Improving Software Designs via the Minimum Description Length Principle

Joseph Arthur Wood

492

January 1998

ISSN 1350–3162

Cognitive Science Research Papers

Gender

Male pronouns have been used in this thesis to refer to people of both sexes in order to smooth the flow of the text rather than imply any sexual bias.

Nomenclature

The word Ada without qualification refers to the Ada83 programming language, defined in Ichbiah et al. (1983).

HOOD, without qualification, is used to refer to HOOD version 3, defined in Delatte et al. (1993). All references to HOOD 4 (HOOD HRM, 1995) are explicit.

A number of words are used in the literature (e.g., function, procedure, operation, and routine) to refer to a similar concept. Frequently, each word has a slightly different meaning; for example, functions are often seen as procedures without side-effects. In this thesis we do not require these distinctions, and so all such words are equivalent. In general we shall use HOOD's term *operation*.

In this thesis, the word *object* is used to refer to a collection of co-operating items, whereas the word *module* is generally used to refer to the older concept of sub-programs, see Page-Jones (1992). Typically, a module is just a HOOD operation.

Trademark Acknowledgments

A number of trademarks are used in this thesis and for brevity are declared once here as follows but apply throughout the thesis:

Trademark	Trademark Owner
Ada	U.S. Department of Defense, Ada Joint Program Office.
ANSI	American National Standards Institute.
AT&T	AT&T.
HOOD	HOOD User Group.
POPLOG	University of Sussex.
PostScript	Adobe, Inc.
SADT	SofTech, Inc.
UNIX	AT&T.

All other trademarks are acknowledged.

Typographic Conventions

A few type definitions are given in Chapter 7, these are presented in VDM, see for example Casey (1994) or Dawes (1991). We have adopted the convention that type-names start with an uppercase letter, and record field names start with a lowercase letter.

Acknowledgements

This thesis is dedicated to the memory of my father, Arthur Wood (1905–1990). I would like to thank my mother for all the support and encouragement that she has given me.

As with all human endeavours the author owes a debt of gratitude to numerous people. In particular I should like to thank Dr. Luca Aceto, Dr. Theo Avantigis, Dr. Steve Easterbrook, Dr. Rüdi Lutz and Dr. Des Watson for acting as my Thesis Committee, and for all their invaluable advice and encouragement.

I wish to thank AT&T for permission to use their graph visualisation software (`dotty`, `dot` and `lefty`). I also wish to thank CISI Enginerie (especially Maurice Heitz), Centre National d'Etudes Spatiales (CNES) and the ESA for supplying a large sample of HOOD designs. My thanks also to CRI A/S for a copy of the RAISE Method Manual before its publication in England. I also wish to thank Rüdi Lutz and Robert Duncan for giving me access to a full version of Linux Poplog, this was a great help.

Numerous people have helped in this endeavour (in no particular order): Amer Al-Rawas, David Blackman, Joanna Brook, Frank Calliss, David Carlisle, Stuart Clark, Changiz Delara, Robert Duncan, Liz Eastwood, Alan Jeffrey, Stuart & Sharon Lythgoe, Robert Milne, Andrew Ormsby, David Perella, Peter Williams, Jeremy Withrington, Sarah Wood and Sean Young. I gratefully acknowledge all their kind help, support and advice.

I must also extend a warm thanks to the COGS support team, (in particular John Gibson, James Goodlet, Sharon Groves, Roger Sinnhuber and John Williams) who kept the system up and running, and answered all my strange questions.

I also wish to thank my contacts at BT, for helpful comments during my periods of industrial placement, especially Peter Utton, Betsy Cordingley, Jan Buknowski, Stephen Corley and Jeremy Wilson.

My special thanks must go to my supervisor Dr. Rüdi Lutz for all his advice, support, encouragement, for answering numerous questions, and for keeping this project on course.

The author gratefully acknowledges the financial support of the Engineering and Physical Sciences Research Council in association with British Telecom Laboratories.

Improving Software Designs via the Minimum Description Length Principle

Joseph Arthur Wood

Summary

This thesis studies the creation of good quality, modular software designs. It focuses on architectural design, that is, the identification of components, their purpose, and interactions. This view of software architecture naturally leads to representing a design's structure as a graph.

Complexity can be viewed along several dimensions including coupling and cohesion. Cohesion is the 'togetherness' of a software component, and coupling is the 'separateness' of software components. These two concepts have been studied for some time, and various definitions have been proposed. Indeed, both concepts have widely accepted ordinal measurement scales. However, a design must achieve a trade-off between these two concepts, and little has been written about how to achieve this trade-off. Complexity needs to be controlled in order to create understandable, maintainable and cost effective solutions.

This thesis investigates design complexity at the architectural level, by applying Kolmogorov complexity to the graph's structure. To measure complexity, we use the Minimum Description Length principle; such that when a proposed design's structure is represented by a message, the length of this message is taken as a measure of the design's complexity. We show that this metric satisfies all of Weyuker's complexity properties.

We propose that this model of complexity is an improvement over using disjoint and less precise measures of coupling and cohesion. Our metric captures information about the structure of a software design; it says nothing about ease of construction, maintenance or even satisfying the requirements. However, we suggest that *ceteris paribus* a simpler structure is preferred over a more complex structure, because it is cheaper and more reliable.

A prototype tool, *Morpheus*, is described which manipulates a design's architecture, to produce a shorter representation, we suggest that this new design is simpler than the original. We demonstrate its use with a medium sized example in HOOD notation.

Submitted for the degree of D. Phil.

University of Sussex

January 1998

Contents

I Background **1**

1 Introduction **3**
1.1 Purpose of the Research . . . 3
1.2 Motivation . . . 4
1.3 Complexity: the Central Problem . . . 4
1.4 Synopsis of the Research . . . 5
1.4.1 Assumptions . . . 5
1.4.2 Goals . . . 5
1.4.3 Anticipated Benefits . . . 5
1.5 Thesis Structure . . . 6
1.6 Outline of our Solution . . . 6

2 Software Design **8**
2.1 Design Theory . . . 8
2.1.1 What is Design? . . . 9
2.1.2 Values in Design . . . 9
2.1.3 Design Problems . . . 9
2.1.4 Design Forms . . . 10
2.1.5 Design Evolution . . . 11
2.1.6 Summary of Design Theory . . . 12
2.2 Psychological Aspects of Software Design . . . 12
2.3 Design Notations . . . 13
2.3.1 Natural Language . . . 14
2.3.2 Graphical Notations . . . 15
2.3.3 Formal Languages . . . 15
2.3.4 Choice of Language . . . 16
2.4 Design Methods . . . 16
2.4.1 Functional Decomposition . . . 16
2.4.2 Data Structured Design . . . 17
2.4.3 Object Oriented Design . . . 17
2.4.4 Formal Methods . . . 18
2.5 Complexity: The Scourge of Engineering . . . 19
2.6 Architectural Design . . . 20
2.6.1 What is an Object? . . . 20
2.6.2 Are these the Right Objects? . . . 20
2.6.3 What does Ψ Measure? . . . 23
2.7 Design as a Graph . . . 23

3 An Overview of HOOD **25**
3.1 Introduction . . . 25
3.1.1 What is HOOD? . . . 25
3.1.2 Where does HOOD sit in the Software Life-cycle? . . . 25
3.1.3 The HOOD Design Method . . . 26
3.2 Example: Controlling the Traffic Lights . . . 27

3.3 Objects - Architectural Components . . . 28
3.3.1 Traffic Lights - Graphical Notation . . . 28
3.4 HOOD Components . . . 29
3.4.1 Passive Objects . . . 29
3.4.2 Active Objects . . . 30
3.4.3 Operation Control Objects . . . 30
3.4.4 Environmental Objects . . . 30
3.4.5 Visibility . . . 30
3.5 HOOD Entities . . . 31
3.6 Textual Representation . . . 31
3.7 Unused HOOD Facilities . . . 36
3.8 Rationale for Choosing HOOD . . . 37
3.9 Augmented HOOD . . . 37
3.10 Further Reading . . . 37

4 Complexity Measures **38**
4.1 Requirements for a Complexity Measure . . . 39
4.2 Existing Complexity Metrics . . . 39
4.2.1 Program Complexity Metrics . . . 39
4.2.2 Design Metrics . . . 40
4.2.3 Object Oriented Design Metrics . . . 41
4.2.4 Information Theory and Design Metrics . . . 41
4.3 Combining Different Measures . . . 42
4.4 The Dual Problem: Reverse Engineering . . . 42
4.5 Towards a Merit function . . . 42
4.6 Validating Complexity Measures . . . 43

II Theory **45**

5 Mathematical Background **47**
5.1 Graph Theory . . . 47
5.1.1 Basic Terms . . . 47
5.1.2 Operations on Graphs . . . 49
5.1.3 Simple Design Graphs . . . 51
5.1.4 Hierarchical Graphs . . . 52
5.1.5 Full Design Graphs . . . 52
5.1.6 Design Graph Concatenation . . . 53
5.2 Information Theory . . . 53
5.2.1 Kolmogorov Complexity . . . 54
5.2.2 Minimum Description Length Principle . . . 55
5.2.3 Prefix Encoding of Positive Integers . . . 55
5.2.4 Length of Code Words with Known Probabilities . . . 57
5.2.5 Further Reading . . . 58

6 Describing a Graph **59**
6.1 The Message Passing Metaphor . . . 59
6.2 Ψ: The Complexity of a Design Graph . . . 59
6.2.1 Describing the Edges in a Graph . . . 59
6.2.2 Connections in a Single Object . . . 62
6.2.3 Extensions for Multiple Objects . . . 63
6.2.4 Further Extensions for HOOD . . . 65

6.3 A Complexity Measure? . . . 67
6.4 Theoretical Validation . . . 67
6.4.1 Weyuker's Properties . . . 67
6.5 Conclusion . . . 72

III *Morpheus* **73**

7 *Morpheus***: A Prototype System** **75**
7.1 Extensions to HOOD - Augmented HOOD . . . 75
7.2 Implementation . . . 76
7.2.1 Basic Structure . . . 76
7.2.2 Parser . . . 76
7.2.3 Data Analyser . . . 78
7.2.4 Improvement Engine . . . 82
7.3 Limitations . . . 85
7.3.1 Physical Resources . . . 85
7.3.2 Missing Information . . . 86

8 Empirical Evidence in Support of Ψ **87**
8.1 Initial Experiments . . . 87
8.1.1 Varying Group Size . . . 87
8.1.2 Moving Basic Entities . . . 92
8.1.3 Reducing Cohesion . . . 92
8.1.4 Increasing Coupling . . . 99
8.2 A Small Example: Traffic Lights . . . 106
8.2.1 Discussion on the Traffic Lights Design . . . 107
8.3 *TriviCalc* - A System to Design . . . 114
8.3.1 Improvement? . . . 115
8.3.2 Support for Future Changes? . . . 115

9 Summary and Conclusion **117**
9.1 Summary of this Thesis . . . 117
9.2 Evaluation . . . 118
9.2.1 Achievements . . . 118
9.2.2 Industrial Application . . . 119
9.3 Further Work . . . 119
9.4 Contribution of This Thesis . . . 120

Bibliography **121**

A Augmented HOOD **128**
A.1 Changes to Existing Syntax . . . 128
A.1.1 Pseudo Code . . . 128
A.2 Pseudo Code Enhancements . . . 128
A.3 Semantics of Augmented HOOD . . . 129

B *TriviCalc* - An Example **130**
B.1 *TriviCalc* Reference Manual . . . 130
B.2 Original *TriviCalc* design . . . 135
B.3 *TriviCalc* Module Structure . . . 169
B.3.1 Original *TriviCalc* design Module Structure . . . 169
B.3.2 Final *TriviCalc* design Module Structure . . . 174

C Glossary and Abbreviations **179**

D Notation Summary **183**

List of Figures

1.1 Overview of *Morpheus* . . . 3

2.1 Design requirements . . . 10
2.2 Design forms . . . 11
2.3 Interaction of Coupling and Cohesion . . . 22

3.1 Waterfall Model of the Software Life-cycle . . . 26
3.2 Traffic Junction with Lights . . . 27
3.3 HOOD graphical notation - Traffic Lights . . . 29

4.1 Overview of *Morpheus* . . . 38

5.1 A graph . . . 48
5.2 A tree . . . 49
5.3 Two graphs . . . 50
5.4 Graph union . . . 50
5.5 Graph intersection . . . 50
5.6 Design Graph of Simple Stack . . . 51
5.7 A hierarchical graph . . . 52
5.8 Top graph of the hierarchical graph . . . 52
5.9 Design Graph of Stack ADT with Caller . . . 54
5.10 Object Structure of Stack ADT with Caller . . . 54
5.11 Graph of $\log_2^*$ . . . 57

6.1 Simple graph . . . 60
6.2 Chain Graph . . . 61
6.3 Star Graph . . . 62
6.4 A Nested Design Graph . . . 63
6.5 Modular Tree Structure of Nested Design Graph . . . 64

7.1 Architecture of *Morpheus* . . . 77
7.2 Data Analysis Phase . . . 79
7.3 Symbol Table . . . 79
7.4 Object Structure Table . . . 81
7.5 Entity Tree . . . 81
7.6 Entity Structure Table . . . 81
7.7 Linkage Table . . . 82
7.8 Secondary Information . . . 82
7.9 Improvement Engine Phase . . . 83
7.10 *Morpheus*'s Search Strategy . . . 84
7.11 Active List . . . 84
7.12 History List . . . 85

8.1 Grouping with 2 groups of 3 entities . . . 88
8.2 Grouping with 2 groups of 5 entities . . . 89
8.3 Grouping with 2 groups of 10 entities . . . 90
8.4 No structure with 2 groups of 5 entities . . . 91

8.5 Cohesion with 2 groups, one with 10 links 93
8.6 Cohesion with 2 groups, one with 9 links 93
8.7 Cohesion with 2 groups, one with 8 links 94
8.8 Cohesion with 2 groups, one with 7 links 94
8.9 Cohesion with 2 groups, one with 6 links 95
8.10 Cohesion with 2 groups, one with 5 links 95
8.11 Cohesion with 2 groups, one with 4 links 96
8.12 Cohesion with 2 groups, one with 3 links 96
8.13 Cohesion with 2 groups, one with 2 links 97
8.14 Cohesion with 2 groups, one with 1 link 97
8.15 Cohesion with 2 groups, one with no links 98
8.16 Coupling with 2 groups, and no links between groups 100
8.17 Coupling with 2 groups, and 1 link between groups 100
8.18 Coupling with 2 groups, and 2 links between groups 101
8.19 Coupling with 2 groups, and 3 links between groups 101
8.20 Coupling with 2 groups, and 4 links between groups 102
8.21 Coupling with 2 groups, and 5 links between groups 102
8.22 Coupling with 2 groups, and 6 links between groups 103
8.23 Coupling with 2 groups, and 7 links between groups 103
8.24 Coupling with 2 groups, and 8 links between groups 104
8.25 Coupling with 2 groups, and 9 links between groups 104
8.26 Coupling with 2 groups, and 10 links between groups 105
8.27 Graph of original Traffic Light design 109
8.28 Graph of flat Traffic Light design without environment 110
8.29 Graph of flat Traffic Light design with environment 111
8.30 Graph of *Morpheus*'s Traffic Light design without environment 112
8.31 Graph of *Morpheus*'s Traffic Light design with environment 113

B.1 Diagram of initial display 131

List of Tables

6.1 Calculation of Chain Graph's Message Length 61
6.2 Calculation of Star Graph's Message Length 62

8.1 Effect of Increasing Group Size 88
8.2 Effect of Reducing Cohesion within a Group 92
8.3 Effect of Increasing Coupling between Groups 99
8.4 Experiments with Traffic Light Design 106

Part I

Background

Chapter 1

Introduction

> Programming is neither science nor mathematics. Programmers are not adding to our body of knowledge; they build products.
> ...
> The majority of engineers understand very little about the science of programming or the mathematics that one uses to analyze a program, and most computer scientists don't understand what it means to be an engineer.
>
> Parnas (1997)

Software design is a hard problem involving domain knowledge, creativity and software engineering skills. This thesis focuses on the application of software engineering for the creation of good quality, modular designs. This thesis does not address domain knowledge or creativity, leaving these instead to human experts and other research.

Software engineering has many dimensions, including: economics of software production and ownership, management of software projects, the quality and safety of software products as well as the more limited role of programming.

1.1 Purpose of the Research

This research aims to develop a prototype tool for improving software designs. We have called our prototype *Morpheus*, as a reminder that it changes the shape of a software design. In our case *Morpheus* manipulates a design's architecture to produce a simpler and 'better' representation, see Figure 1.1.

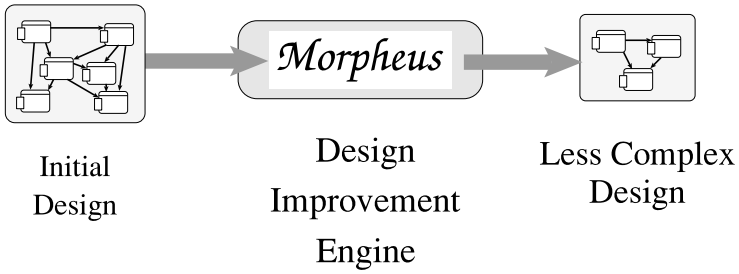


Figure 1.1: Overview of *Morpheus*

Clearly, the development of such a system raises many questions, most notably

- What is a software design?

- How is the input of a design to be expressed?
- How are alternative designs created?
- What constitutes a 'better' design?

It is the purpose of this thesis to try and answer these questions.

