states, which are the possible values of its attributes.

In Chapter 6 we described the design of a tool, STAT, that makes use of the representation finite state machine concept to generate test cases from a state-transition diagram. This chapter also introduced an interface to access the representation finite state machine of a class. By combining the state-based testing approach of Chapter 5 with equivalence partitioning and boundary-value analysis on the arguments of a class's methods we were able to achieve good coverage of the behaviour of three of the four kinds of classes in Binder's (1996) classification of class behaviour.

In the remainder of this chapter we identify aspects of the interaction of testing and object technology that would benefit from further research. Once again, we dis-

cuss these under the three headings of object-oriented programming, object-oriented analysis and design, and testing in the development process.

## **7.1 Future directions**

### **7.1.1 Testing and Object-Oriented Programming**

Many of the issues to do with testing object-oriented programs have been dealt with in the research literature, as we have already seen. New languages and new technologies, however, will introduce their own specific problems to which test technology must respond. At the time of writing, some of the interesting new technologies are Java and middle-ware for distributed computing, in particular CORBA and COM. While new technologies make feasible new kinds of applications they also introduce new kinds of software faults. Software engineers will require new tools and techniques to either avoid or test for these faults.

One of Java's significant contributions to object-oriented programming is to popularise concurrency. While support for concurrency in object-oriented programming languages is by no means new, Java's wide adoption means that significant systems with strong quality requirements are being built in Java, using Java's concurrency mechanisms. Test techniques for concurrent programs are not well developed. Practical techniques are needed for identifying potential dead-locks, race conditions, resource sharing problems and other defects that can occur in concurrent programs.

### **7.1.2 Testing and Object-Oriented Analysis and Design**

The future of software test tools is as integrated components of a software development environment. CASE tools are a case in point: there are many more possibilities for extracting information for test design from CASE models, than the state-transition diagrams used by STAT. In particular, use cases, object interaction diagrams and message sequence diagrams are used to design the behaviour of object-oriented software and so have behavioural information which potentially could be used to generate test cases.

New object-oriented design methods are being developed to address emerging technologies, such as PARSE (Gorton et al. 1995) for concurrency and distributed objects, and Catalysis (D'Souza 1996) for Java. These offer the opportunity to develop test techniques that extract test requirements from design artifacts.

### **7.1.3 Testing and Object-Oriented Software Development**

Software is built for an enormous variety of applications, with a variety of quality requirements, and by a variety of development teams. No single software develop-

ment process is suitable for all these situations. The testing required for a fly-by-wire avionics system in a commercial aircraft is substantially different from that which creates an animated picture in a web page, as are the teams which undertake the testing. It remains to tailor testing processes to specific development processes and environments. Some of the variables that will to be addressed in the design of software testing processes are

**scale** While inspections are a recommended practice for defect reduction in a large team, a lone tester, or a developer performing their own testing, requires other means for objectively assessing the quality of their product. Test techniques that are suitable for a software development project of one size and with one set of available resources can be unmanageable or inefficient on another.

**quality requirements** A risk-based approach to testing is needed that selects test techniques and directs effort with the aim of reducing the likelihood and consequences of software failures.

**emerging technologies** Applications built on specific technologies often require specific testing techniques. We have mentioned some these for Java and CORBA in the previous sections. Applications based on the World Wide Web require significant cross-platform testing and have a range of security issues, this will be particularly important for the various proposed e-commerce platforms.

---

## An Example of STAT test cases

---

This Appendix lists the test case code generated by the STAT tool for the ACCOUNT example described in Chapter 6. It also lists the complete input files used to generate the testcases.

The first code file below defines the state-transition diagram for ACCOUNT in Figure 6.1. STAT uses the grammar for state-transition diagrams developed by Lucas (1993).

```
# account.sm
#
# State machine model for Account
# Author: Oscar Bosman (CSIRO CMIS) <oscar@cmis.csiro.au>
# $Id$
#
# -----

StateChartName Account
Version $Id$

Initial          create  Closed balance := 0

Closed           open    (Opened H InBalance)
                  # Opened has History, initial state is InBalance

(Opened)         close   Closed

InBalance: Opened
  credit(amt)                    InBalance balance := balance + amt
  debit(amt) [balance - amt >= 0] InBalance balance := balance - amt
  debit(amt) [balance - amt < 0]  Overdrawn balance := balance - amt

Overdrawn: Opened
  credit(amt) [balance + amt >= 0] InBalance balance := balance + amt
  credit(amt) [balance + amt < 0]  Overdrawn balance := balance + amt
  debit(amt)  [balance - amt >= lim] Overdrawn balance := balance - amt
```

The next file is the Sather source code for the class ACCOUNT. STAT uses the signatures defined in this file to determine method calls on the class.

```
-- account.sa:
-- Author: Oscar Bosman (CSIRO CMIS) <oscar@cmis.csiro.au>
-- $Id$
--
-----
```

```
type $ACCOUNT is
```

```
    open;

    close;

    credit(amount: MONEY);

    debit(amount: MONEY);
```

```
end; -- class $ACCOUNT
-----
```

```
class ACCOUNT < $ACCOUNT, $STAT_TESTABLE is
```

```
    readonly attr balance: MONEY;
    private attr isOpen: BOOL;
```

```
    create: SAME is
        res: SAME := new;
        res.isOpen := false;
        res.balance := #MONEY(0);
        return res;
    end;
```

```
    open is
        isOpen := true;
    end;
```

```
    close is
        isOpen := false;
    end;
```

```
    credit(amount: MONEY) is
        if isOpen and amount >= 0 then
            balance := balance + amount;
        end;
    end;
```

```
    debit(amount: MONEY) is
        if isOpen and amount >= 0 and balance >= 0 then
            balance := balance - amount;
        end;
    end;
```

```
--    reset is
--        isOpen := false;
--        balance := #MONEY(0);
--    end;
```

```
    state: $STR is
        if isOpen then
            if (balance >= #MONEY(0)) then
                return "InBalance";
            else
                return "Overdrawn";
            end if;
        end if;
```



```

        end;
    else
        return "Closed";
    end;
end;
end;

end; -- class ACCOUNT
-----

```

The test data set in the next file is used by STAT to provide values for method calls. A set of values should be supplied for each class that appears as a parameter in a transition. If no values are supplied, STAT will use a single value that is created by the default constructor for the type. However this is usually not sufficient to adequately test the class.

```

-- money.td
-- Author: Oscar Bosman (CSIRO CMIS) <oscar@cmis.csiro.au>
-- $Id$
--
--
Type
  MONEY
values
  #MONEY(-0.01),
  #MONEY( 0),
  #MONEY( 0.01),
  #MONEY( 0.17),
  #MONEY( 0.99),
  #MONEY( 1.00),
  #MONEY( 1.01),
  #MONEY(52.30),
  #MONEY(21474836.47),
  #MONEY(21474836.48),
  #MONEY(2147483646.99),
  #MONEY(2147483647.99),
  #MONEY(2147483648.99)
end

```

The file below (test\_tree\_account.sa) is compilable Sather source code that has been generated by STAT from the state-transition specification (account.sm), the class definition (account.sa) and the test data set (money.td). Some of the lines have been edited to fit onto the page.

```

-- test_tree_account.sa:
--
-- Generated transition tree coverage for class ACCOUNT (account.sa) from
-- state-transition diagram (account.sm).
--
-----
class TEST_TREE_ACCOUNT is
  include STAT_TEST_SUPPORT;

  attr test_label: STR;

  --
  -- Test input data and initialisation for type MONEY
  --