1.2 Motivation

We know from empirical studies (Boehm, 1981), that the cost of correcting defects grows significantly the later in the development process the problem is uncovered. Therefore the more potential errors that are found in the early stages of development reduces the economic costs of owning the software. This potential for significantly decreasing costs means that the design phase of software development is an area which merits further research. Moreover, software design is a sophisticated human skill worthy of study for its insights into other intelligent behaviour.

Most Computer Aided Software Engineering (CASE) tools available today, are little better than glorified drawing packages sometimes with associated databases. Such tools provide support for drawing pictures, and recording information about the software being designed. The more sophisticated systems allow information to be shared by several engineers, and detect improper use of notation and missing elements. Although useful these facilities are limited and perform only a shallow examination of the software being designed. What is needed are tools which provide constructive guidance to produce better designs. Such tools would involve a much deeper analysis of the product being designed.

1.3 Complexity: the Central Problem

Having recognised that software design needs improving, we need to identify what we mean by 'better'? Clearly there are a range of answers to this question involving concepts such as faster, cheaper, less complicated, easier to maintain and more reliable.

In this thesis we define 'better' as being structurally less complex. It seems reasonable that *ceteris paribus* a less complex artefact will be preferred over more complex artefacts. Moreover, by making the design less complex, it should be easier for the designer to spot other problems, which should increase overall reliability as well as reducing costs.

Parnas (1972) introduced the concept of modules into software engineering; a technique well known in other engineering disciplines. Parnas's module was a much simpler affair than today, broadly being a sub-program.[1] However, the concept evolved through structured programming, and abstract data types into what today would essentially be called an object.[2]

Early in the evolution of modules, people began to realise that some modules were 'stronger' than others, and less influenced by changes in other parts of the system. This was aptly named the 'ripple-effect', because seeming minor changes in one part of the system could cause perturbations right across the entire system. Thus people became very interested in how to create modules which were self-contained and had little interdependency with their neighbours. These notions were formalised into the concepts of cohesion and coupling. Cohesion being the singleness of purpose of a module, i.e., everything in the module should contribute to the module's purpose and nothing else. Coupling is the interdependency between modules, i.e., how vulnerable the module is to changes in other modules.

This early work implicitly assumed that modules did not themselves contain further modules. However, there is no reason why a module should not itself contain sub-modules which are, of course, modules in their own right. By analogy with family trees it is convenient to call a module's

[1] In the subroutine sense of the word.

[2] The encapsulation of a state together with a well-defined interface to access and change the state.

sub-modules its *children*, and the module containing a given sub-module its *parent*. Modules which may contain sub-modules are called *nestable*. Modules, unlike humans, can only have *one parent*. Further, if a module is contained in another (larger) module, then the whole of the sub-module must be contained in the single parent.

When an entity has to be shared between several modules, there is potentially some tension as to which module should 'own' the entity. This problem is exaggerated by nesting modules. Since, by implication of good design, a module (at any depth) must be cohesive; however a module's children (if any) must have low coupling between them. As Müller et al. (1993) have observed, humans are good at identifying building blocks, given sufficient time, but such time is often not available.

Whilst much has been written about coupling and cohesion in isolation, little has been said about how to make these trade-offs. Still less has been written about how to perform trade-offs in the presence of nestable modules. We conclude therefore that an automatic tool for reducing structural complexity would have significant benefits. Such a tool would have to encompass 'knowledge' of how to balance coupling, cohesion and module size to achieve a better structure.

1.4 Synopsis of the Research

1.4.1 Assumptions

In the last few years, much has been written about object-oriented software. This thesis adopts an object-based view rather than a strictly object-oriented view. The concept of an object being an instance of a class, and indeed the entire concept of a class hierarchy is not provided in the object-based view. In the object-based paradigm objects have well-defined interfaces which provide services to other objects. This reflects the authors' belief that we still do not understand coupling and cohesion in this simpler object-based paradigm. Recently, doubts have been expressed about the maintainability of large object-oriented systems, due to separation in space and time encouraged by inheritance (e.g., Binder, 1996). Furthermore, until fairly recently textual formalisms for capturing object-oriented architectural designs were not readily available; this situation has changed with the development of the Unified Modelling language (UML) (see Fowler and Scott, 1997; UML, 1997).

1.4.2 Goals

We can state *Morpheus*'s goal succinctly, as: Given an initial *closed*[3] design, *Morpheus* seeks a structurally less complex (alternative) design. *Morpheus* takes the lowest level connections formed by the designer as *fixed* and manipulates the design's modular structure.

1.4.3 Anticipated Benefits

Several benefits flow directly from this research

- A structural complexity measure, which permits the complexity of alternative designs to be compared. Such measures are currently not available.
- A method for finding alternative architectural designs.
- An automatic system for finding simpler designs (if possible) to a proposed design.

Potential benefits of this work include

- Easier to understand designs.

[3] A system is said to be closed, if there are no references in the system to entities not defined in that system.

- More defects found at design time, because the design is more coherent and easier to understand.
- Lower software ownership costs.

1.5 Thesis Structure

The remainder of this chapter provides a brief sketch of our proposed solution. Chapter 2 examines the meaning of software design, what it means for a design to be *good*, different ways of capturing designs and the important notion that an architectural design can be viewed as a graph. Chapter 3 provides a brief overview of HOOD. Chapter 4 reviews previous work on measuring software design complexity.

Chapter 5 provides the necessary mathematical background for understanding the calculation, derivation and theoretical validation of our proposed complexity measure. Chapter 6 presents our complexity measure, Ψ, in detail, explaining how it is calculated and why it is a complexity measure.

Chapter 7 describes our extensions to HOOD for capturing a more detailed description of the proposed software architecture and the implementation of our prototype system (*Morpheus*) for improving designs. Chapter 8 describes the application of *Morpheus* to a moderately sized software design. As our sample project we have chosen a simple spreadsheet, which the authors designed and then processed through *Morpheus*. We examine *Morpheus*'s proposed changes and consider whether the design has been improved.

Finally, Chapter 9 provides a summary of this thesis and a critical evaluation of what has been achieved in this research.

1.6 Outline of our Solution

This section provides a brief overview of the proposed solution to the problem outlined above.

After reviewing several possible options for expressing designs, the HOOD notation was selected. It has the advantages of both a graphical and a textual representation and is designed for capturing architectural designs (Delatte et al., 1993; Rosen, 1997).

An architectural design can be represented as a hierarchical graph, where nodes may expand into further graphs.

We now need a complexity measure for our hierarchical graphs. This is loosely based on Kolmogorov complexity, using a form of the Minimum Description Length principle (Rissanen, 1978). Our measure, Ψ, is the length of a decodable message describing the structure of the design graph.

The mathematical theory covering Kolmogorov complexity assures us that the lengths of such messages provide an approximate measure of the absolute complexity of the underlying object, assuming, of course, that the coding used to describe a graph's structure is a reasonable approximation to the best possible. In reality such an assumption is unprovable. However, the approach does give us a handle into finding a complexity measure.

Using our complexity measure, we are able to develop a few theorems (with proofs) that suggest that our measure does not run counter to intuition. Further, we prove that all of Weyuker's (1988) proposed complexity properties are satisfied by our measure.

Once we model an architectural design as a hierarchical graph, finding an alternative design just becomes a matter of manipulating the graph. This combined with our complexity measure, allows us to decide easily if a proposed design is better than another.

The beginning of wisdom is found in doubting;
by doubting we come to the question,
and by seeking we may come upon the truth.

PIERRE ABELARD (1079–1142)
French scholastic philosopher, theologian

All human knowledge thus begins with intuitions,
proceeds thence to concepts,
and ends with ideas.

EMMANUAL KANT (1724–1804)
Quoted in Hilbert's *Foundations of Geometry*

Chapter 2

Software Design

Synopsis

This chapter examines the meaning of software design in more detail. We start by asking "What is design?", and looking at the variety of different functions that a design has to perform. In particular we shall see that a design is not purely mechanical but captures the value judgements of those who contribute to the design. We shall then look at various ways for capturing designs and briefly review a broad range of design methods. We shall then examine the established properties of a *good* design. We conclude by looking at the meaning of architectural design and the idea of a design as a graph.

2.1 Design Theory

Design[1] theory is concerned with the nature of design in a largely domain independent manner. This allows us to look at software design in a more abstract style, and illustrates why many software design methods have failed to meet the expectations of their advocates and the software industry.

This and following sections summarise (based on Dasgupta's 1991 exposition) the main results of applying design theory to software.

One of Dasgupta's reasons for this examination of design is to address philosophical questions relating to the role of design as a science; and the relationship between science and engineering. We, however, are more interested in what this study has to say about the nature of design from a pragmatic engineering (i.e., usability) perspective. It should be noted that Dasgupta argues that the methodical distinction between science and engineering is extremely blurred, if it even exists. We have no reason to disagree with this view.

Following Dasgupta we shall look at the following questions about design:

- What is design?
- The role of values in design.
- How is a design expressed?
- How do designs evolve over time?

[1] This section is heavily based on the work of Dasgupta (1991).

A variety of design methods have appeared over the years (see Section 2.4), most of which advocate a strict approach to design, usually top-down, and occasionally bottom-up. But both experience and studies of professional programmers suggest that an opportunistic approach is taken to design (see Visser and Hoc, 1990), see also Section 2.2 below. Thus a design method should not force a particular approach but rather foster the design development process. Moreover if design is a search activity, it is unlikely to progress smoothly and it will evolve in several directions at once.

Attempts have been made to automate the process of reverse engineering by identifying concepts in programs and designs, for example Biggerstaff et al. (1994). However these are large undertakings and are some way from demonstrating their general applicability. Traditionally the best solution we have to identify poor design is by inspection of some kind—generally a formal design review (e.g., Fagan (1976) and Yourdon (1986)). However, such an approach is critically dependent on the reviewers and consumes hours of experienced labour which might be better employed elsewhere. There is a tendency for reviewers to concentrate on the surface presentation of the document rather than its deeper (harder to perceive) message. What we would like to do is provide some confidence that the design is 'reasonable'[2] at an early stage. This would help the designer, management and subsequent reviewers to focus on those problems which cannot be automated such as potential changes.

2.1.1 What is Design?

This seemingly innocent question belittles the complexity of design. At the outset we should recognise that design is both the process of creating a design, and the final output of the design process.[3] Hence we see that no simple definition of design will suffice. The next stage is to consider the how, what and why of designs in greater detail.

2.1.2 Values in Design

A design is *only* produced in response to a set of (possibly fuzzy) requirements. However, a set of requirements is only produced because someone recognises a need to change the current situation. The very act of perceiving such a need implies a *value judgement* that the current situation is inadequate and could be improved. Moreover, the acceptability of one solution vis-a-vis another involves further value judgements.

Recognition simply asserts that we do not believe that the current situation is the best that can ever be achieved. Evaluation is more interesting and admits that designs are assessed based on our (internal) beliefs. Such beliefs may involve such concepts as 'understandability', 'simplicity', 'value for money', 'aesthetics', etc. It should be borne in mind that mathematicians employ such concepts in judging proofs, for example "Beauty is the first test; there is no permanent place in the world for ugly mathematics " (Hardy, 1947, p.25).

Now we see the beginnings of a problem; engineering endeavours far from being purely positivist, have become normative in nature. Such a conclusion does not bode well for creating a purely analytical design tool. Whilst we may be able to differentiate between two competing designs, this must in part be done on the basis of personal values. So another individual may prefer an alternative solution based on their own value system. The difficulty with this is that such value systems are generally hidden and extremely hard to make explicit.

2.1.3 Design Problems

We saw earlier that a design is produced in response to a set of requirements. The design process is then concerned with the production of a design (what Dasgupta calls 'form'), such that if the

[2]There is an implicit assumption here that a reasonable design is in some sense "correct". The word "correct" is not used at this stage since it implies a formal notion of correctness with respect to requirements and/or specifications.

[3]This dichotomy is illustrated by the grammar of the word, which is both a noun and a verb.

design is implemented, the resultant system will satisfy the requirements (see Figure 2.1). This would also suggest that the success of a design cannot be isolated from its implementation.

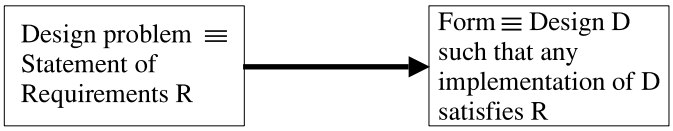


Figure 2.1: Relationship between requirements, design form and implementation, (Dasgupta, 1991, p.13).

Hence the design process can be seen as finding a solution to a set of constraints. Whilst the requirements may be heterogeneous, the final form must be internally consistent, and capable of translation into a working system.

If the produced artefact (from Dasgupta) is to be tested against the requirements, these requirements must be capable of observation. This does not necessitate that the requirements are purely functional, only that the requirements are capable of repudiation, otherwise anything would satisfy the requirements.

The phrase '*well-structured*' was introduced by Simon (1973) to refer to problems in which *all* the requirements were empirical (i.e., the fulfilment of each requirement could be empirically refuted),[4] whilst design problems containing *any* non-empirical requirement were termed '*ill-structured*' problems. The notion of a system being 'user friendly' is typical of ill-structured requirements but an individual can decide if a system is 'user friendly'.[5] It should be clear that most (all) non-trivial software design problems are ill-structured, because they contain at least one non-empirical requirement. A consequence of ill-structured problems is that the distinction between requirements and design is blurred, because the requirements must be *clarified* as part of the design process, so that the conformity of the design form can be assessed against the requirements.

The term '*bounded rationality*' was introduced by Simon (1976) as a description of the situation where a decision maker cannot or will not consider all the constraints or consequences of a decision. Again it is clear that software design problems are generally resolved under conditions of bounded rationality. Indeed it is probably the existence of bounded rationality that gives rise to changes, inconsistencies and incompleteness in requirements. We will consider bounded rationality again, when we consider the evolutionary nature of design solutions.

2.1.4 Design Forms

So far we have looked at the basic characteristics of the design problem. We must now examine the forms a design solution must take. Recall that form is Dasgupta's word for the output from the design process, i.e., *the design*.

Potential forms must satisfy some broad criteria or they are inadequate as representations of designs.

All design forms must be appropriate for communicating the conceptual (Dasgupta) design to an implementer. Depending on the capabilities of the implementer, the level of detail in the form may vary; i.e., the form may permit different 'expressions' (Bundy and MacQueen, 1994) of the artefact.[6] Hence, the division between the end of the design process and the implementation is blurred. What is apparent, however, is that the design must capture the inter-relationships between the components of the design, i.e., the macro level, whilst the micro level[7] is of slightly less

[4] This accords strongly with Popper's (1968) view of science.

[5] By limiting this comparison to an individual we have side-stepped the difficulty of defining 'user friendliness'.

[6] When plays are produced, the final interpretation of the play is left to the director. Similarly the exact interpretation of a high level language is left to the compiler; that is, although the semantic meaning may be determined by the language specification, the final instruction stream is left to the compiler.

[7] A design problem in its own right.

importance (see Section 2.7). That is, the correct identification and connectivity of components is essential to the design meeting its requirements; however, simply connecting a set of components at random does not of itself constitute a design, the whole must be a unified system.

Dasgupta also makes the observation that a design form must serve as a user guide. At first this may seem strange to software engineers who are used to separate user guides. Nonetheless, we do expect this information in a design form. Given a new object, the first few questions are likely to be "what does it do, and how do I use it?", i.e., we want a user guide. Only when we have received satisfactory answers to these questions, do we inquire into the connectivity of the object.[8]

Dasgupta's final requirement for a design form is normally not addressed by software design methods, and its absence is responsible for much current research in software and Computer Supported Collaborative Work (CSCW); a little reflection confirms that it is a necessary condition. The design form must capture the justification (and history) of a design, so that it can be critically examined and support changes. That is, the design form must encapsulate some notion of why this is the preferred design. An immediate consequence is to change the nature of the design from a static document to a dynamic form. This area is fraught with difficulties, firstly because of the volume of information and secondly the designer may be reluctant to explain his reasoning due to satisficing (see Section 2.1.5).

These differing requirements for the design form, are captured diagrammatically in Figure 2.2.

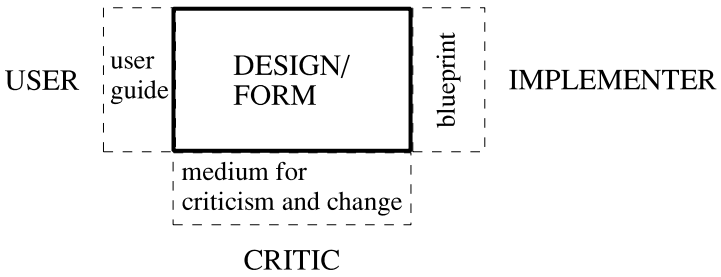


Figure 2.2: The three functions of a design form, (Dasgupta, 1991, p.57).

2.1.5 Design Evolution

The previous section examined what form the 'final' design should take, this section examines in more detail how designs are produced and evolve.

We saw earlier that for a problem to be 'well structured' it is necessary that all the (initial) requirements can be stated empirically. However, this is not sufficient. The interaction of the design components may be so complex as to be analytically intractable. Such problems are also regarded by Simon (1973) as ill-structured. The principle causes of such difficulties are

> ... the nature, variety, and mutual interdependence of the choices available to the [designer, and are such that] ... the space of possible designs or design choices is, in a practical sense unbounded.
>
> Dasgupta (1991, p.64)

Designers (equipped with only bounded rationality) and ill-structured problems respond by *satisficing*.[9] An individual satisfices when faced with a decision which must simultaneously satisfy a number of constraints, he is content to find *a solution* rather than the *optimum*. This of course saves resources (conspicuously time) that would be used in searching for an optimum solution. We recognise that such behaviour is probably the norm in software houses and may be exacerbated

[8] Dasgupta calls this part of the design form, the *context*.

[9] The term satisficing is used by Dasgupta (1991) and Simon (1981), but as far as we know the term originated in the economic literature.

by the presence of externalities.[10] It is also clear that externalities exist in software design; for example a software house which produces highly maintainable code will not receive a higher fee for it.

An immediate result of satisficing is that software producers may be reluctant to report how much of the design space has been explored; and the design will be sub-optimal.

2.1.6 Summary of Design Theory

Before looking at what we can do about software design problems it is worth recapitulating what we have learnt from looking at design theory.

The distinction between the design process and the final form of the design is blurred. Further, designs are not value free, but instead encapsulate the values of their creators and critics. Hence the objective analysis of a design is not an achievable goal, since it would imply the ability to assess values which we regard as 'highly' intelligent behaviour.

The form used for expressing designs must be able to serve as a blueprint (Dasgupta, 1991), as a user guide, and capture the justification for the particular solution chosen. It is precisely this latter requirement that most software design methods fail to meet.

A design is produced in response to initial (possibly fuzzy) requirements; these requirements must be refined and unified into a final internally consistent system prior to implementation. However, the author of the requirement specification also possesses bounded rationality, and so is unlikely to produce *the requirement specification* prior to design (and implementation) commencing. The implementer has some freedom in the precise detail of the produced artefact. All of which implies that the distinction between requirements and design is extremely ambiguous; to the extent that the design activity can be viewed as the successive refinement of requirements. This attitude is also supported by the work of Galton (1992) who examines the relationship between specification and implementation.

We have seen that most software designs are ill-structured, particularly large systems, which in the presence of bounded rationality and limited resources means that designers will seek a solution rather than the best solution. We must further accept that such limitations exist during the requirement stage. Therefore changes to a system are the norm rather than the exception.

2.2 Psychological Aspects of Software Design

> The principal limitations on the programs we write are often imposed by our inability to comprehend the design - not by the physical capabilities of computers
>
> Wulf et al. (1981, p.1)

The first computers were used only for calculations, and designing systems was not a complex problem, but as computers became larger and the associated applications grew, so did the problems of designing systems (Dijkstra, 1972). Structured programming started by Dijkstra (1968) was the first attempt to reduce this complexity, and although not universally accepted (see Grogono, 1980; Weinberg et al., 1975; Atkinsom, 1977), it became accepted as *de facto* practice (Dahl et al., 1972; Wirth, 1974; Dijkstra, 1976).

In 1976 de Remer and Kron introduced the notion of programming-in-the-large, and highlighted the difference (in kind) between small programs and large systems. Large systems are seen as both technical and managerial problems. A study by Nakajo and Kume (1991) identified a partial taxonomy of design errors, the top-level classes being: program faults, human errors causing interface faults and human errors causing function faults. An examination of these classes suggests that the program faults correspond to small scale programs whilst the latter two correspond

[10] Externalities exist when there are 'economic costs' associated with a product/decision which are not borne by the decision maker.

to large systems. The significant feature of these latter classes is a breakdown in communications between people.

This explains our assertion in Section 2.1.4 that the macro level of design is more important than the micro level. The identification of a design's building blocks and how they interrelate makes the design of the internals of a object much easier; because we have already identified its requirements and its operating environment. This significantly aids the software engineer and management. Management can be better informed how long each object will take to construct because the problem has become smaller (c.f., divide and conquer). Also, the identification of objects serves to identify the necessary communications channels between engineers rather than making it a free for all. If, unfortunately, the wrong objects are identified, this will lead to greater stress on the design as it will be unclear which object is responsible for each part of the whole.

Studies of expert designs Visser and Hoc (1990) suggest that far from using a single pass top-down (Yourdon and Constantine, 1979), designers re-iterate several times and employ an opportunistic approach. Features of opportunistic development include

> Starting the decomposition in the middle of the tree. Working simultaneously on two distinct branches. Making interruptions for digressions at other than the current level, for example, to deal with other subproblems or to define primitive operations, that is, elements at the lowest level. Descending in the decomposition tree, but coming back afterwards, for example to introduce a whole new solution decomposition level.
>
> Visser and Hoc (1990, p.242)

This leads us to conclude that a design system must facilitate switching between levels. The results reported by Nakajo and Kume (1991) indicate that breakdowns in communications between individuals and groups (of individuals) are responsible for many of the difficulties experienced in large projects. This would suggest that allowing all personnel access to the same design model (and corresponding information) would help to reduce these problems. This corresponds to the blackboard model in artificial intelligence, see for example Englemore and Morgan (1988).

2.3 Design Notations

This section examines various languages for representing designs. We shall examine the advantages and disadvantages of the different language forms used for software design.

In Section 2.1 we examined the nature of design and the design process. This section looks at how designs are typically represented and constructed in software engineering. It should be appreciated we cannot entirely separate the design notation from the design method used. Section 2.4 looks at various methods used for solving the design problem.

Language is vitally important in design, and at least four potentially different languages can be identified in the design process:

- The language used in the requirement specification to represent what the system is required to do. This language is important because it sets the tone for subsequent development. How do you express a requirement in a form understandable by non-mathematicians whilst still being suitable for machine reasoning?
- The language used between designers to communicate and discuss ideas.
- The language used in the formal design documentation produced at the end of the design phase of a project.

  By using a naturally rich language for the design, we can concentrate on the design and not keep wondering about how low-level libraries need modifying to handle new operations. The nearer the design language is to the problem, the more confidence we have in the solution and the easier it is to understand.

This language need not be the same as the previous language.

- The target implementation language. This impacts on the types of abstraction which will be considered by the designer. Whilst it is true that all software ultimately runs in machine code, some languages are better suited to specific tasks, because they provide better abstractions for the required task.

It should be noted that Customers often impose languages (e.g., RSL or Z, Ada or C). This is a pity because it limits the engineers' choices for exploring the problem domain.

Each time a designer changes between different languages, there are two consequences. Firstly, effort has to be put into the translation, and there is always the danger of information being dropped because it does not translate well, i.e., semantic distance. Secondly, changing languages brings a change in perspective, which may or may not be beneficial.

In the following sections we look at three potential languages.

2.3.1 Natural Language

Natural language plays a vital part in the design process not only because it is how we communicate with each other; but because it is how the original requirements will have been first represented. Two distinct but related phenomena may occur. Firstly, the same word or phase may be used by different people in different ways (Curtis et al., 1988), and hence an implicit confusion is introduced. Secondly, different words may be used for what is in essence an identical concept consider (for example) the pairs "on/off" and "valid/invalid", they are both fundamentally Boolean but we find it useful to retain distinct names related to their intended use. Therefore we would like a design assistant to understand natural language, but this seems an unreasonable expectation in the medium term. Perhaps in the shorter term it might be possible to match 'features' against a 'basic' understanding. This would require at least a semi-formal language, so that requirements, etc. can be processed, compared and contrasted. Such a semi-formal language must be wide spectrum to permit many different concepts to be expressed. Unfortunately, such a wide spectrum language carries its own problems in recognising that apparently different descriptions are in fact coincident. However, a semi-formal language should have associated proof rules to demonstrate the equivalence of different sentences. This would require the construction of a minimal sub-language.

A common method for capturing designs is just plain natural language. This has been criticised as being imprecise etc. However, this is not necessarily a key feature of natural language and may well be caused by various groups higher up the design process not wishing to commit themselves. We should be concerned about natural language because it is large and not amenable to automatic analysis. Bear in mind though that it is understandable by all groups involved in software production. Moreover it is a language used to express our vague ideas.[11]

We need to represent designs clearly and unambiguously, so that our tool *Morpheus* can examine the design and check for consistency etc.

Once we move away from natural language three difficulties become apparent:

- Can we express our ideas in the new smaller language?
- If our new language is accompanied by natural language, which takes precedence?
- Translation between languages can loose or gain concepts in the process.

Natural language can lack structure and may be hard to follow whereas a more artificial language may impose structure.

[11] We are not suggesting or denying it is the language of thought—merely that we do use it for communication, often before we are prepared to commit ourselves to a more formal notation.

We are interested in architectural design, so what does this tell us about our design notation? Architecture is concerned with how the bits fit together, *not* with the detailed operation of a part. Therefore a good candidate language would have strong emphasis on connectivity and less emphasis on internal details.

2.3.2 Graphical Notations

It has been said many times that "a picture is worth a thousand words". This may be true of the right sort of picture. Clearly, in the context of architectural design, showing relationships between parts is a central issue. However, we cannot directly analysis a picture—even through people find these useful for communication and clarification of ideas. Pictures are good at explaining how parts interrelate, precisely what is often vague in a natural language. However, if we choose a language with a well-defined mapping between figures and a more formal textual representation, we can analyse the text.

Pictures convey information, and certainly help understanding, but they can take a long time to draw, and are hard to maintain.

A picture (for example a structure chart) must conform to conventions and in so doing a syntax is imposed. The difficulty with free text systems is that anything goes. Imposing a syntax on text reduces its simplicity and more effort is required to write valid text. Many modern design methods require the production of a large number of pictures. These pictures generally convey information very well, and certainly help understanding, but they take a long time to draw,[12] and are often harder to maintain. Note maintenance might just be a name change. A solution to this criticism is simple, diagrams must be easily convertible between graphic and textual representations, so that engineers can perform systematic changes using a text editor. Some CASE tools[13] are beginning to provide this type of support. Hence we would like our semi-formal language to have a pictorial representation, even if, the picture represents only a subset of the full language. We can imagine a designer producing a rough design using pictures that are subsequently refined into a semi-formal (textual) language. The reader will observe that there is nothing new in such a system, however, the ability to re-engineer a 'draft' design does not seem to be provided by current CASE tools. Most of the current generation of CASE tools present a graphical front-end. The 'intelligence' of these systems is often shallow, concerned more with superficial design rules concerning the syntax of diagrams rather than the deeper knowledge used by a human in constructing the underlying system (Bujnowski, 1993).

A further point to consider with a pictorial presentation is "what exactly does the picture mean?". There is research in progress (Ward and Mellor, 1985; France, 1992; Fuggetta et al., 1993) to give Data Flow Diagrams (DFD) a precise semantics, but this is generally accomplished by adding more symbols into the visual language, and hence moving away from a simple picture, and therefore requires more knowledge to interpret the presentation.

2.3.3 Formal Languages

Formal languages, for example RSL (George et al., 1992), VDM-SL (Dawes, 1991), CSP (Hoare, 1985) and Z (Spivey, 1989), aim to remove the imprecision inherent in natural languages by using a mathematically based notation for capturing designs. It is true that programming languages are formal in the above sense, but they are not sufficiently wide for specification languages (Pressman, 1992, p.288). By wideness, we mean that the semantic domain which can be expressed in a programming language is limited to the set of computable functions, whereas a formal specification language permits a larger set of functions to be represented. Usually specification languages carry a proof obligation to show that the resulting system is implementable (e.g., George et al., 1995, p.240). Whilst it may seem strange to permit such a situation, such functions can simplify the

[12] For example, try drawing a Data Flow Diagram in MacDraw.

[13] For example Software through Pictures.

development and expression of a specification. Formal notations are intended to be accompanied by a natural language description of what the mathematics is modelling.

Formal languages have the advantage of being precise, unambiguous and amenable to rigorous analysis using all the leverage that mathematics can bring to bear. Moreover they permit the engineer to move away from the fuzzy languages used in the initial specification, and use a more abstract and precise notation. Precise notation allows the designer to look for missing parts of the design/specification and ambiguities, whilst also permitting a more abstract model to be developed which allows alternatives to be explored.

However, formal languages are not without their problems. Most notably their very reliance on mathematical notation and reasoning which the average engineer is unfamiliar with. This is not unreasonable since the software engineers must communicate with customers and other non-specialists. Also as Jackson (1995, p.116) has noted "formalists often forget the need to tie their descriptions to the reality they describe". Fetzer (1988) observed that it is impossible to mechanically (completely) derive an implementation from a specification, which some advocates of formal methods seem to believe. The cost (in terms of time) of producing a formal model, can be quite high and may not be justifiable in terms of the benefits to the project.

There are undoubted areas on some projects where the advantages of formal methods outweigh their disadvantages, but they should not be seen as a panacea, but rather as a valuable part of software engineering's toolkit.

2.3.4 Choice of Language

By this stage the reader may be wondering about our choice of design notation. We believe that pictures without too many different types of symbols are an excellent media for discussing and describing the structure of a design. However, for automatic processing the pictures must be supported by suitable text based notation. It important that the designer supports the pictures and formal text by informal annotation explaining how his model relates to the problem at hand. We shall return to this topic in Section 3.8, when we examine our reasons for choosing HOOD as our design notation.

2.4 Design Methods

In this section we look at a range of different design methods. Broadly, design methods can be categorised into the following groups: Functional Decomposition, Data-Driven and Object-Oriented Design. Formal methods span all of these. The following sections will examine each of these in turn, focusing on their key features and providing various examples. It should be borne in mind that this categorisation is neither exhaustive nor clear cut.

2.4.1 Functional Decomposition

The overall system is viewed as performing a small number of functions, each of which is successively refined until sufficient detail is achieved to permit coding to start. Generally these methods are interested in the flow of data (de Marco, 1979; Gane and Sarson, 1979; Yourdon, 1989) round the system.[14] Functional decomposition is the oldest design method and the most natural. Examples of these methods include:

- Structured Design (SD) also known as Stepwise Refinement (Wirth, 1971),
- System Architect's Apprentice (SARA) (Lor and Berry, 1991),
- Structured Analysis and Design Technique (SADT) (Marca and McGowan, 1988),

[14] For real time systems see Ward and Mellor (1985)

- Modular Approach to Software Construction Operation and Test (MASCOT) (MASCOT, 1987)

Let us briefly consider stepwise refinement as an example of this design methodology. Stepwise refinement was proposed by Wirth (1971). The design is developed by successively refining the previous procedural detail. Thus a system is progressively decomposed from high level functional statements until programming language statements are reached. This process can be though of as elaborating the design, at each iteration we provide more detail.

At least three different "rules" for refinement have been identified, namely (Grogono, 1980): divide and conquer, make finite progress, and analyse cases. It is important to realise that at each iteration a decision (there are always choices) must be made on the "best" way to proceed. Following this method can lead to dead-ends, and therefore it may be necessary to backtrack and re-iterate again.

The method is not prescriptive and does not guarantee a solution, nor indeed does it always provide a notation. It is heavily biased towards the Waterfall model, and is often used as a basis for teaching design.

The criticisms raised against functional decomposition stem from three main observations. Firstly, the top level decomposition must be made when knowledge of the problem is least developed and the method offers no certainty that we have identified the top level function correctly or that our refinement is not a blind alley. (Think of this as a search, are we starting from the root node and which child do we visit next?) Secondly, Jackson (1983) has argued that the functions change over the life of the system as opposed to the structure of the data. Thirdly, the design of key data structures etc. can permeate the entire program.

It is the second and third problems have led to the evolution of object-oriented design.

2.4.2 Data Structured Design

These methods seek to mould the program (structure) to the structure of the data. An archetypal example is file handling. These methods do not attempt to model the flow of data through the system, but rather the static structure of the data. Examples of these methods include:

- Jackson Structured Programming (JSP) (Jackson, 1975)
- Jackson System Development (JSD) (Jackson, 1983)
- Warnier-Orr (Orr, 1971)

The major problem with these methods is their rigidity; the necessity to identity the data's structure. Additionally implementations tend to be slow; JSD tends to lead to a large number of processes, and context switching is expensive (Deitel, 1984). JSP tends to be more mechanistic than some other design methods, and has been used as the basis for some undergraduate design courses. However JSP can lead to dead ends caused by structure clashes due to discrepancies between different real-world data structures.

2.4.3 Object Oriented Design

In this group of methods, the problem domain is seen as being composed of objects and classes of objects. An object encapsulates both algorithms and data. Objects are potentially related to each other in a variety of ways, not all of which are hierarchical in nature. For example, a filled red square could be derived from a square and red coloured objects, both of which could be derived from closed objects. Examples of this method include:

- Hierarchical Object-Oriented Design (HOOD) (Delatte et al., 1993)
- Booch's Method (Booch, 1991)

- Object-Oriented Software Engineering (OOSE) (Jacobson et al., 1994)
- Unified Modelling Language (UML) (Fowler and Scott, 1997)

However, much has been written about general OOD methods without giving each a distinct label, see for example Cox (1986); Meyer (1988). OOD represents a new strand of development, which is still evolving. Unfortunately, it has developed its own specialised vocabulary which obscures its meaning. Since OOD has become popular, a number of older methods have suddenly become object oriented. However we believe that a weak form of OO can be identified. (This is not intended to imply that we should categorise methods as being weakly or strongly OO or that there is any value in doing so, merely that some OOD methods use a purer form of OO than others). By weak OOD we mean the ideas of information hiding, modularity and abstract data types, in contrast strong OOD includes the notion of class, inheritance and explicit polymorphic functions.

OOD arose from information hiding, abstract data types and in particular the development of Simula 67 (Birtwistle et al., 1973). It is apparent that to conduct a design in an OOD fashion requires suitable input from the requirement analysis phase. However a common theme is to identify objects, which it is suggested is best done by looking at the nouns and verbs in the problem description (requirement specification) (Booch, 1987); perhaps natural language processing has something to offer here. This may sound simple, but we are entitled to wonder how well this works in practice.