```

---

```

attr money_type_test_data_list: ARRAYMONEY;

test_data_init is
  money_type_test_data_list := |
                                #MONEY(-0.01),
                                #MONEY( 0),
                                #MONEY( 0.01),
                                #MONEY( 0.17),
                                #MONEY( 0.99),
                                #MONEY( 1.00),
                                #MONEY( 1.01),
                                #MONEY(52.30),
                                #MONEY(21474836.47),
                                #MONEY(21474836.48),
                                #MONEY(2147483646.99),
                                #MONEY(2147483647.99),
                                #MONEY(2147483648.99)
                                |;
end;

--
-- Test case execution
--
main is
  --
  covered : BOOL; -- check for test case reach
  amt1: MONEY;    -- generated argument
  amt2: MONEY;    -- generated argument

  test_data_init;

  amt1_values: ARRAYMONEY := money_type_test_data_list.copy;
  amt2_values: ARRAYMONEY := money_type_test_data_list.copy;

  class_name("ACCOUNT");

  -- test cases
  event_seq_001;

  event_seq_002;

  event_seq_003;

  event_seq_004;

  loop amt1 := amt1_values.elt!;
    event_seq_005(amt1);
  end;

  covered := false;
  loop amt1 := amt1_values.elt!;
    covered := event_seq_006(amt1) or covered;
  end;
  if ~covered then warning(test_label,"No tests for event seq. 006."); end;

  covered := false;
  loop amt1 := amt1_values.elt!;
    covered := event_seq_007(amt1) or covered;
  end;
  if ~covered then warning(test_label,"No tests for event seq. 007."); end;

```

---

```
covered := false;
loop amt1 := amt1_values.elt!;
  covered := event_seq_008(amt1) or covered;
end;
if ~covered then warning(test_label, "No tests for event seq. 008."); end;

covered := false;
loop amt1 := amt1_values.elt!;
  covered := event_seq_009(amt1) or covered;
end;
if ~covered then warning(test_label, "No tests for event seq. 009."); end;

covered := false;
loop amt1 := amt1_values.elt!;
  loop amt2 := amt2_values.elt!;
    covered := event_seq_010(amt1, amt2) or covered;
  end;
end;
if ~covered then warning(test_label, "No tests for event seq. 010."); end;

covered := false;
loop amt1 := amt1_values.elt!;
  loop amt2 := amt2_values.elt!;
    covered := event_seq_011(amt1, amt2) or covered;
  end;
end;
if ~covered then warning(test_label, "No tests for event seq. 011."); end;

finish;
end;

--
-- Event sequences
--
event_seq_001 is
  test_label:= "start state";
  ob:= #ACCOUNT;
  test(test_label, ob.state, "Closed");
end;

event_seq_002 is
  test_label:= "event seq. 002: open";
  ob:= #ACCOUNT;
  ob.open;
  test(test_label, ob.state, "InBalance");
end;

event_seq_003 is
  test_label:= "event seq. 003: open, close";
  ob:= #ACCOUNT;
  ob.open;
  ob.close;
  test(test_label, ob.state, "Closed");
end;

event_seq_004 is
  test_label:= "event seq. 004: open, close, open";
  ob:= #ACCOUNT;
  ob.open;
  ob.close;
  ob.open;
  test(test_label, ob.state, "InBalance");
```

```

end;

event_seq_005(amt1: MONEY) is
  test_label:= "event seq. 005: open, credit("+amt1+)";
  ob:= #ACCOUNT;
  ob.open;
  ob.credit(amt1);
  test(test_label, ob.state, "InBalance");
end;

event_seq_006(amt1: MONEY): BOOL is
  test_label:= "event seq. 006: open, debit("+amt1+)[balance - "+amt1+" >= 0]";
  ob:= #ACCOUNT;
  ob.open;
  if (ob.balance - amt1 >= 0) then
    ob.debit(amt1);
    test(test_label, ob.state, "InBalance");
    return true;
  else
    return false;
  end;
end;

event_seq_007(amt1: MONEY): BOOL is
  test_label:= "event seq. 007: open, debit("+amt1+)[balance - "+amt1+" < 0]";
  ob:= #ACCOUNT;
  ob.open;
  if (ob.balance - amt1 < 0) then
    ob.debit(amt1);
    test(test_label, ob.state, "Overdrawn");
    return true;
  else
    return false;
  end;
end;

event_seq_008(amt1: MONEY): BOOL is
  test_label:= "event seq. 008: open, debit("+amt1+)[balance - "+amt1+
                                                    " < 0], close";
  ob:= #ACCOUNT;
  ob.open;
  if (ob.balance - amt1 < 0) then
    ob.debit(amt1);
    ob.close;
    test(test_label, ob.state, "Closed");
    return true;
  else
    return false;
  end;
end;

event_seq_009(amt1: MONEY): BOOL is
  test_label:= "event seq. 009: open, debit("+amt1+)[balance - "+amt1+
                                                    " < 0], close, open";
  ob:= #ACCOUNT;
  ob.open;
  if (ob.balance - amt1 < 0) then
    ob.debit(amt1);
    ob.close;
    ob.open;
    test(test_label, ob.state, "Overdrawn");
    return true;

```

```

else
    return false;
end;
end;

event_seq_010(amt1, amt2: MONEY): BOOL is
    test_label:= "event seq. 010: open, debit("+amt1+")[balance - "+amt1+
                " < 0], credit("+amt2+")[balance + "+amt2+" < 0]";
    ob:= #ACCOUNT;
    ob.open;
    if (ob.balance - amt1 < 0) then
        ob.debit(amt1);
        if (ob.balance + amt2 < 0) then
            ob.credit(amt2);
            test(test_label, ob.state, "Overdrawn");
            return true;
        else
            return false;
        end;
    else
        return false;
    end;
end;

event_seq_011(amt1, amt2: MONEY): BOOL is
    test_label:= "event seq. 011: open, debit("+amt1+")[balance - "+amt1+
                " < 0], credit("+amt2+")[balance + "+amt2+" >= 0]";
    ob:= #ACCOUNT;
    ob.open;
    if (ob.balance - amt1 < 0) then
        ob.debit(amt1);
        if (ob.balance + amt2 >= 0) then
            ob.credit(amt2);
            test(test_label, ob.state, "InBalance");
            return true;
        else
            return false;
        end;
    else
        return false;
    end;
end;

end; -- class TEST_TREE_ACCOUNT
-----

```

The file below (stat\_test\_support.sa) is included by the test class generated by STAT. It simply includes the Sather system class TEST and adds a method for printing warning messages.

```

-- stat_test_support.sa:
-- Author: Oscar Bosman (CSIRO CMIS) <oscar@cmis.csiro.au>
-- $Id$
--
-----
class STAT_TEST_SUPPORT is
    -- A collection of utility functions used by the generated test code

    include TEST; -- class_name($STR), test($STR,$STR,$STR) and finish

```

```
warning(label,msg:$STR) is
  #ERR + "Warning: " + label + ": " + msg + "
n";
  end;

end; -- class STAT_TEST_SUPPORT
-----
```

---

# Overview of Sather

---

Sather is a statically typed object-oriented programming language that emphasises correctness and runtime efficiency. It supports statically checked strong typing, multiple inheritance, separate implementation and type inheritance, garbage collection, iteration abstraction, higher order functions, assertions, preconditions, postconditions and class invariants, all in a small, clean, orthogonal syntax. It is also free. This appendix gives an overview of its syntax and major features.