There is wide divergence among programming languages on how these facilities are provided (compare Smalltalk (Goldberg and Robson, 1983) and C++ (Stroustrup, 1994)), and this inevitably spills over into design. Smalltalk regards everything as an object (e.g., integers, strings, etc.), but this seems unnatural especially in relation to commutative binary operators,[15] whilst C++ is an extension of C (ANSI C, 1989) to incorporate classes, thus making it possible to program without objects at all.

All OOD methods require several iterations in order to identify all the classes and the relationships between them. It is hoped that OOD will lead to increased reuse of components. However this raises a question, how does the designer know what classes exist and what the relationships are between classes? This requires advances in indexing and identifying concepts.

What can inheritance do? Inheritance can be used to modify the behaviour of a previous class, this can be achieved in one of three ways:

- Add a new method or field.
- Delete an existing method or field.
- Modify the behaviour of an existing method, or override an existing field.

In general, deleting a method is not permitted, because most OOD systems require a class hierarchy progressing from the most abstract to the more specialised.

OOD is derived from a rigorous abstract data structure approach. Harrison and Ossher (1993) have suggested that this strict view of an object needs to be weakened somewhat to handle cases where the same objects belong to multiple classes depending on their observer's viewpoint.

2.4.4 Formal Methods

This is to some extent a collection of divergent methods, linked by utilising mathematics to provide a well-defined syntax and semantics, see also our discussion in Section 2.3.3 on formal languages. Examples include:

- Z (Spivey, 1989)

[15] Additionally in Smalltalk $a + b \times c$ means $(a + b) \times c$ (Meyer, 1988, p.438)!

- Vienna Design Method (VDM) (Jones, 1986)
- OBJ2 (Goguen et al., 1985)]

Also included in this category are:

- Finite State Machines (FSM) (Rayward-Smith, 1983)
- Petri Nets (Reisig, 1985)

The latter may be useful complements to other methods for example DFDs.

Formal methods are based on defining a precise formal specification, and then refining this specification until sufficient detail is achieved to permit implementation to occur. The obvious advantage of such a specification is that it is specified in mathematics, and can thus be subjected to the wealth of knowledge and techniques known to mathematics. The specification is produced in abstract terms. Another advantage is that such a specification facilitates the removal of ambiguities. Additionally formal methods are seen as aiding the construction of executable specifications (and automatic programming).

Several objections to formal methods can be put forward. Not least that people are uncomfortable with the required mathematics, and that most examples are small (if not trivial). Moreover, such methods have not been demonstrated on large scale systems. More serious however is that our study of design casts grave doubt on the ability to construct an initial (complete) specification. Fetzer (1988) challenged the entire concept of formal methods by distinguishing between an algorithm (which can be formally verified) and a program (running on a physical machine) which is only empirically observable.

We believe that formal methods are best used as an adjunct to less precise methods for handling areas where a rigorous proof is required.

2.5 Complexity: The Scourge of Engineering

The principle difficulty facing software engineering is complexity. As Booch (1987) puts it:

> The fundamental cause of the software crises is that massive, software-intensive systems have become unmanageably complex. Furthermore, we cannot expect them to become any less complex, for as we improve our tools and gain experience in designing such systems, we actually open up even more complex problem domains. As a solution to this crises, we must therefore apply a disciplined artistry, using tools that help us manage this complexity.
>
> Booch (1987, p.28)

How is this complexity to be brought under control? Pressman (1992) suggests the following criteria for good design:

1. A design should exhibit a hierarchical organization that makes intelligent use of control among components of the software.
2. A design should be modular; that is, the software should be logically partitioned into components that perform specific functions and subfunctions.
3. A design should contain distinct and separable representations of data and procedure.
4. A design should lead to modules ... that exhibit independent functional characteristics.
5. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.

> 6. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
>
> Pressman (1992, p.318)

Although the term module is used above, it is clear that our definition of object could equally be used in its place.

2.6 Architectural Design

In this section we look in more detail at the concept of architectural design. In particular we examine why we regard architectural design as more significant to the success or failure of a design than detailed design.

By architectural design we mean the identification of the major components of the design, especially their purposes and interfaces. How we can see the reason why we claim that detailed design is less important; detailed design is concerned with designing the internals of the identified components. As Fowler and Scott (1997, p.22) observed " ... the biggest technological risks are inherent in how the components of a design fit together, rather than present in any of the components themselves". Moreover, designing the internals is obviously a much smaller and self-contained problem than the original problem.

In practice, of course, once a 'large' component has been identified, the design of its internal structure is also architectural in nature not just detailed design. We regard the architecture of 'large' components to be part of the architectural design phase. Specifically we classify detailed design as deciding *how* a component's services should be provided rather than deciding *what* services should be provided.

We saw in the previous section that good design requires objects which are largely independent and have a good logical structure. These two concepts are captured by *loose coupling* and *high cohesion*, respectively. These concepts are further examined below, after we have described exactly what is meant by a component.

2.6.1 What is an Object?

So far, we have been deliberately rather vague about what we mean by a software component or object. We now offer a more precise definition.

> An object is a model of a real-world entity or a software solution entity that combines data and operations in such way that data are encapsulated in the object and are accessed through the operations. An object thus provides operations for other objects, and may in turn also require operations of another object. An object may have a state, either explicitly to provide control or implicitly in terms of the value of the internal data.
>
> Robinson (1992a, p.34)

This definition accords with Pressman (1992) earlier properties for good design, and gives us a good definition of an object. It is important to note that an object (generally) both provides services to other objects and requires services from other objects. This definition does not rule out mutual recursion, but normally this is rare.

Most modern programming provide the object concept, albeit under a variety of different names: class, cluster, module, package and structure.

2.6.2 Are these the Right Objects?

Having defined the term 'object', and a definition of good design properties, we how require some guidance on determining the quality of a proposed architectural design.

2.6.2.1 Coupling

Coupling describes how the components of a system are interdependent. Page-Jones (1988) describes a spectrum of coupling. For example, if a routine simply takes in a parameter and returns a result based solely on the input, then the routine is only loosely coupled (*data coupling*) to its environment. Whereas, if a search routine presumes that the data (to be searched) is already sorted and *in the same location*, it is tightly coupled (at least *common coupling*) to its environment.

A good system is only loosely coupled. However, in practice, moderate coupling is unavoidable,[16] and is not too serious provided it is restricted to well defined parts of the system.

A generally recognised coupling scale from necessary to unacceptable is shown below, (Pressman, 1992, p.336):

No Direct Coupling No direct linkage between modules.

Data Coupling Arguments are only passed between modules via parameters and returned results. Further, the data exchanged between modules is only that directly required.

Stamp Coupling Similar to Data Coupling, but extra (unwanted) data is exchanged. For example, passing an employee's full personal data to a module which just calculates the number of years with the company.

Control Coupling Two modules are control coupled when a 'flag' is used to instruct the caller to perform different actions on the supplied data.

External Coupling A module exhibits external coupling when it has knowledge about the world outside the software system. For example, specific knowledge about how social security numbers are formatted.[17]

Common Coupling Modules have common coupling when they have knowledge about global data areas. For example, a payroll module reads an employee's salary directly from a central area rather than using another dedicated module to extract the required information.

Context Coupling Modules have context coupling when one module has knowledge about the internal details of another module. For example, jumping into a module after the parameters have been verified, in the belief that the data must be correct.

Common coupling is generally regarded as the cut-off point for acceptable coupling. Note that these definitions refer to individual modules as well as objects.

2.6.2.2 Cohesion

In contrast to (undesirable) coupling, cohesion describes how well the internals (sub-components) of a component fit together. Page-Jones (1988) describes a spectrum of cohesion. For example an object to update a display may have high cohesion (*functional cohesion*); whilst objects constructed by arbitrarily cutting up a program into fixed length chunks has low cohesion (*coincidental cohesion*).

A good system has components with high cohesion. As with coupling, some low cohesion is generally unavoidable.[18]

The advent of OOD has to some extent invalidated the traditional cohesion scale. Since concentrating on one area of a data structure (i.e., *communicational cohesion*) (Pressman, 1992, p.335) is less than ideal. For example, in a library system a module may be responsible for manipulating

[16] For example, external interfaces and calls to the OS, see Pressman (1992, p.338).

[17] In England a social security number consist of two uppercase letters followed by three pairs of digits, terminated by a single uppercase letter from the set {A,B,C,D}.

[18] For example, initialisation routines, exhibit only *temporal cohesion*.

books, which results in communicational cohesion and hence is traditionally considered unsatisfactory. However, in OOD using an object to represent an abstract data type is considered good practice.

A generally accepted cohesion scale from highly desirable to accidental is shown below, (Pressman, 1992, p.334):

Functional Cohesion All components of the module contribute to a *single* task.

Sequential Cohesion The module's components are used in some fixed order to perform a task; but it lacks a strong sense of single mindedness.

Communicational Cohesion The components are located in the same module because they use the same input or output data rather than having functional cohesion.

Procedural Cohesion The components are related because they are used in some fixed order at particular moments in time. For example, the use of procedure *B* must always be preceded by the use of procedure *A*.

Temporal Cohesion Like procedural cohesion, but the components are just related in time without any predefined functional ordering. Unlike procedural cohesion, the usage of procedures *A* and *B*, can occur in any order.

Logical Cohesion Components are grouped because they share a common theme rather than a common purpose.

Coincidental Cohesion No meaningful relationship between components at all.

Procedural cohesion is generally regarded as the cut-off point for acceptable cohesion. Note again that these definitions are based on modules, but upgrade to objects.

2.6.2.3 The Relationship between Coupling and Cohesion

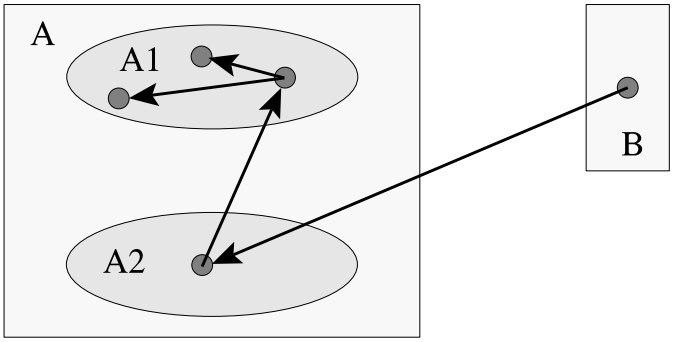


Figure 2.3: Interaction of Coupling and Cohesion

Having read these two definitions, the reader may be wondering what the precise difference is between coupling and cohesion. Cohesion measures the internal strength of an object whilst coupling measures the interdependencies between objects. In a sense these two concepts are not disjoint since changing the cohesion is likely to have a knock on effect on coupling amongst the surrounding objects. For example, see Figure 2.3, moving the entity in object *B* into *A*2 (say) would effect the coupling between *A* and *B* and the cohesion of *A*. We cannot say what would happen to *B* since we do not have sufficient information.

If cohesion within objects is increased, it may well serve to reduce the coupling between objects. If this seems a little odd, consider an arbitrary system, if the internal composition is good

then the individual objects will communicate cleanly and each object will have a definite well defined purpose.

However, with nest-able objects the relationships all become more complicated. It is worth noting that most work on coupling concentrates on the relationship between modules at a similar depth in the hierarchy rather than discussing what happens between nesting levels.

2.6.3 What does Ψ Measure?

In Chapter 3 we shall describe HOOD, our chosen design notation, and in Chapter 6 we shall describe our theoretical model for calculating Ψ, our measure of design complexity. At this point we should consider what effect our choice of input language has on measuring coupling and cohesion. Firstly, however, it should be stressed that Ψ does not measure coupling or cohesion, *per se*, but rather a balance between these two abstract concepts. Our measure, Ψ, makes no attempt to identify the type of coupling or cohesion—just the overall effect across the entire design. To do otherwise would require far greater knowledge about the nature and use of links than our present model possess. In particular knowledge of the purpose of an object, calling sequences and the internal structure of data. None of which can be expressed adequately in HOOD.

HOOD itself cannot be used to express designs which contain *common* or *context* coupling, because the language is based upon a strong notion of an object only providing services as determined by a well-defined interface. The other forms of coupling (see Section 2.6.2.1) can be expressed in valid HOOD. Note however that *external* coupling exists primarily in the 'mind' of the designer rather than the formal design. On the other hand, designs containing all forms of cohesion (see Section 2.6.2.2) can be expressed in HOOD.

The reader may feel that a metric which purports to find better designs without first explicitly determining coupling and cohesion is in some sense misguided. However, Ψ, seeks to find better designs based on the services required by each component of the design and grouping components by common requirements. This idea of grouping by requirements is not itself new, (see for example Calliss and Cornelius (1989)). From a mechanism view this is the essence of coupling and cohesion. It also means that this approach may be useful for maintaining legacy systems. It is anticipated that by seeking modules with a common set of required services that we will find a balance between coupling and cohesion. Moreover, complexity, is not just a measure of coupling and cohesion, but must involve some concept of an object's size and the size of its external interface.

Calliss's work was based on looking for particular kinds of grouping (i.e., grouping by type-families, grouping by imports and grouping by state variables) and several different kinds of graphs; all of which were reverse engineered from the source code. Our works takes a unified view of links and graphs, considering them all to be fundamentally similar. This gives us a wider choice of link, but perhaps at the cost of more specific knowledge of how to process the link.

The mathematical model presented in Chapter 6 is built upon the hypothesis that all kinds of links are fundamentally similar. A future project would be to investigate the use of a link's kind. However, we speculate that this would make the model much more complex and unwieldy.

2.7 Design as a Graph

Once we accept the notion of architectural design as being the identification of objects with their respective interfaces and sub-components, it is not a great step to realising that a design can be represented as a graph.

The structure of a software design can be regarded as a directed graph $G(\mathcal{N}, \mathcal{E})$, where the nodes, $\mathcal{N}$, represent design entities, and the directed edges, $\mathcal{E}$, represent a *requires* relation. That is, if an entity A directly requires B in order for A to provide its services, there is an edge from A to B in the design graph. For example, a routine `sqrt` to find the square root of a real number would require a `real` type, and a set of arithmetic operations on the reals.

Unfortunately, this description is a little too simple, because most large designs require some form of encapsulation/nesting mechanism. We call this encapsulation mechanism an *object*. We permit objects to nest without limit. Further, an object may contain any design entity. In the remainder of this thesis we use the term *entity* to include objects and the term *basic entity* to indicate entities which may not be objects in their own right.

Such a model of a design's structure is very general. Different design methodologies may impose different restrictions on the use, placement and connectivity of entities and objects. This model of design requires a client to know which objects and basic entities provide its required services. However, the server needs no knowledge of who uses it services.

The desire to understand the world
and the desire to reform it
are the two great engines of progress.

BERTRAND RUSSELL (1872–1970)
Marriage and Morals

Chapter 3

An Overview of HOOD

Synopsis

This chapter provides a brief overview of HOOD version 3, which is the authors chosen method for capturing designs.

Note 3.1. *This thesis is based on* HOOD *3, rather than* HOOD *4, since* HOOD *4 was released too late into the development process.* HOOD *4 is based on Ada95, which aims to provide a purer form of OOD than Ada83.* HOOD *4 extends* HOOD *3's notion of class.*

3.1 Introduction

This chapter introduces the main features of HOOD, used in this study. A number of other features are not used, and are discussed in Section 3.7, below.

3.1.1 What is HOOD?

HOOD stands for Hierarchical Object Oriented Design, but unfortunately this tells us little about HOOD. HOOD has evolved out of work done by the European Space Agency (ESA) as part of its adoption of the Ada Programming Language for very large scale projects. HOOD was derived by marrying ideas from Abstract Machines with those from Object Oriented Design.

Although HOOD is named after OOD, its principle target language is Ada. Some would argue (e.g., Booch, 1991), that Ada is not an object oriented language, and by implication HOOD is not an OOD method. There is certainly merit in this argument, Ada and HOOD lack a number of the characteristics of fully object oriented languages, for example classes, inheritance and type polymorphism. However, this thesis is not specifically an examination of object oriented designs, but rather the use of structural methods for analysing large scale designs, and there can be no doubt that HOOD is being used to create large designs for ESA.[1]

The fundamental building blocks of a HOOD design are objects, and we shall address the question of what an object is shortly (see Section 3.3).

3.1.2 Where does HOOD sit in the Software Life-cycle?

In discussing software engineering, it is useful to have a model of the production process. There are several alternative models,[2] with varying degrees of sophistication. However, for our purposes

[1] HOOD has been used on Columbus and Arianes 5 (HOOD HUM, 1996).

[2] For example the Spiral model.

it suffices to use a basic model, called the Waterfall model (Pressman, 1992). The Waterfall model is depicted in Figure 3.1. It must be stressed that the Waterfall is an idealised model, and not a description of what may happen on a real project.

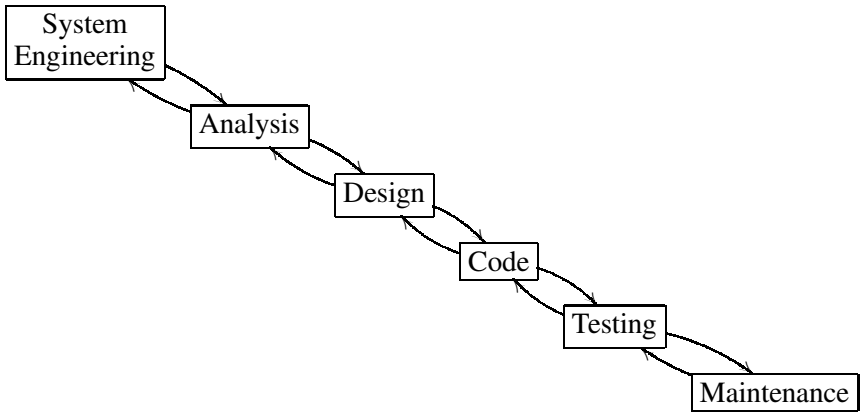


Figure 3.1: Waterfall Model of the Software Life-cycle

The European Space Agency (ESA) explain HOOD's position in the software life-cycle with reference to the Waterfall style model. HOOD can be used from the tail end of the Analysis phase until the latter stages of the testing phase.

What does HOOD provide? HOOD encourages decomposition, information hiding, encapsulation and abstraction. A design in HOOD consists of a tree like structure with the main objects at the top of the tree along with several environmental objects (see Section 3.4.4). The top objects are called *root objects*. The concept of decomposition permits both refinement and abstraction. Refinement because once an object has been defined with reference to its siblings in the design tree, it can be considered in isolation and refined in a much detail as required. The objects identified from such a decomposition are called *child objects* and the original object is called the *parent object*. Clearly an arbitrary object can (in general) be both a parent and a child object, with respect to other objects. The leaf nodes of a design tree are called *terminal objects*, and any non-leaf object is called a *non-terminal object*. Of course, if refinement shows that the parent object was incorrect it is necessary to revisit the design of the parent object. Abstraction is supported because we can represent an entire sub-tree by just one object, (i.e., the sub-tree's parent object).

We are interested in software architecture, and to quote from Robinson

> One of the major objectives of the architectural design is to provide a clear identification of the components of the design and their interfaces. Such architectures are most clearly expressed using good diagrams, especially data flow and control flow diagrams.
>
> Robinson (1992a, p.4)

ESA's experience has been that people are reluctant to commit to a particular architecture until the code was complete (Robinson, 1992a). *Not a good idea.* To help overcome this problem HOOD should support graphics and encourage the capture of designs in a more formal sense than the back of an envelope. However, we cannot perform analysis or more detailed design on a simple graphic, so HOOD provides a textual representation, called an Object Description Skeleton (ODS), which is capable of capturing all the information in the diagram and more besides (see Section 3.6).

3.1.3 The HOOD Design Method

The HOOD design method has four principle steps which are reiterated as required until the desired level of detail is reached. The four steps are (Delatte et al., 1993, p.135):

1. Problem definition

(a) Statement of the problem

(b) Analysis and structuring of the requirement data

2. Elaboration of an Informal Solution Strategy

3. Formalisation of the strategy

(a) Identification of Objects

(b) Identification of Operations

(c) Grouping Operations and Objects

(d) Graphical descriptions

(e) Justification of the design decisions

4. Formalisation of the solution

(a) Parent Object Description Skeleton

(b) Child Object Description Skeleton

3.2 Example: Controlling the Traffic Lights

In order to ease the exposition of the HOOD design notation, this section outlines a small design problem. This design is taken from Robinson (1992a), most of the HOOD design is quoted verbatim.

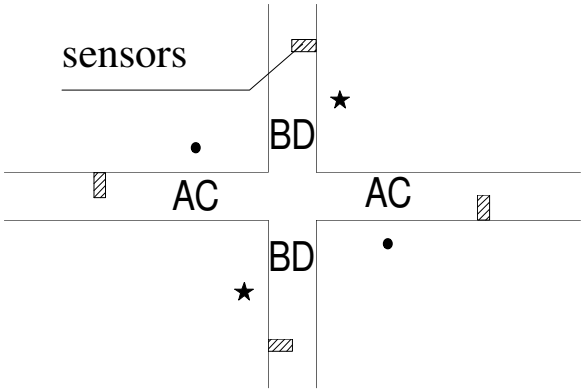


Figure 3.2: Traffic Junction with Lights

At a cross-road (see Figure 3.2 there are two sets of two lights. The lights are to be controlled by a computer so that (a) the two streams of traffic do not attempt to cross the junction at the same time, and (b) if traffic is waiting for more than 20/40 seconds, the lights change to prevent a build-up of traffic.

We shall sketch the solution as we progress in order to illustrate various HOOD facilities.

HOOD designs are based on objects as building blocks, see Sections 3.3 and 3.4. HOOD employs several different flavours of objects, and so before describing the objects in detail, it is helpful to identify the entities which make up an object.

3.3 Objects - Architectural Components

HOOD regards *objects* as the architectural building blocks. This section explains the nature of objects, and in particular objects in HOOD. What is an object? Unfortunately, this is not a simple question, and many definitions pervade software engineering. However, to provide an answer consider the following definition.

> An object is defined by the services it provides to its users, the services it requires from other objects and its behaviour, whereas the internal structure is hidden to the user, thus giving a view of how it appears to other objects. Robinson (1992a, p.27)

This looks fine, but it still begs the question *what is an object*? Basically anything the designer chooses. Put differently, it is anything that a computer system must model in software to reflect the behaviour of the real-world.

HOOD objects have three principle components:

Provided interface defining the services that the object provides to its clients.

Required interface identifies the services required from other objects, so that the current object can satisfy its own provided interface.

Internals the *hidden* details of the object's implementation.

Objects provide operations to operate on their hidden parts. An object may provide a type definition for data. Objects may encapsulate data, and hence have state. This is fundamentally different from a functional programming view, in which values are only defined whilst the function is being executed.

We prefer to think of an object as encapsulating a secret and providing services to operate on this secret. Where a secret can be for example, a piece of data, a data type or a constant. By this we mean that the details of the secret are not visible to the object's user only a set of types and operations for manipulating the secret in a well-defined way. This means that we can change the *internal* details without any impact on the object's users; since by implication they cannot *know* these details. There is however an obligation on the designer to provide a sufficient interface and not to allow unnecessary detail to be seen outside the object. For example, if an object maintained a list of books in a library, an enumerator's documentation must not suggest that the books are returned sorted by author, if this is not a necessary requirement. Otherwise, if the object later returned the books in a random order the users' of the object would have problems.

3.3.1 Traffic Lights - Graphical Notation

Returning to our traffic lights example from Section 3.2, we can how sketch a solution in HOOD's graphical notation, see Figure 3.3.

The largest curved box labelled `traffic_lights` is the top of the design tree. It exports (shown by the large rectangular box) a single operation called `second`. `Traffic_lights` consists of three children, namely: `seconds`, `traffic_sensors` and `lights`. Each child provides a single operation, respectively: `count`, `check` and `change`.

`Traffic_lights` receives an interrupt on address '1234', shown by the 'lightening bolt' and labelled 'ASER_BY_IT'. The 'A's in the corners of the objects `traffic_lights` and `seconds` shows that they are *active objects*, i.e., interrupt driven. The other objects (`traffic_sensors` and `lights`) are both *passive objects*, i.e., not interrupt driven.

Strictly speaking (but not always observed) a parent object, should not contain anything except other objects. Hence we see the dotted line from `traffic_lights.second` to `seconds.count`. Incidentally, the previous two names are called *dotted notation* for obvious reasons, and apart

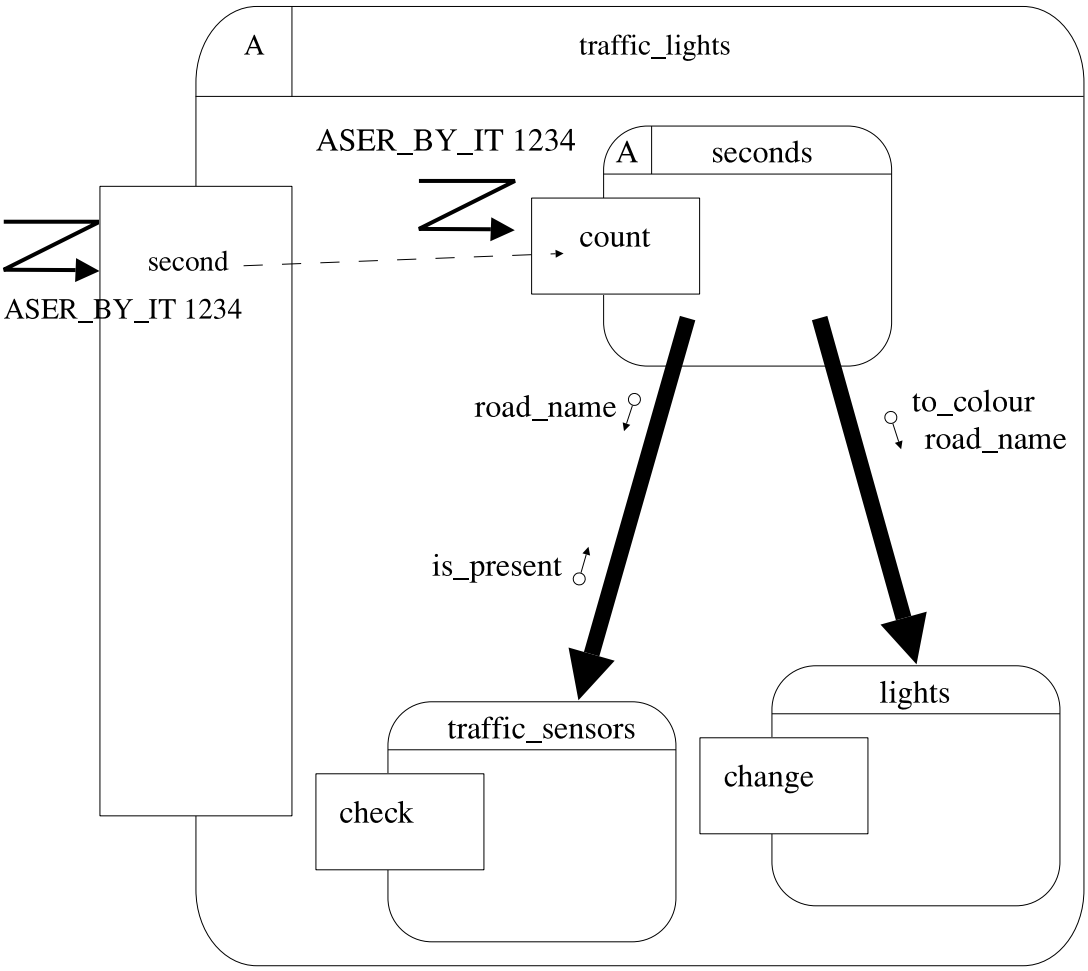


Figure 3.3: HOOD graphical notation - Traffic Lights

from possible overloading fully specify a HOOD entity. In particular, all objects in a tree must be uniquely named, and an entity is *not* named by its full path, just as object.name.

On being invoked `seconds` *uses* `traffic_sensors` and `lights` to fulfil some of its services. This is shown by the broad arrows. We cannot tell from the diagram what `seconds` requires from either `traffic_sensors` or `lights`. However, in this case, it would be a good *guess* that `traffic_sensors.check` and `lights.change` form part of the required services. It should be noted that `seconds` is not required to use `traffic_sensors` or `lights` on every invocation, but the designer is required to list all possible required objects.

The small little arrows (labelled `road_name`, `is_present`, `to_colour` and `road_name`) show data flows along the uses relationship. The direction of the arrow indicating the direction of data flow. This information is fully optional, and is generally used only the major data flows around an object.

3.4 HOOD Components

In this section we shall examine the basic components of a HOOD design in more detail.

3.4.1 Passive Objects

The basic HOOD building block, this is an object which provides operations and datatypes. A passive object can be either terminal or non-terminal. A terminal object has no children, a non-

terminal object has *at least two child objects*. It makes no sense for an object to decompose into itself, as having a single child would imply.

A passive object has a provided interface, a required interface and internals. It is important to appreciate that if an object is non-terminal all its functionality must be decomposed into its children, i.e., the non-terminal object is just a shell to aid understanding. However, this rule is not always followed and has been debated in the HOOD community (e.g., Robinson, 1992b)

3.4.1.1 The Include Relationship

If a passive object is decomposed into children, then these objects are said to be included in the parent object; hence each child object has exactly one parent. Moreover the complete functionality of the parent must be reflected in the children. That is the parent is an abstraction of the children's behaviour.

3.4.1.2 The Uses Relationship

One object uses another if the using object requires at least one operation from the other. Such usage requires that the provided interface of the server provides the desired operation and that the client cites the required operation and object in its required interface.

3.4.2 Active Objects

These are closely akin to passive objects, but are generally interrupt driven and always have a control-flow semantics.

3.4.3 Operation Control Objects

This is an object that implements the mapping between one parent operation and multiple operations of child objects. It has no provided interface, and cannot contain other objects. It is a degenerate kind of object, useful in a few situations, but it can always be replaced by a full-blown object.

3.4.4 Environmental Objects

Environmental objects represent the provided interface of another object used by the system being designed, but which is not part of the current HOOD design tree. For example the interface to the underlying operating system or part of the system being designed by another group (say).

3.4.5 Visibility

The key to understanding a HOOD design is to appreciate HOOD's visibility rules. This is a slightly complex issue. There are two cases to consider, intra-object and inter-object visibility. Not surprisingly, an entity is only visible if it has been declared. However, the order of declaration is irrelevant.

3.4.5.1 Intra-Object Visibility

- Entities declared in the Provided Interface are visible throughout the object.
- Entities declared in the Required Interface are visible throughout the object. Note however that all non-local entities must be fully named (i.e., the dotted notation).
- Entities declared only in the Internals section are only visible inside the object's internal section. Further entities named in the provided interface, must be declared again in this section.

3.4.5.2 *Inter-Object Visibility*

The question of inter-object visibility really boils down to the question, what objects are visible to the Required Interface of the object being considered. Note that all such entity references must be resolved by including the object's name.

- All environmental objects are visible throughout the system.
- The Provided Interface of all of an object's siblings are visible.
- The Required Interface of the object's parent (if not a root object) is visible.
- Nothing else is visible.

3.5 HOOD Entities

This section describes the entities which may make up a HOOD object.

Types in HOOD serve the same role as types in conventional programming languages. Types are used to identify the set of values which a variable may take, and the formal parameters of an operation.

Types are not shown in HOOD's graphical notation. Type details are shown in the textual representation. However, the Standard Interchange Format (SIF), does not define any universal mechanism for describing a type other than by name. If a type is refined, it must be specified as a comment in the target implementation language. Hence, the only form of type equivalence supported in 'pure' HOOD is name equivalence. This means that the SIF cannot represent any relationships (except name equality) between different types.

Variables are called *data* in HOOD, and may have an associated type. They cannot be referenced outside of the encapsulating object, and hence may only be declared in the internals of an object. Variables are never shown in HOOD graphical notation.

Constants are just like their counter-parts in conventional programming languages. Constants are never shown on HOOD graphics.

Operations corresponding to functions and procedures in conventional programming languages. However, HOOD makes no graphical distinction between these two categories, even the textual notation just calls them both operations.

Operation names (in the ODS) can be overloaded by declaring two operations with the same name, but with different signatures.

Operation Sets are purely a graphical convenience, permitting a collection of operations (in a single object) to be given a single name.

Exceptions are just like their counter-parts in conventional programming languages. Exception flow is shown in a HOOD diagram as a short bar across the uses arrow.

3.6 Textual Representation

Each HOOD object is described by an Object Description Skeleton (ODS). ODSes are the principle input to *Morpheus*, and are described in this section, by reference to the Traffic Lights example from Section 3.2.

The order of declaration is unimportant, and upper and lower case letters are treated as identical.

The design for `traffic_lights` is shown below

```
OBJECT traffic_lights IS ACTIVE

  DESCRIPTION
    --The traffic lights system controls four traffic lights at a crossroads.
       The traffic sensors inform the system of waiting traffic.--

  IMPLEMENTATION_CONSTRAINTS
    --The system is driven by a 1Hz clock--

  PROVIDED_INTERFACE
    OPERATIONS
      second ;

  OBJECT_CONTROL_STRUCTURE
    DESCRIPTION
      --Each second, traffic_lights is activated to look at the traffic
      sensors and to change the lights.--
    CONSTRAINED_OPERATIONS
      second CONSTRAINED_BY ASER_BY_IT --|#1234|-- ;

  REQUIRED_INTERFACE
    NONE

  INTERNALS
    OBJECTS
      seconds ;
      traffic_sensors ;
      lights ;
    TYPES
      road ; --| is (AC, BD) defines road configuration |--
    OPERATIONS
      second IMPLEMENTED_BY seconds.count ;

    OBJECT_CONTROL_STRUCTURE
      IMPLEMENTED_BY seconds ;

END_OBJECT traffic_lights
```

The **description** section introduces a textual comment describing the problem. It may contain anything the designer wishes. All comments in an ODS are bracketed by '`--{`' and '`}--`'. In addition to comments, an ODS may contain *free text*, bracketed by '`--|`' and '`|--`'. Free text may only occur in specific places in the ODS, and used as a mechanism for passing additional information to other tools, the text has no *defined* meaning in HOOD. The **implementation_constraints** section is used in the same way, but is intended to document implementation restrictions.

The **provided_interface** section lists the provided interface of the object. All exported entities must be listed. In this example we just have the `seconds` operations.

The visible **object_control_structure** section describes any semantics for interacting with other objects, usually via task rendezvous.

The **required_interface** section in this object is empty, as expected for a root object. In this example the keyword **none** is used to indicate that the filed is empty. It is perfectly acceptable to leave out any empty fields.

The **internals** section describes the internal details of the object. In this case it declares `traffic_lights` three child objects, a type `road` and an operation `second`. It also states that the operation `second` is fully implemented by the operation `seconds.count`. Finally the **internals** section (in this example) tells us that the semantics of object are fully implemented by the child object `seconds`.