More information, including the free compiler, can be found at the Sather home page: <http://www.icsi.berkeley.edu/~sather/>. There is also an Internet news group devoted to Sather: `comp.lang.sather`.

## B.0.1 A little background and history

Sather is being developed at the International Computer Science Institute, in the University of California at Berkeley. The original version 0.2 was released in June 1991. It was originally intended as a smaller, more efficient Eiffel, but the two languages have since evolved in different directions. Version 0.5 (also known as “Canberra Sather”) was the work of a few people at CSIRO-DIT, and was released in late 1993. At the time of writing, the current version is 1.0. Sather is named after the Sather tower, a building on the campus of the University of California at Berkeley.

Version 0.1 was modified at Karlsruhe University, Germany, where it is used for teaching and research. This version is known as Sather-K and has an extensive algorithm library, Karla. Further information can be accessed from its own home page: <http://i44www.info.uni-karlsruhe.de/~frick/SatherK>.

## B.1 A brief overview of Sather syntax

### B.1.1 Classes

Sather programs consist of a list of class declarations. There are essentially two kinds of classes: abstract classes and concrete classes. Class names in Sather are written in

```

type $STACK{T} < $ELT{T} is
  -- An abstract stack

  push(elt: T);
  -- Push elt to the top of the stack

  pop: T;
  -- Return and remove the topmost element

  top: T;
  -- Return the topmost element

  size: INT;
  -- Number of elements in the stack

  is_empty: BOOL;
  -- True if size = 0

  elt!: T;
  -- Yield elements in FIFO order.

end; ---- type $STACK{T}

```

**Figure B.1:** An abstract class

all uppercase characters.

Abstract classes declare interfaces. They are introduced by the keyword `type` and their names must begin with a dollar sign (`$`). The abstract class `$STACK` in Figure B.1 is an example. Notice that it only lists the class's features (its signature); there are no implementations. An abstract class specifies exactly what we need to know in order to (syntactically correctly) access objects of that type.

The “<” after the class's name indicates subtyping (specification inheritance). In Figure B.1, the abstract type `$STACK{T}` is a subtype of `$ELT{T}`, which means that it has at least all the features of `$ELT{T}`. Thus any object that is a subtype of `$STACK{T}` can handle any message that is allowed for a `$ELT{T}`. (In the Sather libraries, the abstract class `$ELT{T}` merely declares the feature `elt! : T`, so the appearance of that feature in `$STACK{T}` is redundant). If consecutive dashes appear on a line (“--”), the rest of the line is a comment. Sather also has a supertype operator “>” which declares that an abstract class is a parent of the listed classes.

Concrete classes declare an implementation and are introduced by the keyword `class`. An example appears in Figure B.2, below. Concrete classes can be instantiated, that is we can create objects from them. Abstract classes of course cannot be instantiated.

`STACK{T}`, `$STACK{T}` and `$ELT{T}` are parameterised classes. They are similar to templates in C++ and Eiffel's generic classes. The formal parameter types are listed in braces after the class name. In order to use a parameterised class, we must replace the arguments with real class names, for example `STACK{INT}` or



```

class STACK{T} < $STACK{T} is
  -- An array-based stack implemented by delegation to an FLISTT,
  -- which allocates space by amortized doubling.

  private attr s: FLIST{T};

  create: SAME is
    res ::= new;
    res.s := #FLIST{T};
    return(res);
  end;

  create(n: INT): SAME is
    -- Preallocate n elements
    res ::= new;
    res.s := #FLIST{T}(n);
    return(res);
  end;

  push(e: T) pre ~void(self) is
    s := s.push(e);
  end;

  pop: T pre ~void(self) and ~is_empty is
    return(s.pop);
  end;

  top: T pre ~void(self) and ~is_empty is
    return(s.top);
  end;

  elt!: T pre ~void(self) is
    loop yield(s.elt!) end;
  end;

  size: INT pre ~void(self) is return(s.size) end;

  is_empty: BOOL pre ~void(self) is return(size = 0) end;

  contains(e: T): BOOL pre ~void(self) is
    return(s.contains(e));
  end;

  str: STR is
    return s.str;
  end;

end;

```

Figure B.2: A concrete class

TEMPERATURE\_SENSOR}.

Parameter types can be constrained, for example in

```
class SORTED_LIST{T < $IS_LT} is ...end;
```

Here element classes must be subtypes of \$IS\_LT. The abstract class \$IS\_LT declares

```

class STR < $IS_EQ{STR}, $IS_LT{STR}, $HASH, $STR, $ELT{CHAR} is
  -- Strings.
  -- Strings are represented as arrays of characters. Every character
  -- is significant.
  --
  -- References: Gonnet and Baeza-Yates, "Handbook of Algorithms and
  -- Data Structures", Addison Wesley, 1991.

  private include AREF{CHAR} aget->aget;
  -- Make modification routines private.

  ... other features specific to STR.

```

**Figure B.3:** Sather string class STR reuses the implementation of array of char.

a single feature which is the “is less than” operator. The “< \$IS\_LT” qualifier ensures that we will only create sorted lists of objects that can be compared. It also means that when we write the implementation of SORTED\_LIST, we can safely assume that its elements will understand the < operator.

A concrete class can reuse the implementation of another with an include clause. For example, strings behave a lot like arrays of characters, so that class was reused to implement Sather strings, as in Figure B.3. It is as though all the features in the included class are textually copied into the class in place of the include clause. If we don’t want some of the included features, we can rename them like this:

```
oldname -> newname
```

or remove them like this:

```
oldname ->
```

In Figure B.3 all the included routines are made private (by putting that keyword before the include clause) and then aget is renamed to make it public.

The concepts of subtyping and code inclusion are combined in nearly every other object-oriented programming language, although you can achieve the effect of code inclusion in C++ with private inheritance. Java’s implements is pure subtyping, but extends also combines subtyping with code reuse. Inheritance in Sather is discussed in Section B.4.1.

Two more special kinds of classes need to be mentioned: value classes and external classes. By default Sather objects are created on the program’s heap and accessed by reference semantics, i.e. a variable is a “pointer” to an object, more than one variable may refer to the same object (alias) and assignment only changes the pointer, so the old object still exists. When an object has no references to it, it is deleted by the garbage collector. If we precede the class declaration with the keyword value object created from it will have value semantics, like integers and chars in most languages. That

means a variable “contains” a value object and assignment means overwriting the old value with a new one, so aliasing is not possible.

External classes are used to provide access to code in other programming languages. A feature declared in an external class is either the name of a routine written in an other language that can be called from Sather code, or a Sather routine that can be called from another language.

## B.1.2 Features

Sather’s classes are nothing more than a list of their features. There are two kinds of features, attributes and routines (for those familiar with C++, these correspond to “data members” and “member functions”). Features can be made *private*, in which case they are not visible outside the class.