The design for `lights` is shown below

```
OBJECT lights IS PASSIVE

  DESCRIPTION
    --{Object lights is used to set a traffic light pair to a selected colour;
       allowing for proper sequencing of all lights as necessary for safety.}--

  IMPLEMENTATION_CONSTRAINTS
    --{In this simulation, text_io is used to provide a readable output.}--

  PROVIDED_INTERFACE
    TYPES
      colour ; --| is ( RED, RED_AMBER, GREEN, AMBER ) |--
    OPERATIONS
      change ( road_name : IN traffic_lights.road ;
               to_colour : IN colour ) ;

  REQUIRED_INTERFACE
    OBJECT traffic_lights
      TYPES
        road ;

    OBJECT text_io
      TYPES
        string ;
      OPERATIONS
        put_line ( item : IN string ) ;  --| print a string |--

  INTERNALS
    DATA
      other_road : traffic_lights.road ;

    OPERATION_CONTROL_STRUCTURES

      OPERATION change ( road_name : IN traffic_lights.road ;
                         to_colour : IN colour )
        DESCRIPTION
          --{The data item other_road is initialised to the opposite of the
             value of road_name. If the requested colour is GREEN, operation
             change controls the full sequencing from GREEN to AMBER to RED
             for one light set, and RED to RED-AMBER to GREEN for the other
             light set.
             If the requested colour is RED or AMBER, operation change simply
             sets the requested light to RED or AMBER.}--

        USED_OPERATIONS
          text_io.put_line ( item : IN string ) ;

        PSEUDO_CODE
          --|if road_name = AC then
               set other_road = BD
             else
               set other_road = AC
             end if ;
             if to_colour = GREEN then
               set other_road lights to AMBER ;
               set road_name  lights to RED-AMBER :
```

```
            set other_road lights to RED ;
            set road_name  lights to GREEN :
          else
            set road_name lights to to_colour ;
          endif |--
    END_OPERATION change

END_OBJECT lights
```

Much of this is as for `traffic_lights` so we will only discuss the new sections.

The **required_interface** section now says that `lights` requires types `traffic_lights.road` and `text_io.string`, in addition to the operation `text_io.put_line`.

The new section **operation_control_structures** contains an entry for each operation declared in the **internals**. Each operation is described as required. This is followed by a list of used operations, and optionally as comments) **pseudo_code** and the final **code**.

The designs for `seconds` and `traffic_sensors` are shown below

```
OBJECT seconds IS ACTIVE

  DESCRIPTION
    --{Object seconds is activated from its parent object traffic_lights by the
       operation traffic_lights.second. It checks for traffic and changes the
       lights if appropriate.
       Seconds keeps a count of the time since the last light change and the
       road pair that is GREEN (AC/BD).
       After 40/20 seconds elapsed, seconds checks the traffic_sensors each
       second. When the traffic sensors show that there is traffic waiting at
       the other road, the lights are changed.}--

  IMPLEMENTATION_OR_SYNCHRONISATION_CONSTRAINTS
    --{Operation count of object seconds is activated once every second by
       interrupt at address 1234.}--

  PROVIDED_INTERFACE
    OPERATIONS
      count ;   --{ activated by interrupt }--

  OBJECT_CONTROL_STRUCTURE
    DESCRIPTION
      --{Seconds keeps a count of the time since the last light change and the
         road pair that is GREEN (AC/BD).
         After 40/20 seconds elapsed, seconds checks the traffic_sensors each
         second.}--

    CONSTRAINED_OPERATIONS
      count CONSTRAINED_BY ASER_BY_IT --|#1234|-- ;

  REQUIRED_INTERFACE
    OBJECT traffic_lights
      TYPES
        road ;

    OBJECT lights
      TYPES
        colour ;
      OPERATIONS
        change ( road_name : IN traffic_lights.road ;
```

```
                to_colour : IN colour ) ;

    OBJECT traffic_sensors
      TYPES
        present
      OPERATIONS
        check ( road_name  : IN traffic_lights.road ;
                is_present : IN OUT present ) ;

    DATAFLOWS
      road_name => lights ;
      to_colour => lights ;
      road_name => traffic_sensors ;
      is_present <= traffic_sensors ;


    INTERNALS
      TYPES
        second ; --| is new integer |--
      DATA
        elapsed : second --| := 0 |-- ;
        ac_present : traffic_sensors.present ;
        bd_present : traffic_sensors.present ;
        current_green_pair : traffic_lights.road
                                --| := traffic_lights.AC |-- ;

      OBJECT_CONTROL_STRUCTURE

      OPERATION_CONTROL_STRUCTURES

        OPERATION count
          DESCRIPTION
            --{This operation implements the logic of operation seconds to
               control traffic light changes according to the presence of
               traffic on the two road pairs.}--
          USED_OPERATIONS
            lights.change ( road_name : IN traffic_lights.road ;
                            to_colour : IN colour ) ;
            traffic_sensors.check
                          ( road_name  : IN traffic_lights.road ;
                            is_present : IN OUT present ) ;
          END_OPERATION count

END_OBJECT seconds


OBJECT traffic_sensors IS PASSIVE

  DESCRIPTION
    --{Object traffic_sensors reads the hardware sensor data to find out if
       traffic is present, and returns the value is_present set to TRUE or
       FALSE.}--

  PROVIDED_INTERFACE
    TYPES
      present ; --| is boolean |--
    OPERATIONS
      check ( road_name  : IN traffic_lights.road ;
```

```
              is_present : IN OUT present ) ;

  REQUIRED_INTERFACE
    OBJECT traffic_lights
      TYPES
        road ;

  INTERNALS
    TYPES
      latch ;
    DATA
      ac_sensors : latch ;
      bd_sensors : latch ;
    OPERATIONS
      read_sensor ( sensor : IN latch ) RETURN present ;
      check ( road_name  : IN traffic_lights.road ;
              is_present : IN OUT present ) ;

    OPERATION_CONTROL_STRUCTURES

      OPERATION check ( road_name  : IN traffic_lights.road ;
                        is_present : IN OUT present )
        DESCRIPTION
          --{Operation check reads the hardware sensors for the road given in
             the parameter road_name to find out if traffic is present on
             either side, and returns the value is_present set to TRUE or
             FALSE.}--
        USED_OPERATIONS
          read_sensor ( sensor : IN latch ) RETURN present ;
      END_OPERATION check

      OPERATION read_sensor ( sensor : IN latch ) RETURN present
        DESCRIPTION
          --{Operation read_sensor reads a hardware sensor at the given sensor
             latch, and returns the value TRUE or FALSE.}--
      END_OPERATION read_sensor

END_OBJECT traffic_sensors
```

Finally the design of the environmental object `text_io` is below, recall that such objects have nothing except a **provided_interface**.

```
OBJECT text_io IS ENVIRONMENT PASSIVE
  PROVIDED_INTERFACE
    TYPES
      string ;
    OPERATIONS
      put_line ( item : IN string ) ;  -
END_OBJECT text_io
```

3.7 Unused HOOD Facilities

As mentioned in the introduction to this chapter, some features of HOOD were not used in this thesis. This section briefly outlines these unused features, and explains why they were omitted.

Generic Classes in HOOD allow the creation of Ada generic objects, which can then be instantiated to form objects. Their use in HOOD is not very common, and how this kind of information should be handled in our complexity measure is far from clear.

Data Flows show the flow of information between objects. They are optional, and are used to document information already captured in the details of the ODS.

Exceptions Flows like data flows are an optional documentation aid. Note however that we do (in principle) process exception handling information, which identifies which operation raises the exception, and which operation is prepared to handle it.

Active Objects are (in principle) treated just like passive objects.

Interrupts are currently ignored.

Virtual Nodes allow a HOOD design to be distributed amongst a number of logical nodes which in due course are mapped to another (smaller) set of physical nodes.

3.8 Rationale for Choosing HOOD

This section examines why we choose HOOD.

The design method we have chosen to use as the basis of this thesis is HOOD which supports decomposition, reuse and object oriented design (at a simple level). A HOOD design is primarily architectural and in particular identifies the services needed to support an object. This is not unique, MASCOT supported similar ideas. It should be apparent that such a system permits the easy interchange of components provided the interfaces are kept the same. This helps testing and integration. More importantly the HOOD method can be applied opportunistically to individual objects as and when they are identified. The benefits of a well-defined interface have been highlighted by work on Abstract Data Types and Object Oriented Design. Such interface definition is fostered by HOOD. It should further be apparent that HOOD makes (nearly) impossible the worse forms of coupling.

3.9 Augmented HOOD

In order to fully capture all the architectural information in a HOOD design, we have slightly extended the textual representation. These extensions are fully described in Section 7.1, along with the rationale for their introduction. Appendix A details the precise syntax changes to the input language's BNF.

3.10 Further Reading

The HOOD Reference Manual (Delatte et al., 1993), provides the definitive definition of the HOOD language. The more recent edition (HOOD HRM, 1995) also covers HOOD 4. For a more tutorial introduction Robinson (1992a) provides a description of HOOD, and the evolution of object-oriented design. The HOOD User Manual (HOOD HUM, 1996) presents the definitive treatment on using HOOD in real-world projects. The Ada programming language is defined in Ichbiah et al. (1983).

When we try to pick out anything by itself,
we find it is tied to everything else in the universe.

JOHN MUIR (1838–1914)
U.S. naturalist, explorer

Chapter 4

Complexity Measures

Synopsis

This chapter provides a critique of previous work on measuring software design complexity. The advantage of a design metric is that it can be used early in the production process to identify potential trouble spots thereby reducing the costs of production. Design metrics are therefore required to detect over complex objects. We conclude the chapter by considering ways to validate proposed complexity measures.

In Chapter 2, we examined the problems associated with capturing designs, in terms of design as an activity, the effects of different design notations and the different design methodologies. In Chapter 3, we introduced HOOD as our chosen design notation and method for reducing complexity. The Chapter concluded by demonstrating that HOOD is a reasonable choice in the light of Chapter 2. Before developing our model further this chapter provides a critical review of related work in determining design complexity.

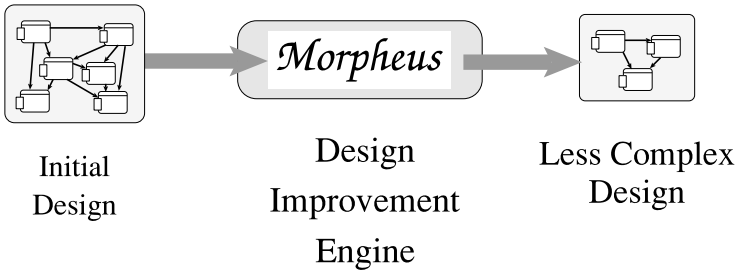


Figure 4.1: Overview of *Morpheus*

In order to provide a relevant critique of related work, we must be clear about what our system, *Morpheus*, is intended to achieve. *Morpheus*'s basic intention is simple (see Figure 4.1), "given an initial *closed* design, *Morpheus* seeks a structurally less complex (alternative) design. *Morpheus* takes the lowest level connections formed by the designer as *fixed* and manipulates the design's modular structure." It immediately follows that we need a systematic method for comparing two designs, and a method for producing alternatives. However, not only do we need a method for comparing two designs, we need some basis for believing that the results of our comparisons are reasonable, otherwise we could just pick "the best" design by some random process. However we want a repeatable process, we do not want one based on whim or *magic*.

4.1 Requirements for a Complexity Measure

Based on the above overview, we want our complexity measure to satisfy the following requirements

- It must be possible to evaluate a design's complexity without reference to its implementation.
- Complexity must be rigorously defined, so that its definition is unambiguous.
- Complexity must be computable for all possible designs.
- Complexity must be automatically computable from the design, when expressed in some suitable notation.
- The process must be repeatable.
- There must exist a *weak ordering* amongst all designs, i.e., for any two arbitrary designs A and B (say), it must be possible to determine if $A \succeq B$ or $B \succeq A$, where $A \succeq B$ implies that A is more complex than B.
- The complexity measure should capture notions of good software engineering. In particular, obviously poorly structured designs should have a higher complexity than well structured designs.
- The complexity measure should permit trade-offs between engineering attributes, for example coupling and cohesion, in a hierarchical framework.

4.2 Existing Complexity Metrics

As we saw in Chapter 1, we choose structural complexity as the design property that *Morpheus* should aim to improve. There are clearly other possible choices, for instance just the size of the design, the coherence of individual objects or just the coupling between objects. However, these attributes, whilst all individually important do not capture the essential nature of the trade-off between all these properties that a good design must encompass. As we shall show in Section 6.3, our proposed complexity measure permits this trade-off to occur.

The majority of the complexity measures proposed in the literature are not suitable for our needs. The following sections will briefly examine a number of proposed metrics and explain why they were rejected.

4.2.1 Program Complexity Metrics

Probably, the first attempt at a measure of program complexity was the number of source lines of code (LOC). Whilst such a measure might be naively appealing, it fails to meet our requirements on several counts:

- The required information becomes available only after the code has been written, which is too late in the production process to be useful in assessing the quality of the design before the next phase of development.
- LOC are difficult to define rigorously, and are very responsive to different styles and hence yield different values, (for example unrolling a simple iterative loop). Very similar designs may yield significantly different values.
- LOC take no account of software engineering practice, for example the distribution and size of objects. Indeed adding a modular structure could make the LOC increase!

- LOC tells us nothing about how to make complex designs less complex.

We have been very critical of LOC, perhaps unfairly as this measure was never meant to reflect design quality or complexity. However, this analysis does serve to lay a framework for discussing other proposed program complexity measures.

The most notable code metrics are Software Science (Halstead, 1977) and Cyclomatic Complexity (McCabe, 1976). Both have been quite well researched; and were initially regarded quite favourably, but more recently their theoretical underpinnings have been shown to be weak (see Shepperd and Ince, 1993, p.28–40).

Since software engineers use such a wide variety of notations, some researchers have tried to extract design information from the resulting program code rather than the design (Shepperd, 1993, p.8), but as Shepperd comments "this must be considered a last resort". The problem is that the information is available so late and furthermore the code implementation may have an impact on what *exactly* is measured.

Clearly, due to their late availability and doubts over their value as complexity metrics, code metrics are unsuitable for our purposes. So we shall now look at some of the proposed design metrics.

4.2.2 Design Metrics

A number of design metrics have been proposed, for example Embley and Woodfield (1987), Embley and Woodfield (1988), McCabe and Butler (1989), Rotenstreich (1994), Bieman and Ott (1994) and Chidamber and Kemerer (1994). However, most of these measures fail to satisfy our requirements for a complexity measure. The obvious failures are those metrics which only use a singe factor as the determinant of complexity, such as coupling or cohesion. These measures provide no means of assessing the relative merit of *both* cohesion and coupling. In a similar way we can reject measures based solely on size.

Another widely used metric is based on data flow (exemplified by Henry and Kafura (1993)). We have two principal objections to this approach: Firstly, information flow is being used as a proxy for coupling with length serving as a proxy for the internal complexity of subcomponents without any obvious justification for such an assumption. Secondly, Henry and Kafura compute information flow based on

$$\text{length} \times (\text{fan\_in} \times \text{fan\_out})^2$$

without any adequate explanation of why this should compute information flow (see Shepperd and Ince, 1993, p.41–50). Interestingly however, Henry and Kafura (1993, p.105–110), do apply their metric to find a better abstraction model for part of the UNIX system hierarchy; but their method is not adequately defined and in our view still seems to rely on intuition.

Shepperd and Ince (1993, p.119–133) proposed an interesting alternative called a 'work' metric based on object requirements. The metric tries to measure the work done by each object in a system. Their contribution is important since it is based on a formal algebraic model and has been subjected to a theoretical validation. However, we reject it because the metric requires the establishment of a mapping from functional requirements (derived from the requirement specification) onto individual objects. This process cannot (yet) be carried out automatically and it is unlikely that different engineers would agree on *a unique object*, especially for 'large' requirements. Further this mapping would be particularly prone to object restructuring and changes in the requirements.

Another interesting idea is the use of cluster analysis to group similar objects together, see for example Hutchens and Basili (1993) and Neil and Bache (1993). This is an idea which we find intuitively appealing. However as Shepperd (1993, p.9) comments "the greatest stumbling block is that the shape of the ... dendrograms is highly dependent upon the choice of cluster algorithm". This was confirmed by our own experiments. We found the situation to be much worse than

Shepperd reports because distance metrics which yielded intuitive results with one design failed to produce acceptable results for other examples. Even small changes to a design could make the resultant dendrogram unappealing to our intuitive notions of a good design. We conclude therefore that this approach was unsuitable for our purposes.

4.2.3 Object Oriented Design Metrics

As explained earlier this thesis is not based on object oriented design but rather object based design concepts, but given the current interest in the work of Chidamber and Kemerer (1994), we deal briefly with this subject. Chidamber and Kemerer proposed a set of six metrics for measuring a variety of attributes of object-oriented systems (by examining the program code). These attributes are: weighted methods per class, depth of inheritance tree, number of children of a class, coupling between object classes, response for a class (i.e., the number of methods potentially called by a class) and lack of cohesion in methods.

Their work, which has become a *de facto* standard for object-oriented metrics, includes a philosophical basis and theoretical validation against the Weyuker (1988) property set for complexity measures. However Chidamber and Kemerer offer no method for trading between the measured attributes, for example coupling and cohesion. Churcher and Shepperd (1995) have also observed that Chidamber and Kemerer definitions need to be made more precise in the light of differences between languages—so that cross comparisons amongst different work can be carried out. Briand et al. (1996) also show that Chidamber and Kemerer metrics do not satisfy their proposed requirements for complexity metrics. However, Chidamber and Kemerer never claim that their metrics were intended to be complexity measures.