### Attributes

Attributes are introduced by the keyword *attr*. Space is allocated for them in an object. An attribute declaration implicitly defines two routines: an accessor and a modifier. So the attribute declaration `attr s : FLIST{T};` in Figure B.2, means the class has the routines `s : FLIST{T}` (accessor for `s`) and `s (x : FLIST{T})` (modifier for `s`). The compiler translates an assignment to an object attribute to a call on the latter method, thus `stack.s := new_s` becomes `stack.s (new_s)`. Notice that it is not possible to tell from the from a message whether it is implemented in a particular class by an attribute or a routine. In fact, given an abstract class with either of the above two signatures, it would be possible to implement it using an attribute in one subtype and a routine in another.

Attributes can be made immutable with the *const* keyword, in effect they have no modifier routine. The *readonly* keyword is the same as making the modifier routine *private*, so the attribute can not be changed outside the class. An attribute preceded by the keyword *shared* is class allocated, i.e. all objects of that class access the same value.

### Routines

The features `create`, `push`, `pop`, `top`, etc. in Figure B.2 are all examples of routines. Notice the two `create` routines: Sather allows overloading on method names. Routines with the same name are disambiguated by the number and type of arguments, and return type.

The `create` feature is called to instantiate objects of that class. The expression `#A_CLASS (arg)` is shorthand for `A_CLASS :: create (arg)`

Sather is garbage collected, so there is never any need to explicitly destroy objects and memory is automatically recovered from objects that are not referenced. It is

```

protect
...
    raise #ERROR_CLASS;

when ERROR_CLASS then ...
when $ERROR then ...
else
end;

```

**Figure B.4:** Exception handling

possible to ensure specific tasks, such as closing files, are performed when objects are recovered (this is known as *finalisation*). Objects can be manually deallocated, if needed.

In Sather, declarations have a Pascal-like syntax, which shows its heritage from Eiffel. The special type **SAME** is replaced by the name of the enclosing class. **SAME** is useful in code that will be reused by including it in another class. **SAME** can not appear in an abstract class.

Most other kinds of statements will no doubt be familiar. Branching is by the usual

```
if ...then ...else ...end
```

construct, multi-way branching by

```
case <variable> when <expression> then ...else ...end
```

There is only one way to loop `loop ...end`. The reason for this will become clear when we discuss Sather's `iters` in Section B.4.4.

Sather supports exceptions based on the common "throw and catch" model. An exception is "thrown" by a `raise` statement. If an exception is raised in a block of statements preceded by the `protect` keyword, as in Figure B.4, it will be caught in a following `when` clause if the exception's type conforms to the type expression in that `when` clause. The `else` clause catches all exceptions not caught by the `when` clauses.

Most of Sather's expressions will be obvious to those familiar with another object-oriented programming language. The few that perhaps need explanation are `new` which allocates space for an object, `self` refers to the current object (much as "this" does in C++), `void` refers to a non-existent object of the required type, and `void(a)` returns the boolean value "true" if `a` references `void`, or "false" otherwise.

```
class HELLO is
  main is
    #OUT + "Hello, world" + '\n';
  end; -- main
end; -- class HELLO
```

Figure B.5: A “hello world” program in Sather

## B.2 Compiling and running Sather programs

A routine whose name is `main` is treated specially. If a class containing such a method is specified when Sather compiler is run, the program it generates will begin executing by calling that routine. Figure B.5 shows a “hello world” program in Sather. The expression `#OUT` creates an object attached to the stream `std.out`. To print, we concatenate strings to the stream object.

If the code in Figure B.5 is in a file called `hello.sa` then the command

```
cs -main HELLO -o hello hello.sa
```

compiles it and generates an executable called `hello`. The flag `-main HELLO` tells the compiler the name of the class in which to find the main routine where execution starts.

## B.3 Programming environment

To a large extent a programming language is only as useful as its programming environment: the libraries, compiler and tools to support program development.

The Sather compiler provides a large range of options for runtime checking. Far fewer bugs get through the Sather compiler than a typical C++ program, and with checking enable many more are easily found. Garbage collection also removes possibility of the memory management bugs that plague C and C++.

An interpreter and debugger are being developed. The compiler generates C code that is sufficiently readable to be debugged with `gdb`. In my experience, there is little need to use a debugger because most bugs are found by the compiler or are readily located with the compiler’s runtime checking options.

Sather comes with class libraries covering basic data types, common data structures, file I/O and user interface (based on `tcl/tk`). There are an increasing number of contributed libraries. Sather is positioning itself as a language for scientific and numerical computing, so contributed libraries include classes for neural net simulation, image processing and numerical algorithms. We will soon see a collection of classes

interfacing to the BLAS libraries. Most of the examples in this appendix are taken from the Sather libraries.

Many of the features a developer would hope for can be found in the sather-mode for emacs. These include class browsing, template editing, interfacing with the compiler. There is a separate class browser that uses the tcl/tk-based user interface.

## B.4 Special features

This section describes a few of the unique features of Sather. These include the separation of subtyping and implementation inheritance, pre- and postconditions, type-safe down casting, iteration abstraction and higher-order functions.

### B.4.1 Inheritance in Sather

Subtyping (or as it is sometimes called, specification inheritance) and implementation inheritance are not the same thing (Leavens 1991). They are orthogonal concepts, but in nearly every object-oriented programming language they are combined in a single inheritance mechanism. In Sather they are distinct.

Subtyping is a modelling tool, it relates abstract data types. If class *A* is a subtype of *B*, then *A* can receive any message that *B* understands. Implementation inheritance on the other hand is about code reuse: a child class need not re-implement a method when the parent's can be reused.

To tie code reuse to subtyping introduces a number of problems, some of which we have discussed in Section 3.4. The Sather string class *STR* in Figure B.3 is a good example. To implement it, we regard it as an array of characters. But in object modelling terms, a string is no more a-kind-of array of characters than an integer is a-kind-of array of bits. There are times when we want to handle strings as arrays of characters, or integers as arrays of bits, so these classes provide the appropriate conversion methods.

Sather's type system enforces the rule that all super-classes are abstract. As well as making the type system simpler, there are many good data modelling reasons for this (Hürsch 1994). The classical "hierarchy of polygons" modelling problem serves to illustrate just one of them. If the classes *TRIANGLE* and *SQUARE* are subclasses of *POLYGON* then either *POLYGON* cannot have a *add\_vertex* method or the method has different semantics in *TRIANGLE* and *SQUARE*. In Sather, the classes *POLYGON*, *TRIANGLE* and *SQUARE* would be subtypes of *\$POLYGON*, and so *POLYGON* can correctly have a *add\_vertex* method.

```

class FSTR < $IS_EQ{FSTR}, $IS_LT{FSTR}, $HASH, $STR is
  -- Buffers for efficiently constructing strings by repeated
  -- concatenation using amortized doubling.
  ...
  substring(beg,num:INT):SAME
  -- The substring with 'num' characters whose first character
  -- has index 'beg'. Self may be void if beg=0 and num=0.
  pre num>=0 and beg.is_bet(0,size-num)
  post result.size = num
  is
    if void(self) then return void end;
    r::#SAME(num); r.acopy(0,num,beg,self); r.loc := num;
    return r
end;

```

Figure B.6: Pre- and postconditions in Sather

## B.4.2 Programming by contract

A routine's interface is a contract between its user and its implementor. The user is required to provide arguments of the specified type and the implementor to return a value of the specified return type. This is a syntactic constraint, but Bertrand Meyer has suggested that we can extend this idea to the routine's semantics (Meyer 1988, Meyer 1992a).