4.2.4 Information Theory and Design Metrics

There have been a few measures of software complexity based on information theory. Khoshgoftaar and Allen (1994) survey information theory and software metrics. The following section is derived from their survey findings.

Mohanty (1981) uses a measure of excess entropy[1] to study the information shared between objects. Mohanty regarded this as a measure of interface complexity, but Khoshgoftaar and Allen see this as a measure of object coupling. Whatever Mohanty is measuring, his approach does not offer any form of trade-off between object properties.

Lew et al. (1988) take measurements of several different kinds of connectivity between objects, based on message type (control or data) and the static structure of the exchanged data types, to produce three different entropy measures. These measures are then combined into a single measure of complexity. Lew et al.'s use of distinct measures for different design attributes reflects their different role in a design, but forming a single measure from unrelated sources seems unjustified.

Harrison (1992) proposed a complexity measure based on measuring the entropy of a program in terms of used operations. Harrison's approach is similar in nature to Halstead's Software Sciences and suffers from the obvious problem of being code based rather than design based. However, Harrison did validate his proposed metric against Weyuker's property set, and showed that it should be considered as a contender for measuring complexity. Harrison's metric is quite closely related to our proposed metric (for a given graph), but unfortunately uses out-degree rather than (total-)degree for each node. Furthermore, Harrison does not extend his metric to handle hierarchical structures.

These uses of information theory based on entropy clearly have the advantage of moving towards an objective basis, i.e., the information content of a system. However, they offer no solid basis for claims to measure complexity.

[1] Based on a feature signature of each object, which serves as a form of feature measure between objects.

4.3 Combining Different Measures

We have already observed that design involves trade-offs between different attributes, for example coupling and cohesion. Strictly speaking, however, we cannot make these comparisons, because the two attributes are measured on different scales in different dimensions. Some researchers leave these comparisons to the individual engineer's judgement (Shepperd and Ince, 1993, p.134–137). Others have applied some form of multi-argument function (e.g., Hops and Sherif (1995)) to compute a single value.

Both approaches raise the question of whether or not this is a valid way to perform trade-offs based on different scales. The essential problem is that the various dimensions are measured in isolation, rather than having a single unified basis. We shall argue in Section 6.3 that our proposed complexity measure involves elements of coupling, cohesion and size. But since our model derives a single measurement from a unified approach, trade-offs are not performed in some *ad hoc* manner. Indeed with our proposed complexity metric, one cannot isolate these individual attributes.

4.4 The Dual Problem: Reverse Engineering

One of our requirements for constructing *Morpheus* is a method for deciding which entities should be placed in a particular object and how objects should be nested. Whilst, metrics generally aim to provide a test for object structure, they do not provide any insight on how to grow objects. Reverse engineering provides a useful insight into this problem.

Calliss (1989) proposed three methods for identifying objects from the raw source code, namely: grouping by type-families, grouping by imports and grouping by state variables. We find this idea intuitively appealing as it accords with our intuition on one of the ways in which we look at designs to see if proposed objects are reasonable. However, two areas are not covered in Calliss's work. First how to determine which possible set of groupings gives the best objects, and second, how to find the next and subsequent layers of the resulting structure. Calliss's work was based on what we have termed design entities.

Choi and Scacchi (1990) use the files containing source code as the basis for identifying objects and they then introduce an algorithm for finding the system's higher structure. Anecdotal evidence (Howell, 1996) suggests that the content of a source file may have more to do with the programming language being used than with any notions of object structure, so source files may not be the best indication of the system's modular structure. Choi and Scacchi's algorithm seeks to minimise the impact of potential changes to any part of the design, determined by how many objects are visible to a higher level (controlling) object and coupling between objects. The drawback with Choi and Scacchi's approach is the apparent mixing of different measures without any rigorous justification.

Benedusi et al. (1992) presents a method for reverse engineering source code to produce design documents and structure charts. The system uses an algebraic system for abstracting away from the code detail to a design. Attention is focused on the low-level code detail and is more akin to the program restructuring school (e.g., Rich and Waters, 1990).

4.5 Towards a Merit function

Müller et al. (1993) describes a method for building systems from the lowest level upwards by examining coupling a lower levels. This section outlines their method.

A graph $G(\mathcal{N}, \mathcal{E})$ consists of a set of nodes, $\mathcal{N}$, (representing the software building blocks) and a set of edges, $\mathcal{E}$, (representing the connectivity between the nodes). Each edge is an ordered pair (v, w) representing the flow of resources from v to w, further each edge has an associated edge capacity, EC, representing the set of resources flowing from v to w. A resource is a set of syntactic

entities, e.g., procedures, types and variables. There is no reason why both the edges (v,w) and (w,v) should not exist in the same graph.

A node v will in general require a set of *requisitions*, $\mathrm{Req}(v)$, and offer a set of *provisions*, $\mathrm{Prv}(v)$, algebraically,

$$\mathrm{Prv}(v) = \bigcup_{x \in V} \mathrm{EC}(v,x)$$
$$\mathrm{Req}(v) = \bigcup_{x \in V} \mathrm{EC}(x,v)$$

However, most languages do not offer precise control over imports and exports, so Müller et al. defines *exact requisitions*, $\mathrm{ER}(v,w)$, (of v from w) and *exact provisions*, $\mathrm{EP}(v,w)$, (of v to w) between nodes, which can be calculated as below

$$\mathrm{ER}(v,w) = \mathrm{Req}(v) \cap \mathrm{Prv}(w)$$
$$\mathrm{EP}(v,w) = \mathrm{Prv}(v) \cap \mathrm{Req}(w)$$

Having defined exact provisions and exact requirements, Müller et al. now defines a measure of *interconnection strength*, $\mathrm{IS}(v,w)$, as the exact number of resources flowing between the two nodes v and w as

$$\mathrm{IS}(v,w) = |\mathrm{ER}(v,w)| + |\mathrm{EP}(v,w)|$$

Their system provides controllable high coupling threshold T_h and low coupled threshold T_l. These thresholds are adjusted to look for modules which are highly coupled relative to their neighbours. The idea being that if two or more modules are highly coupled, they should be merged to form a cohesive (super-)module. Interconnection strength as defined above relates two individual objects together, and does not yield a more general measure of the coupling within the system. Müller et al. then defines the coupling between an object and the rest of the system as *system strength*, SS, by summing over all other system objects, i.e.,

$$\mathrm{SS}(v) = \sum_{x \in V} |\mathrm{ER}(v,x)| + |\mathrm{EP}(v,x)|$$

Müller et al. suggest searching for objects with strong coupling and grouping these objects together to form a high cohesion subsystem. The resultant (super-)object would have some cohesion because of the existing coupling between objects. Hence Müller et al. clearly regard coupling and cohesion as just dual aspects of the same phenomenon.

We believe that Müller et al.'s work provides a very interesting foundation for our work, since it is based on identifying the specific interface between objects. Whilst Müller et al. uses a simple count for determining the size of the interface between objects, we use a statistical method which takes account of the relative importance of each node. One objection to Müller et al.'s work is that they force their systems to be $(k,2)$ partite graphs,[2] and we see no reason for this assumption. There is no reason to suppose that once an object has been decomposed into smaller (internal) entities that these entities should be in any way related to other internal entities from another object.

4.6 Validating Complexity Measures

Having proposed any kind of complexity measure, we need to validate the measure to ensure that it effectively performs its intended function. Validation should be both theoretical and empirical. Unfortunately, due to time constraints and the unwillingness of industry to allow access to

[2] A $(k,2)$ partite graph consists of a series of graph layers $G_1, \ldots, G_n$. Layers are connected by means of *vertical edges*; however vertical edges may only connect adjacent layers. Moreover the number of nodes per layer is bounded by k (paraphrased from Müller et al., 1993, p.186).

their designs, we have been unable to carry out any empirical validation. However, we have conducted a number of informal experiments (see Chapter 8) and we are satisfied that the measure is reasonable.

Weyuker (1988) proposed a set of nine properties which any measures of *program complexity* should satisfy. Her proposal is widely accepted (Shepperd and Ince, 1993) as a basis for theoretical validation, although it has some shortcomings. For example Fenton (1994) argues that two of Weyuker's properties (i.e., W-Property 6.5 and 6.6, see Chapter 6) capture different notions of complexity versus comprehension. However, we do not see why different members of a set of properties should not try to capture different aspects of a relationship. We would be concerned if the property set was internally inconsistent.

Briand et al. (1996) proposed a set of five properties for a *design complexity* measure. Like Shepperd and Ince, Briand et al. partly rephrased Weyuker's properties in terms of design complexity. Briand et al. concluded that their proposed property set did not contradict Weyuker's set. However, Briand et al.'s properties assume that complexity has a strictly additive nature, and we see no reason *a priori* to accept such a hypothesis. Therefore our theoretical validation is based on Weyuker's property set, restated for design complexity rather than code complexity.

I often say that when you can measure what you are speaking about,
and express it in numbers, you know something about it;
but when you cannot measure it,
when you cannot express it in numbers,
your knowledge is of a meager and unsatisfactory kind.

LORD KELVIN (1824–1907)
English physicist and mathematician

Part II

Theory

Chapter 5

Mathematical Background

Synopsis

This chapter provides the necessary mathematical background for understanding the calculation, derivation and theoretical validation of our proposed complexity measure.

This chapter formally defines a number of mathematical definitions required for the understanding of our complexity measure, Ψ, defined in Chapter 6. Some standard results are stated, without proof (references are provided), which are useful in the theoretical validation of our complexity measure.

5.1 Graph Theory

The building blocks of basic graph theory are *sets* (see for example Denvir (1986, Chapters 2 and 4) and Carré (1979, p.1–7)), and *relations* (see Denvir (1986, Chapter 7) and Carré (1979, p.17–23)). The reader unfamiliar with these mathematical concepts may wish to refer to the cited references before continuing. This thesis uses standard set notation, which is summarised in Appendix D.

Unfortunately, the graph theoretic literature has not yet evolved a definitive terminology; so we shall start by defining some basic terms. The following definitions are derived from Carré (1979) and Munro (1992).

5.1.1 Basic Terms

Definition 5.1 (Graph). *A* graph*, $G(\mathcal{N}, \mathcal{E})$, is a finite set of* nodes*, $\mathcal{N}$, together with a subset, $\mathcal{E}$, of the Cartesian product $\mathcal{N} \times \mathcal{N}$. The elements of the set $\mathcal{E}$ are called* edges*.*

The order of the elements in the ordered pair representing an edge is significant. Such graphs are sometimes referred to as or *directed graphs* or *digraphs*.

Definition 5.2 (Multi-graph). *A* multi-graph*, $G^*(\mathcal{N}, \mathcal{E}^*)$ is a finite set of* nodes*, $\mathcal{N}$, together with a finite bag, $\mathcal{E}^*$, of the Cartesian product $\mathcal{N} \times \mathcal{N}$. The elements of the bag $\mathcal{E}^*$ are called* edges*.*

Generally in this thesis we only use graphs, rather than multi-graphs. When multi-graphs are required, we shall explicitly say multi-graph.

It is often convenient to represent a graph visually as for example in Figure 5.1. The arrows are used to show the direction of each edge. The set of nodes $\mathcal{N}$ is $\{w, x, y, z\}$, and the set of edges $\mathcal{E}$ is $\{(w,x), (w,z), (z,z), (z,y), (y,x), (z,x), (x,y)\}$. However, it must be stressed that such pictures are only for ease of understanding, it is the two sets, $\mathcal{N}$ and $\mathcal{E}$, that define the graph.

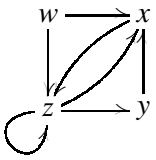


Figure 5.1: A graph (Ross and Moore, 1995)

Definition 5.3 (Predecessor). *In a graph $G(\mathcal{N}, \mathcal{E})$, a node n_i is called a* predecessor *of a node n_j if $(n_i, n_j) \in \mathcal{E}$.*

Definition 5.4 (Initial and Terminal endpoints). *The two nodes in an ordered pair representing an edge, (n_i, n_j), are respectively known as the* initial *endpoint and the* terminal *endpoint of the edge.*

Definition 5.5 (Degree). *An edge, (n_i, n_j), is said to be* incident from node n_i *and* incident to node n_j*. The sum of number of edges incident to and from a node is called the* degree *of the node.*

For example in Figure 5.1, the node w has degree 2. In the case of a *loop*, such as the edge (z, z), the edge counts twice, so that node z has degree 6.

Theorem 5.1. *Let $G(\mathcal{N}, \mathcal{E})$ be a (multi-)graph with n nodes and e edges. Let node n_i have degree d_i for $i = 1, \ldots, n$. Then*

$$\sum_{i=1}^{n} d_i = 2e$$

Proof. See Munro (1992, p.160). □

Definition 5.6 (Paths and Cycles). *A* path *is a finite sequence of edges of the form*

$$(n_{i_0}, n_{i_1}), (n_{i_1}, n_{i_2}), \ldots, (n_{i_{r-1}}, n_{i_r}),$$

i.e., a finite sequence of edges in which the terminal node of each edge coincides with the initial node of the following edge. A path such that the two endpoints (i.e., n_{i_0} and n_{i_r}) coincide is said to be a cycle.

Since a path is uniquely determined by the sequence of nodes $n_{i_0}, n_{i_1}, \ldots, n_{i_r}$ which it visits; it is often convenient to just specify a path by listing the required node sequence.

For example in Figure 5.1, the path z, y, x, z is a cycle. Note also, that there is no path to node w, since no edge has node w as its terminal endpoint.

Definition 5.7 (Acyclic). *A graph is called* acyclic *if is does not contain any cycles.*

Definition 5.8 (Simplification). *The* simplification *of a digraph $G(\mathcal{N}, \mathcal{E})$ is a graph $G_s(\mathcal{N}_s, \mathcal{E}_s)$, in which each edge in $\mathcal{E}$ is replaced by two edges. One edge being the original (n_i, n_j) and the other formed by swapping the order of the two endpoints, i.e., (n_j, n_i).*

Mentally, it is much easier to consider $G_s(\mathcal{N}_s, \mathcal{E}_s)$ as being the original graph $G(\mathcal{N}, \mathcal{E})$ but without any restriction on the direction of edges.

Definition 5.9 (Connections). *Two nodes n_i and n_j in a graph $G(\mathcal{N}, \mathcal{E})$ are* connected *if there is a path from n_i to n_j on the simplification G_s of G.*

Clearly, if such a path exists, there is also a path from n_j to n_i in G_s.

Definition 5.10 (Components). *The set of nodes connected to an arbitrary node is called a* component *of the graph.*

The components of a graph are obviously disjoint, and hence form a partition of the graph.

Definition 5.11 (Connected Graph). *A graph with exactly one component is called a* connected graph.

Definition 5.12 (Trees). *A* tree *is an acyclic graph $G(\mathcal{N}, \mathcal{E})$ in which one node n_r has no predecessors and every other node has exactly one predecessor. The node n_r is called the* root *of the tree. A set of trees is called a* forest.

It follows from the above definition that a tree is a connected graph; and that a forest is not connected.

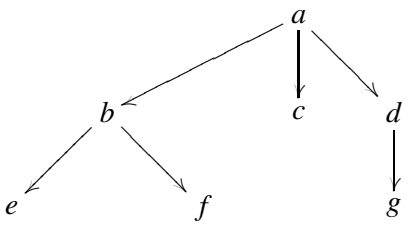


Figure 5.2: A tree

Figure 5.2 shows an example of a tree. The root of this tree is the node *a*. It is frequently convenient to draw trees with the root at the top of the page, as in our example. In such a diagram, it is often convenient to leave out the arrow-heads; there is no ambiguity since parent nodes are shown above their respective children.

5.1.2 Operations on Graphs

Definition 5.13 (Strict Graph Equality). *Two graphs, $G_1(\mathcal{N}_1, \mathcal{E}_1)$ and $G_2(\mathcal{N}_2, \mathcal{E}_2)$, are* equal *if $\mathcal{N}_1 = \mathcal{N}_2$ and $\mathcal{E}_1 = \mathcal{E}_2$.*

Sometimes, it is convenient for an edge to have a *label*, in which case the representation of an edge becomes an ordered tuple (edge_i, edge_j, label_k). In this case two edges are only equal if they have the same tuple. Labels can be drawn from any desired set, and have no relation with the node set.[1]

Definition 5.14 (Subgraph). *A graph $G_1(\mathcal{N}_1, \mathcal{E}_1)$ is a* subgraph *of a graph $G_2(\mathcal{N}_2, \mathcal{E}_2)$, if $\mathcal{N}_1$ is a subset of $\mathcal{N}_2$ and $\mathcal{E}_1$ is a subset of $\mathcal{E}_2$. We use the notation $G_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubseteq G_2(\mathcal{N}_2, \mathcal{E}_2)$ to represent such a relationship.*

Since the subgraph relation is based on the subset of the underlying sets, it follows that a graph is a subgraph of itself.

Definition 5.15 (Proper Subgraph). *A* proper subgraph *can be defined in terms of the subgraph relation above, by adding the requirement that at least one of the set of nodes or edges is a proper subset. We use the notation $G_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubset G_2(\mathcal{N}_2, \mathcal{E}_2)$ to represent this relationship.*

The following definitions of graph union and graph intersection are adapted from Calliss (1989).

Graphs can be combined by *graph union* based on set union of the underlying sets. Figure 5.3 shows two graphs, and Figure 5.4 shows the resultant graph union. Formally, we have the following definition.