A *precondition* is a boolean expression that must be true when a routine is called, similarly a *postcondition* must be true when the routine returns. To abide by the contract, callers of the routine must ensure that the precondition is satisfied and, based on that assumption, the implementation must ensure the postcondition. The benefit for the implementor of the routine is that they don't need to handle problems like out-of-range inputs. The benefit for the user that abides by the precondition is that they can assume the result is valid.

Routines in Sather can have pre- and postconditions, introduced by the keywords *pre* and *post*. An example of their use can be found in Figure B.6. The precondition ensures that the routine's arguments are valid indices, and the postcondition that the returned string is the expected length.

Concrete classes may also have an *invariant*, a boolean valued function which at each method return checks that the object is in a valid state. Precondition, postcondition and invariant checking can be enabled or disabled with compiler switches.

A thorough implementation of the programming by contract paradigm requires that preconditions, postconditions and invariants are inherited. Further, in a subtype preconditions can only be weakened, and postconditions and invariants strengthened. In Sather since preconditions, postconditions and invariants can only be specified in concrete classes, and so cannot be inherited.

```

a: $OB; i: INT;
-- a is assigned an object retrieved from an object database.
typecase a
  when INT then i := a * 2;
  when FLT then i := a.floor;
  when STR then i := a.count('e');
  else
    raise "Unknown type"
end;

```

**Figure B.7:** An example of a typecase statement

```

a,b,c: VECTOR{FLT};
...
x: FLT;
loop
  x := sum!(a.elt! * b.elt!);
end;

```

**Figure B.8:** A dot product of vectors (showing the use of iters)

### B.4.3 Type safe down-casts

It is always safe to assign an object of one type to a variable whose type is a supertype. Occasionally however, we need to do the reverse: assign a supertype to a subtype. This is often the case when retrieving objects from some source external to the program, such as a persistent store. The problem is that to do so directly contravenes type safety.

Sather's answer is the `typecase` statement. In Figure B.7 the variable `a` has type `$OB`, but in a `then` clause of the `typecase` it has the same type as the type expression in the immediately preceding `when`. A `then` clause is only executed if the runtime type of the variable referenced by `a` conforms to its corresponding type expression, thus type safety is guaranteed.

### B.4.4 Iteration abstraction

In Figure B.8 we see an implementation of a vector dot product. This should be sufficiently terse to make even hardened C/C++ programmers feel happy. The methods with names ending in “!” are iterators (or in Sather-speak, “iters”). The iter `elt!` returns the elements of the vector in order on each pass through the loop, and causes the loop to exit when there are no more elements.

The real reason this code is terse is that Sather's iteration abstraction achieves a separation of concerns. We define in one place how a class's data structure is to be



```

class ARRAY{T} < $CONTAINER{T}, $COPY{SAME} is
  -- One-dimensional arrays of elements of type T, including sorting,
  ...
  is_sorted_by(lt:ROUT{T,T}:BOOL):BOOL is
    -- True if the elements of self are in sorted order using
    -- 't' to define "less than". Self may be void.
    if -void(self) then
      loop
        i:=1.upto!(asize-1);
        if lt.call([i],[i-1]) then return false end
      end
    end;
    return true
  end;
end;

```

**Figure B.9:** Using a bound routine in an “applicative” context

traversed, then we can simply use that mechanism where ever we need to retrieve the class’s elements. The user of the class is not exposed to its internal structure.

We have already an implementation of an iter in Figure B.2 (the feature `elt!`). In an iter, the statement “`yield <expression>;`” returns the current value of the expression and returns execution to the enclosing loop that the iter was called from. When called again, the iter resumes immediately after the yield statement. When the end of the iter is reached or a quit statement is encountered, execution resumes at the statement following the enclosing loop.

All classes define the iters `while! (x:BOOL)`, `until! (x:BOOL)` and `break!`, with the semantics you would expect.

Aside from the problems of exposing a class’s internal structure, looping over data structures is a source of “off-by-one” errors. Since iters capture the algorithm for traversing a class’s elements is in one place, these kinds of errors only need to be fixed once.

### B.4.5 Bound routines

Bound routines are similar to function pointers in C or C++ (or to closures in languages such as Lisp). However, Sather’s bound routines are type safe, that is their declaration and use must match in number and type of parameters and return type.

In Figure B.9 a bound routine is passed as an argument to a method of the library class `ARRAY`. The type declaration `ROUT{T, T}:BOOL` means that the bound routine must take two arguments of the array’s element type and return a boolean value (all bound routine declarations begin with “ROUT”).

To apply a bound routine, we use the keyword `call`. In the figure, the expression `lt.call([i],[i-1])` applies the passed bound routine to successive elements of the array. The expression `[i]` on its own is shorthand for `self[i]`.

```

class TEST_STACK is
  include TEST;

  main is
    class_name("STACKSTR");
    s := #STACK{STR};
    s.push("a");
    s.push("b");
    test("push", s.str, "a,b");
    test("size", s.size.str, 2.str);
    ...
    test("is_empty", s.is_empty.str, true.str);
    finish;
  end;
end;

```

**Figure B.10:** Part of a test class for the class STACK

To use the `is_sorted_by` method, we can create a bound routine like this

```
br: ROUT{INT,INT}:BOOL := #ROUT(_:INT.is_lt(_));
```

and then `br` can be passed as an argument. The underscores “\_” are place holders for arguments that will be filled in when the bound routine is called. When we come to apply the bound routine, the expression `br.call(x,y)` is the same as `(x).is_lt(y)`.

Apart from applicative programming such as in Figure B.9, bound routines also have applications in higher-order functions and “call-backs,” such as routines providing functionality for user interface objects.

## B.4.6 pSather

pSather is a parallel object-oriented programming language that has evolved with Sather. It is based on the SPMD (single program, multi data) model. Its additional features support threads, synchronisation, communication and placement of objects. It treats objects and processes as orthogonal concepts, as opposed to Actor type languages which combine them. There are implementations of pSather for the CM5 and networks of UNIX workstations. Further information on pSather can be found on the Sather home page.

## B.4.7 The test class idiom

In a thesis on testing, it would be remiss to fail to mention Sather’s test class idiom. This is not so much of a feature of Sather as it is a part of culture of Sather programmers (which, I believe, is as it should be).

---

A test class has a `main` routine which runs a set of test cases for a class. It will typically have a name of the form `TEST_C` if it has the test cases for the class `C`. An small example, part of a test class for the library class `STACK`, appears in Figure B.10. When done thoroughly, a test class can be larger than the class it tests. The methods `class_name`, `test` and `finish` are provided by the included library class `TEST`. Each test case has a label, and compares an actual result with the expected result. The `finish` method accumulates and reports the test results.

Test classes are kept with the classes they test, usually in the same file, or at least the same directory. That way the test cases can be conveniently updated as the class evolves. Test classes make regression testing automatic because they become part of the “edit, compile, run” development cycle.



---

# Bibliography

---

- Arnold, K. & Gosling, J. (1996), *The Java Programming Language*, Addison-Wesley Publishing Co., USA.
- Balcer, M. J., Hasling, W. M. & Ostrand, T. J. (1989), Automatic generation of test scripts from formal test specifications, in R. A. Kemmerer, ed., 'Proceedings of the ACM SIGSOFT 3rd Symposium on Software Testing Analysis and Validation, Key West, Florida', ACM SIGSOFT, pp. 210–218. (published as ACM SIGSOFT Software Engineering Notes 14(8), December 1989).
- Barbey, S., Buchs, D. & Péraire, C. (1996), Theory of specification-based testing for object-oriented software, in 'Proceedings of the European Dependable Computing Conference, EDCC2', LNCS 1150, Springer Verlag, Taormina, Italy, pp. 303–320.
- Beizer, B. (1984), *Software System Testing and Quality Assurance*, Van Nostrand Reinhold.
- Beizer, B. (1990), *Software Testing Techniques*, 2nd edn, Van Nostrand Reinhold.
- Berard, E. V. (1993a), *Essays on Object-Oriented Software Engineering Volume I*, Prentice Hall.
- Berard, E. V. (1993b), Testing object-oriented software, in 'Proceedings of the Object World '93 – Australia'.
- Bernhard, P. J. (1994), 'A reduced test suite for protocol conformance testing', *ACM Trans. Software Engineering and Methodology* 3(3), 201–220.
- Binder, R. V. (1995), 'The FREE approach to testing object-oriented software: An overview', Available from the the author's WWW site, URL: <http://www.rbsc.com>.
- Binder, R. V. (1996), 'Modal testing strategies for OO software', *IEEE Computer* 29(11), 97–99.
- Boehm, B. W. (1988), 'A spiral model of software development and enhancement', *IEEE Computer* 12(5), 61–72.
- Booch, G. (1986), *Software Engineering with Ada*, 2nd edn, Benjamin Cummings.

- Booch, G. (1991), *Object-Oriented Design with Applications*, Benjamin/Cummings.
- Booch, G. (1994), *Object-Oriented Analysis and Design with Applications*, 2nd edn, Benjamin/Cummings.
- Booch, G. (1996), *Object Solutions*, Benjamin/Cummings.
- Bosman, O. (1996), Testing and iterative development: Adapting the Booch method, in 'Proceedings of the 13th International Conference on Testing Computer Software', Washington, D.C., USA, pp. 1–9.
- Bosman, O. & Schmidt, H. W. (1995a), Object test coverage using finite state machines, in Mingins et al. (1995), pp. 171–178.
- Bosman, O. & Schmidt, H. W. (1995b), Object test coverage using finite state machines, Technical Report TR95-15, Dept. Software Development, Monash University, Melbourne, Australia.
- Chambers, C. (1993), Predicate classes, in O. M. Nierstrasz, ed., 'Proceedings of the European Conference on Object-Oriented Programming: ECOOP'93', LNCS 707, Springer-Verlag, Kaiserslautern, Germany, pp. 268–296.
- Chao, B. P. & Smith, D. M. (1993), 'Applying software testing practices to an object-oriented software development', *OOPS Messenger* 5(3), 49–52. (Addendum to the proceedings of OOPSLA'93).
- Cheatham, T. J. & Mellinger, L. (1990), Testing object-oriented software systems, in 'Proceedings of the 1990 ACM Eighteenth Annual Computer Science Conference', ACM, pp. 161–165.
- Chow, T. S. (1978), 'Testing software design modeled by finite-state machines', *IEEE Trans. Software Engineering* 4(3), 178–187.
- Coleman, D., Hayes, F. & Bear, S. (1992), 'Introducing objectcharts or how to use statecharts in object-oriented design', *IEEE Trans. Software Engineering* 18(1), 9–18.
- Dahl, O. J., Dijkstra, E. W. & Hoare, C. A. R. (1972), *Structured Programming*, Academic Press.
- DeLano, D. E. & Rising, L. (1996), 'System test pattern language', presented at Pattern Languages of Program Design, PLoP'96, URL: <http://www.cs.wustl.edu/schmidt/PLoP-96/index.html>.
- DeMillo, R. A. & Offutt, A. J. (1993), 'Experimental results from an automatic test case generator', *ACM Trans. Software Engineering and Methodology* 2(2), 109–127.

- 
- Doong, R.-K. & Frankl, P. G. (1993), The ASTOOT approach to testing object-oriented programs, Technical Report PUCS-104-93, Polytechnic University, 333 Jay St., Brooklyn NY 11201.
- D'Souza, D. (1996), 'Advanced modeling and design for java systems using catalysis', Tutorial notes, the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Toronto, Canada. Also available from the author's web site, URL: <http://www.iconcomp.com/catalysis>.
- D'Souza, R. J. & LeBlanc, Jr., R. J. (1994), 'Class testing by examining pointers', *Journal of Object-Oriented Programming* 7(4), 33–39.
- DTI (1987), 'The STARTS guide', Department of Trade and Industry, UK, avail. from the National Computing Centre, Manchester, UK. Reprinted in (Thayer & McGettrick 1993).
- Duke, R. (1994), Do formal object-oriented methods have a future?, in Mingins & Meyer (1994), pp. 273–280.
- Duncan, I. M. M. (1993), Strong Mutation Testing Strategies, PhD thesis, Computer Science Division, School of Engineering and Computer Science, University of Durham, Durham UK.
- Fiedler, S. P. (1989), 'Object-oriented unit testing', *Hewlett-Packard Journal* 40(2), 69–74.
- Firesmith, D. G. (1996), 'Pattern language for testing object-oriented software', *Object Magazine* 5(9), 32–38.
- Frakes, W. B., Lubinsky, D. J. & Neal, D. N. (1991), 'Experimental evaluation of a test coverage analyser for C and C++', *J. Systems and Software* 16(2), 135–139.
- Frankl, P. G. & Weyuker, E. J. (1988), 'An applicable family of data flow testing criteria', *IEEE Trans. Software Engineering* 14(10), 1483–1498.
- Frick, A., Zimmer, W. & Zimmermann, W. (1994), On the design of reliable libraries, in R. Ege, M. Singh & B. Meyer, eds, 'Proceedings of Technology of Object-Oriented Languages and Systems: TOOLS 17', Prentice Hall, Santa Barbara, CA, USA, pp. 13–23. Also available from the author's web site, <http://i44s11.info.uni-karlsruhe.de/pub/papers/frick/tools95.ps.gz>.
- Fujiwara, S., v. Bochman, G., Khendek, F., Amalou, M. & Ghedamsi, A. (1991), 'Test selection based on finite state models', *IEEE Trans. Software Engineering* 17(6), 591–603.

- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.
- Gannon, J., McMullin, P. & Hamlet, R. (1981), 'Data-abstraction implementation specification and testing', *ACM Trans. Programming Languages and Systems* 3(3), 211–223.
- Ghezzi, C., Jazayeri, M. & Mandrioli, D. (1991), *Fundamentals of Software Engineering*, Prentice Hall International.
- Gilb, T. (1988), *Principles of Software Engineering Management*, Addison-Wesley.
- Gorton, I., Gray, J. P. & Jelly, I. (1995), 'Object-based modelling of parallel programs', *IEEE Parallel and Distributed Technology* 3(2), 52–63.
- Harel, D. (1987), 'Statecharts: A visual formalism for complex systems', *Science of Computer Programming* 8, 231–274.
- Harrold, M. J., McGregor, J. D. & Fitzpatrick, K. J. (1992), Incremental testing of object-oriented class structures, in 'Proceedings of the 14th International Conference on Software Engineering', ACM Press, Melbourne, Australia, pp. 68–80.
- Harrold, M. J. & Soffa, M. L. (1991), 'Selecting and using data for integration testing', *IEEE Software* 8(2), 58–65.
- Henderson-Sellers, B. & Edwards, J. M. (1990), 'The object-oriented systems life cycle', *Comm. ACM* 33(9), 143–159.
- Henderson-Sellers, B. & Edwards, J. M. (1994), *BOOKTWO of Object-Oriented Knowledge: The Working Object*, Prentice Hall.
- Hetzel, B. (1988), *The Complete Guide to Software Testing*, 2nd edn, QED Information Sciences Inc.
- Hoffman, D. & Brealey, C. (1989), Module test case generation, in R. A. Kemmerer, ed., 'Proceedings of the ACM SIGSOFT 3rd Symposium on Software Testing Analysis and Validation, Key West, Florida', ACM SIGSOFT, pp. 97–102. (published as Software Engineering Notes 14(8), December 1989).
- Hoffman, D. & Strooper, P. (1995), 'The testgraph methodology: automated testing of collection classes', *Journal of Object-Oriented Programming* 8(7), 35–41.
- Hogg, J. (1991), Islands: Aliasing protection in object-oriented languages, in 'Proceedings of Object-Oriented Programming: Systems, Languages and Applications, OOPSLA'91', ACM, ACM Press, Phoenix, AZ, USA, pp. 271–285.



- 
- Hogg, J., Lea, D., Wills, A., deChampeaux, D. & Holt, R. (1992), 'The geneva convention on the treatment of object aliasing', *OOPS Messenger* 3(2), 11–16.
- Hopcroft, J. E. & Ullman, J. D. (1979), *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley.
- Horgan, J. R., London, S. & Lyu, M. R. (1994), 'Achieving software quality with testing coverage measures', *IEEE Computer* 27(9), 60–69.
- Hunt, N. (1994), 'C++ boundary conditions and edge cases', *Journal of Object-Oriented Programming* 8(2), 25–29.
- Hürsch, W. L. (1994), 'Should superclasses be abstract', in M. Tokoro & R. Pareschi, eds, 'Proceedings of the European Conference on Object-Oriented Programming: ECOOP'94', LNCS 821, Springer-Verlag, Bologna, Italy, pp. 12–31.
- Hutchins, M., Foster, H., Goradia, T. & Ostrand, T. (1994), 'Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria', in 'Proceedings of the 16th International Conference of Software Engineering', IEEE Computer Society Press, Los Alamitos, CA, USA., pp. 191–200.
- Hutt, A. T. F. (1994), *Object Analysis and Design: Comparison of Methods*, Wiley QED.
- IEEE (1983), 'ANSI/IEEE Std. 829-1983: IEEE Standard for Software Test Documentation', IEEE Inc.
- IEEE (1990), 'ANSI/IEEE Std. 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology', IEEE Inc.
- Jacobson, I., Christerson, M., Jonsson, P. & Övergaard, G. (1992), *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley.
- Kaner, C., Falk, J. & Nguyen, H. Q. (1993), *Testing Computer Software*, 2nd edn, Van Nostrand Reinhold.
- Kiczales, G., des Rivières, J. & Bobrow, D. G. (1993), *The Art of the Metaobject Protocol*, The MIT Press.
- Kirani, S. & Tsai, W. T. (1994), 'Specification and verification of object-oriented programs', Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, USA.
- Knuth, D. E. (1992), *Literate Programming*, number 27 in 'CLSI Lecture Notes', Center for the Study of Language and Information, Leland Stanford Junior University, chapter The Errors of T<sub>E</sub>X(1989), pp. 243–291.

- Korel, B. (1990), 'Automated software test data generation', *IEEE Trans. Software Engineering* 16(8), 870–879.
- Lakos, J. S. (1992), 'Designing-in quality', Provided by the author (email: john.lakos@warren.mentorg.com).
- Landi, W. (1992), 'Undecidability of static analysis', *Letters on Programming Languages and Systems* 1(4).
- Leavens, G. (1991), 'Modular specification and verification of object-oriented programs', *IEEE Software* 8(4), 72–80.
- Levendel, Y. (1991), 'Improving quality with a manufacturing process', *IEEE Software* 8(2), 13–25.
- Leveson, N. G., Heimdahl, M. P. E., Hildreth, H. & Reese, J. D. (1994), 'Requirements specification for process-control systems', *IEEE Trans. Software Engineering* 20(9), 684–707.
- Lieberman, H. (1986), Using prototypical objects to implement shared behaviour in object-oriented systems, in 'Proceedings of Object-Oriented Programming: Systems, Languages and Applications, OOPSLA'86', ACM, ACM Press, Portland, OR, USA.
- Liskov, B. & Guttag, J. (1986), *Abstraction and Specification in Program Development*, The MIT Press.
- Lokan, C. J. (1993), 'The cleanroom process for software development', *Australian Computer J.* 25(4), 129–134.
- Lorenz, M. (1993), *Object-Oriented Software Development: A Practical Guide*, Prentice Hall.
- Love, T. (1993), *Object Lessons*, SIGS Books.
- Lucas, P. J. (1993), An object-oriented language system for implementing concurrent, heirarchical, finite state machines, Master's thesis, Graduate College of the University of Illinois at Urbana-Champaign.
- Macro, A. (1990), *Software Engineering: Concepts and Management*, Prentice Hall.
- Marick, B. (1992a), 'Testing software that reuses', Technical Note 2, Testing Foundations, Campaign, Il 61820, USA.
- Marick, B. (1992b), *A Tutorial Introduction to GCT*, Testing Foundations, 809 Balboa, Campaign, Illinois 61820, USA.

- 
- Marick, B. (1994), *The Craft of Software Testing*, Prentice Hall.
- Maurer, P. M. (1990), 'Generating test data with enhanced context-free grammars', *IEEE Software* 7(4), 50–55.
- McDaniel, R. & McGregor, J. D. (1994), Testing the polymorphic interactions between classes, Technical Report TR-94-103, Clemson University.
- McDermid, J. A., ed. (1991), *Software Engineer's Reference Book*, Butterworth-Heinemann Ltd.
- McDermid, J. A. & Rook, P. (1991), Software development process models, in McDermid (1991).
- McGregor, J. D. (1994), 'Testing object-oriented software', Tutorial notes, the European Conference on Object-Oriented Programming: ECOOP'94, Bologna, Italy.
- McGregor, J. D. & Dyer, D. M. (1993), 'A note on inheritance and state machines', *ACM SIGSOFT Software Engineering Notes* 18(4), 61–69.
- McGregor, J. D. & Kare, A. (1996), Parallel architecture for component testing of object-oriented software, in 'Proceedings of the 9th Annual Software Quality Week', Software Research, Inc., San Francisco.
- Meyer, B. (1988), *Object-Oriented Software Construction*, International Series in Computer Science, Prentice Hall International.
- Meyer, B. (1992a), 'Applying "design by contract"', *Comm. ACM* 35(10), 40–51.
- Meyer, B. (1992b), *Eiffel: The Language*, Prentice Hall.
- Meyer, B. (1994), Beyond design by contract: Putting more formality into object-oriented development, in Mingins & Meyer (1994).
- Mingins, C., Duke, R. & Meyer, B., eds (1995), *Proceedings of Technology of Object-Oriented Languages and Systems: TOOLS 18*, Prentice Hall, Melbourne, Australia.
- Mingins, C. & Meyer, B., eds (1994), *Proceedings of Technology of Object-Oriented Languages and Systems: TOOLS 15*, Prentice Hall, Melbourne, Australia.
- Murphy, G. C., Townsend, P. & Wong, P. S. (1994), 'Experiences with cluster and class testing', *Comm. ACM* 37(9), 39–47.
- Musa, J. D. & Ackerman, A. F. (1989), 'Quantifying software validation: When to stop testing?', *IEEE Software* 6(3), 19–30.

- Myers, G. J. (1976), *Software Reliability: Principles and Practices*, John Wiley & Sons.
- Myers, G. J. (1979), *The Art of Software Testing*, John Wiley & Sons.
- Ntafos, S. C. (1988), 'A comparison of some structural testing strategies', *IEEE Trans. Software Engineering* **14**(6), 868–874.
- Omohundro, S. M., Bilmes, J., Schmidt, H. W. & Bosman, O. (1993), *The Sather Language*. Provided with the public domain Sather compiler, version 0.5.6.
- Omohundro, S. M. & Soutamire, D. (1995), *The Sather 1.0 Specification*, International Computer Science Institute, University of California at Berkeley. Documentation provided with the Sather compiler, version 1.0, URL: <http://http.icsi.berkeley.edu/Sather>.
- Osmond, R. F. (1994), Components of success: Real life experiences with object technology, in Mingins & Meyer (1994), pp. 281–293.
- Ostrand, T. J. & Balcer, M. J. (1988), 'The category–partition method for specifying and generating functional tests', *Comm. ACM* **31**(6), 676–86.
- Paepcke, A., ed. (1993), *Object-Oriented Programming: The CLOS Perspective*, The MIT Press.
- Palsberg, J. & Schwartzbach, M. I. (1994), *Object-Oriented Type Systems*, John Wiley & Sons.
- Parnas, D. L. & Clements, P. C. (1986), 'A rational design process: How and why to fake it', *IEEE Trans. Software Engineering* **12**(2), 251–257.
- Perry, D. E. & Kaiser, G. E. (1990), 'Adequate testing and object-oriented programming', *Journal of Object-Oriented Programming* **2**, 13–19.
- Poston, R. M. (1994), 'Automated testing from object models', *Comm. ACM* **37**(9), 48–58.
- Pressman, R. S. (1992), *Software Engineering: A Practitioner's Approach*, 3rd edn, McGraw-Hill.
- Ramalingam, G. (1994), 'The undecidability of aliasing', *ACM Trans. Programming Languages and Systems* **16**(5), 1467–1471.
- Ratjens, M. & Steele, R. (1993), An introduction to Classworks: A systems development methodology, Technical report, Class Technology Pty. Ltd.
- Rettig, M. (1991), 'Testing made palatable', *Comm. ACM* **34**(5), 25–29.

- 
- Royce, W. W. (1970), Managing development of large software systems: Concepts and techniques, in 'Proceedings of IEEE WESCON', IEEE, pp. 1–9. also in *Proc. ICSE 9* Computer Society Press 1987, and Thayer, R. H. (ed.) *Software Engineering Project Management, IEEE Tutorial EH0263-4*, 1988.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorenzen, W. (1991), *Object-Oriented Modeling and Design*, Prentice Hall International.
- Schmidt, H. W. & Chen, J. (1995), Reasoning about concurrent objects, Technical Report TR-95-05, Department of Software Development, Monash University, Melbourne, Australia.
- Schmidt, H. W. & Zimmermann, W. (1994a), 'A complexity calculus for object-oriented programs', *Object-Oriented Systems J.* 1(2), 117–147.
- Schmidt, H. W. & Zimmermann, W. (1994b), Reasoning about complexity in object-oriented programs, in 'Proceedings of the International Conference Programming Concepts, Methods and Calculi, San Miniato, Italy'.
- Shlaer, S. & Mellor, S. J. (1988), *Object-Oriented Systems Analysis: Modelling the World in Data*, Prentice Hall.
- Shlaer, S. & Mellor, S. J. (1992), *Object Lifecycles: Modelling the World in States*, Prentice Hall.
- Smillie, J. & Strooper, P. (1995), Testing generic classes in the testgraph framework, in Mingins et al. (1995), pp. 147–158.
- Sommerville, I. (1992), *Software Engineering*, 4th edn, Addison-Wesley.
- Spruler, D. A. (1994), *C++ and C Debugging, Testing and Reliability*, Prentice Hall.
- Taylor, P. (1992), Experiences with object-oriented software development, in J. Potter & B. Meyer, eds, 'Proceedings of Technology of Object-Oriented Languages and Systems: TOOLS 9', Prentice Hall, Sydney, Australia, pp. 171–183.
- Thayer, R. H. & McGettrick, A. D., eds (1993), *Software Engineering: A European Perspective*, IEEE Computer Society Press.
- Turner, C. D. & Robson, D. J. (1992a), A suite of tools for the state-based testing of object-oriented programs, Technical Report TR 14/92, Computer Science Division, School of Engineering and Computer Science, University of Durham, Durham UK.
- Turner, C. D. & Robson, D. J. (1992b), The testing of object-oriented programs, Technical Report TR 13/92, Computer Science Division, School of Engineering and Computer Science, University of Durham, Durham UK.

- Turner, C. D. & Robson, D. J. (1993), State-based testing and inheritance, Technical Report TR 1/93, Computer Science Division, School of Engineering and Computer Science, University of Durham, Durham UK.
- Viswanadha, S. R. & Sankar, S. (1996), Preliminary design of ADL/C++- a specification language for C++, in 'Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)', The USENIX Association, Toronto, Canada, pp. 97-111.
- Vonnegut, Jr., K. (1972), *Slaughterhouse-Five*, Panther Books.
- Walsh, J. F. (1992), Preliminary defect data from the iterative development of a large C++ program (experience report), in 'Proceedings of Object-Oriented Programming: Systems, Languages and Applications, OOPSLA'92', ACM, ACM Press, Vancouver, BC, Canada, pp. 178-183.
- Wang, Y. & Parnas, D. L. (1994), 'Simulating the behaviour of software modules by trace rewriting', *IEEE Trans. Software Engineering* **20**(10), 750-759.
- Weyuker, E. J. (1986), 'Axiomatizing software test data adequacy', *IEEE Trans. Software Engineering* **12**(12), 1128-1138.
- Weyuker, E. J. (1988), 'The evaluation of program-based software test data adequacy criteria', *Comm. ACM* **31**(6), 668-75.
- White, I. (1994), *Using the Booch Method: A Rational Approach*, Benjamin/Cummings.
- Whittaker, J. A. & Thomason, M. G. (1994), 'A markov chain model for statistical software testing', *IEEE Trans. Software Engineering* **20**(10), 812-824.
- Wirfs-Brock, R., Wilkerson, B. & Wiener, L. (1990), *Designing Object-Oriented Software*, Prentice Hall.
- Wulf, W. A., London, R. L. & Shaw, M. (1976), Abstraction and verification in Alphard: Introduction to language and methodology, Technical report, USC Information Science, University of Southern California, Los Angeles, USA.
- Zucconi, L. & Reed, K. (1996), 'Building testable software', *ACM SIGSOFT Software Engineering Notes* **21**(5), 51-55.