Definition 5.16 (Graph union). *The* graph union *of two graphs $G_1(\mathcal{N}_1, \mathcal{E}_1)$ and $G_2(\mathcal{N}_2, \mathcal{E}_2)$ is a third graph $G_u(\mathcal{N}_u, \mathcal{E}_u)$, symbolically $G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcup G_2(\mathcal{N}_2, \mathcal{E}_2)$. The set $\mathcal{N}_u$ is given by $\mathcal{N}_1 \cup \mathcal{N}_2$, and the set $\mathcal{E}_u$ by $\mathcal{E}_1 \cup \mathcal{E}_2$.*

[1] Nodes may be labelled in a similar way, but this extension is not needed in this work.

(a) $G_a(\mathcal{N}_a, \mathcal{E}_a)$ (b) $G_b(\mathcal{N}_b, \mathcal{E}_b)$

Figure 5.3: Two graphs

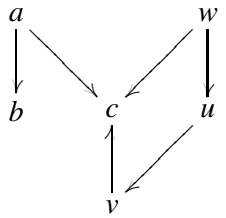


Figure 5.4: Graph union

The example in Figure 5.3, has two nodes $\{c, v\}$ and an edge (v, c) in common. If one graph had instead had an edge (c, v), there would have been two edges (v, c) and (c, v) between nodes $\{c, v\}$ in the resulting graph.

If the two graphs being combined have no nodes in common, graph union still yields a single graph, consisting of at least two distinct components.

Figure 5.5: Graph intersection

Graphs can also be combined by *graph intersection*, again based on the underlying set operation, intersection. Again considering the two graphs in Figure 5.3, the result of graph intersection is shown in Figure 5.5. Formally, graph intersection is defined as:

Definition 5.17 (Graph intersection). *The* graph intersection *of two graphs* $G_1(\mathcal{N}_1, \mathcal{E}_1)$ *and* $G_2(\mathcal{N}_2, \mathcal{E}_2)$ *is a third graph* $G_i(\mathcal{N}_i, \mathcal{E}_i)$*, symbolically* $G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap G_2(\mathcal{N}_2, \mathcal{E}_2)$*. The set* $\mathcal{N}_i$ *is given by* $\mathcal{N}_1 \cap \mathcal{N}_2$*, and the set* $\mathcal{E}_i$ *by* $\mathcal{E}_1 \cap \mathcal{E}_2$*.*

The definitions of graph union and intersection, can be extended to encompass labelled graphs in the obvious manner.

We can construct a more general definition of graph equality than Definition 5.13, by replacing the strict set equality used there by requiring the sets to be isomorphic in the following way.

Definition 5.18 (Isomorphic Graph Equality). *Two graphs,* $G_1(\mathcal{N}_1, \mathcal{E}_1)$ *and* $G_2(\mathcal{N}_2, \mathcal{E}_2)$*, are* equal *if there exists a bijective mapping* f *from* $\mathcal{N}_1$ *to* $\mathcal{N}_2$ *and another bijective mapping* g *from* $\mathcal{E}_1$ *to* $\mathcal{E}_2$*, such that*

$$g\left(e_{1i}, e_{1j}\right) = \left(f(e_{1i}), f(e_{1j})\right) = \left(e_{2i'}, e_{2j'}\right) \quad \forall e_{1i} \in \mathcal{E}_1 \text{ and } \forall e_{2i} \in \mathcal{E}_2$$

and

$$g^{-1}\left(e_{2i}, e_{2j}\right) = \left(f^{-1}(e_{2i}), f^{-1}(e_{2j})\right) = \left(e_{1i'}, e_{1j'}\right) \quad \forall e_{1i} \in \mathcal{E}_1 \text{ and } \forall e_{2i} \in \mathcal{E}_2$$

The identity function obviously satisfies the requirements on f and g, thus yielding the previous definition as a special case. It is this isomorphic equality between two graphs which allows us to validly draw pictures of graphs and represent graphs as data structures in *Morpheus*. Obviously, the definitions of graph union and intersection (above) can be generalised in this framework.

5.1.3 Simple Design Graphs

Before progressing on to look at hierarchical graphs (see Section 5.1.4) we need to pause and consider how a software design can be represented by what we shall call *design graphs*.

Consider the outline definition of the single object shown below.

```
OBJECT a_stack IS PASSIVE
    PROVIDED_INTERFACE
        OPERATIONS
            push ( datum : IN int ) ;
            pop  RETURN int ;

    INTERNALS
        TYPES
            stack ;
        OPERATIONS
            push ( datum : IN int ) ;
            pop  RETURN int ;
        DATA
            my_stack : stack ;

        OPERATION_CONTROL_STRUCTURES
            ...
END_OBJECT a_stack
```

The construction of `a_stack` is not visible outside the enclosing object. A user of this object can only see the provided operations, namely: `push` and `pop`. This object can be used to store a sequence of integers in a last-in first-out order. We assume that the surrounding environment will supply a suitable object with integers without being explicitly referenced. Figure 5.6 shows the design graph generated from this design. Operations are indicated by ovals, variables are shown by boxes, whilst datatypes are shown as trapeziums. The arrows from `push` to `int` indicate that the operation `push` requires 'the services' of the datatype `int`, so that `push` can provide its services to others. Similarly `push` needs the variable `my_stack` for storing the sequence of integers, and the datatype `stack` in order to manipulate `my_stack` This concept is similar to the resource flow graph of Müller et al. (1993) and others.

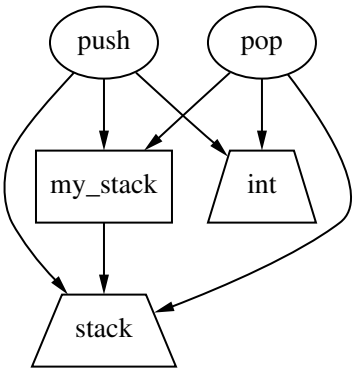


Figure 5.6: Design Graph of Simple Stack

Such a model is fine for 'flat' software architectures, but is not sufficient for true hierarchical designs.

5.1.4 Hierarchical Graphs

The previous sections described standard 'flat' graphs, that is every node is just an element of some set. In this section, we introduce the concept of *hierarchical graphs* or *nested graphs*. In a hierarchical graph, a node may itself expand to contain further nodes and edges, and so on *ad infinitum*.

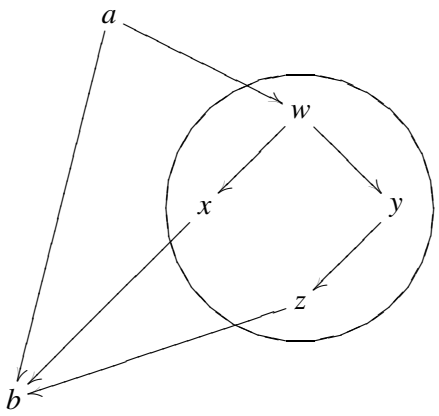


Figure 5.7: A hierarchical graph

Figure 5.7 shows an example of a hierarchical graph, and the nodes $\{w,x,y,z\}$ (inside the circle) nested inside the collapsed node. One might consider the top-level graph[2] to be that shown in Figure 5.8, where the node W represents the collapsed nodes $\{w,x,y,z\}$. However, a little reflection will show this view to be inadequate, because it has only a single link between the nodes W and b.

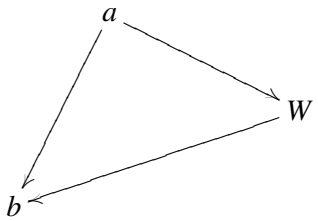


Figure 5.8: Top graph of the hierarchical graph

The subgraph consisting of the nodes $\{w,x,y,z\}$ is of course a graph in its own right. However, such a view loses the three edges (a,w), (x,b) and (z,b), which cross the nested graph's boundary. That is, in a nested graph, we potentially have edges which 'go nowhere'. An alternative view is to have two edges from W to b, however the number of such links may not always be known; nor is such a situation intuitively obvious. With such possible expansion of nodes in hierarchical graphs Theorem 5.1 does not hold.[3]

5.1.5 Full Design Graphs

Referring back to our stack example in Section 5.1.3. Now consider a typical use of a stack. The design might look something like

[2]Technically such a graph is a *condensation* of the nodes $\{w,x,y,z\}$, (see Carré, 1979, p.34).

[3]Proof, look at circled part of Figure 5.7, the problem is that there are edges which 'go nowhere' thus breaking the relationship between edges and degrees.

```
OBJECT stack_adt IS PASSIVE
    PROVIDED_INTERFACE
        OPERATIONS
            push ( datum : IN int ) ;
            pop  RETURN int ;

    INTERNALS
        TYPES
            stack ;
        OPERATIONS
            push ( datum : IN int ) ;
            pop  RETURN int ;
        DATA
            my_stack : stack ;

        OPERATION_CONTROL_STRUCTURES
            ...
END_OBJECT stack_adt



OBJECT caller IS PASSIVE
    PROVIDED_INTERFACE
        TYPES
            int ;
        OPERATIONS
            main ;

    REQUIRED_INTERFACE
        OBJECT stack_adt
            OPERATIONS
                push ( datum : IN int ) ;
                pop  RETURN int ;

    INTERNALS
        ...
END_OBJECT caller
```

The resulting design graph is shown in Figure 5.9. Such a graph actually represents *two distinct* pieces of information, namely: the modular structure (which is always a tree, see Figure 5.10) and the 'requires' relationship which can flow through object boundaries.

5.1.6 Design Graph Concatenation

The property set for judging complexity metrics proposed by Weyuker (1988) requires the definition of a concatenation operation for two designs. Clearly concatenating two designs is non-trivial. However, we must define at least a plausible systematic concatenation operation. We therefore define design concatenation on the basis of graph union, with the following added condition: If the underlying object hierarchy of the standard graph union has two (or more) top-level objects, a single unique object is added at the top of the hierarchy. This ensures we always have a single hierarchical tree.

It follows immediately, that concatenating a design with itself, yields the original design.

5.2 Information Theory

In Chapter 6, we will develop a method for encoding the structure of a hierarchical graph as a (binary) message. This section provides the background theory to explain how the length of such

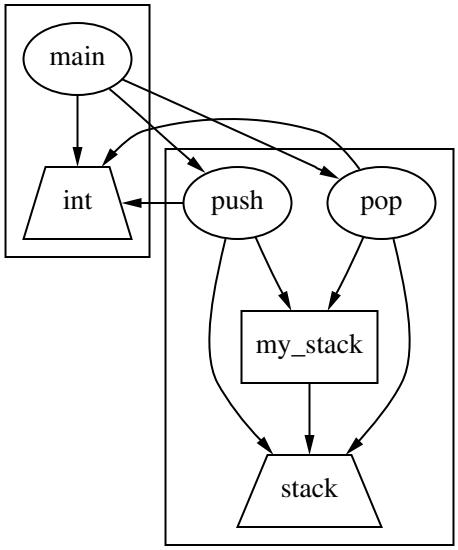


Figure 5.9: Design Graph of Stack ADT with Caller

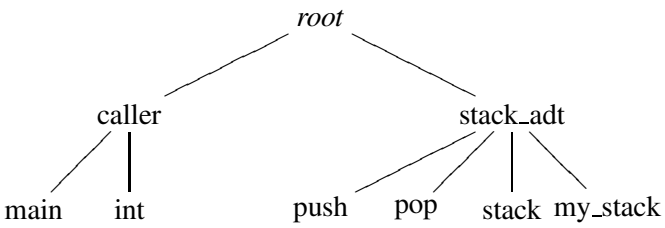


Figure 5.10: Object Structure of Stack ADT with Caller

messages can be calculated. However, we must first explain why the length of such a message is interesting.

The following sections are derived from Li and Vitányi (1997).

5.2.1 Kolmogorov Complexity

Intuitively, if something can be described with a shorter description than something else, we might suspect that the former was in some sense simpler than the latter. Of course, if the second description started from basic principles or used only a simple vocabulary, whilst the first was based on other complex descriptions or used a larger vocabulary, we might question our initial assumption. If, however, both descriptions started from the same baseline, our suspicions would be calmed. This concept of judging complexity based on the amount of information needed to describe something lies at the heart of Kolmogorov Complexity.

The foregoing appealed to intuition, we need a more rigorous basis if we are going to use this as a model of complexity. This concept of complexity can be made formal by insisting on using strings of binary digits and permitting no *a priori* knowledge. The description language is limited to the class of languages that can be processed by a Turing machine (Turing, 1937). This corresponds to the class of partial recursive functions. We then take the length of the *shortest possible* universal Turing machine program which provides the required description. This length is called the Kolmogorov Complexity of the item being described.

Unfortunately, such a function is noncomputable. Rissanen (1978) developed an approach for choosing between competing alternative hypothesises based on the ideas of Kolmogorov Complexity.

5.2.2 Minimum Description Length Principle

Definition 5.19 (Minimum Description Length (MDL) principle). *The best theory to explain a set of data is the one which minimizes the sum of:*

- *the length, in bits, of the description of the theory; and*
- *the length, in bits, of data when encoded with the help of the theory.*

Li and Vitányi (1993, p.308)

Given some data D, MDL states that we should pick that theory T which minimises

$$\text{length}(T) + \text{length}(D \mid T) \tag{5.1}$$

where $\text{length}(T)$ is the number of bits needed to minimally encode the theory T, and $\text{length}(D \mid T)$ is the number of bits needed to minimally encode the data D given the theory T. The resultant message must be decodable, otherwise we could just send 1 bit!

From Kolmogorov complexity theory (Li and Vitányi, 1997), we know that

$$\text{KC}(x) \sim \min_i \text{length}(x_i)$$

where $\text{KC}(x)$ is the Kolmogorov complexity of x. That is the Kolmogorov complexity of x is approximately the length of the minimum message describing x in bits.

This enables us to interpret the MDL principle in Bayesian terms (see Li and Vitányi, 1993, p.309). It can seen that minimising $\text{length}(T) + \text{length}(D|T)$ corresponds to maximising $\text{KC}(T) \times \text{KC}(D \mid T)$ and hence $\text{KC}(T \mid D)$. In practice we cannot enumerate all possible Ts, so we are limited to a set of possible Ts. In this thesis we limit ourselves to a class of preselected theory (see Section 6.2.3) and just transmit the graph's structure according to our chosen theory. A significant piece of future work will be the development of alternative theories.

5.2.3 Prefix Encoding of Positive Integers

In Chapter 6, we will see that part of our model for describing a graph, requires a method for transmitting positive integers between the message's sender and receiver. Since, *a priori* we cannot know the values of the required integers or the lengths of the corresponding binary string representations, we have to use a coding scheme which can unambiguously detect the end of the required integer. Coding schemes which satisfy this criteria are called *prefix codes*. The following two definitions are quoted from Jones (1979, p.45).

Definition 5.20 (Uniquely Decodable). *A code is uniquely decodable if, for each source sequence of finite length, the sequence of code letters does not coincide with the sequence of code letters for any other source sequence.*

Definition 5.21 (Prefix Codes). *A prefix condition code is one in which no code word is the prefix of any other code word.*

We concentrate on prefix codes because every uniquely decodable code can be replaced by an equivalent length prefix code (see Li and Vitányi, 1997, p.74).

To transmit a positive integer, so that it can be decoded, it is not possible to send just the integer's binary representation, because we cannot detect the end of the integer's representation. To overcome this problem; we transmit the length of the required binary string. This length information is encoded using a prefix scheme so that the end of the length field can be unambiguously detected. This is done by doubling up each digit of the length field and adding a single bit (0) at the end of the length field. This sentinel can be detected, since the binary representation of all positive integers start with 1. Hence, the end of the length field is uniquely terminated by the sequence '01', which of course cannot occur whilst doubling up the length's individual binary digits.

To make this clear, consider the transmission of the integer 27_{10}; this has a binary code of 11011_2, and the length of this binary code is clearly 5_{10} which in turn has a binary code of 101_2. Therefore, 27_{10} can be transmitted as

$$27_{10} \cong 11\ 00\ 11 \quad 0 \quad 11011$$

Where $\cong$ indicates that the right and left hand sides are different representations of the same number. The right hand side is the coded representation of the left hand side. Spaces have no significance, and are just added for clarity.

Clearly, the length of this representation is 12 bits. This length can be calculated as

$$2\lfloor \log_2 \lfloor \log_2 27 + 1 \rfloor + 1 \rfloor + 1 + \lfloor \log_2 27 + 1 \rfloor$$

We can approximate $\lfloor \log x + 1 \rfloor$ by $\log x$, with an absolute error of at most 1 bit.

This concept of prefixing an integer by its length, can of course be continued by prefixing the length with the length's (binary string) length, and so on until a length of exactly one bit is achieved.[4] Hence for example in the case of 27_{10} we have

$$27_{10} \cong 1\ 10\ 100 \quad 0 \quad 11011$$

This representation again has a length of 12 bits.

Hence in general we can represent an integer n by the code sequence

$$\mathcal{L}^* n = \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + \cdots$$

The logarithm sequence terminates when $\log(\ldots \log x)$ becomes less than 0.

Using Kraft's Theorem (Jones, 1979, p.48), we can deduce a lower bound on the length of a code for a given integer, represented by $\log_2^*$: where (see Li and Vitányi, 1993, p.75)

$$\begin{aligned} \log_2^* n &= \mathcal{L}^* n + \log_2 2.865064 \\ &= \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + \cdots \quad + \log_2 2.865064 \end{aligned}$$

This function is obviously strictly monotonically increasing. We note that $\log_2^* 27 \approx 9.918$ bits, to which our previous lengths of 12 bits compare quite favourably.

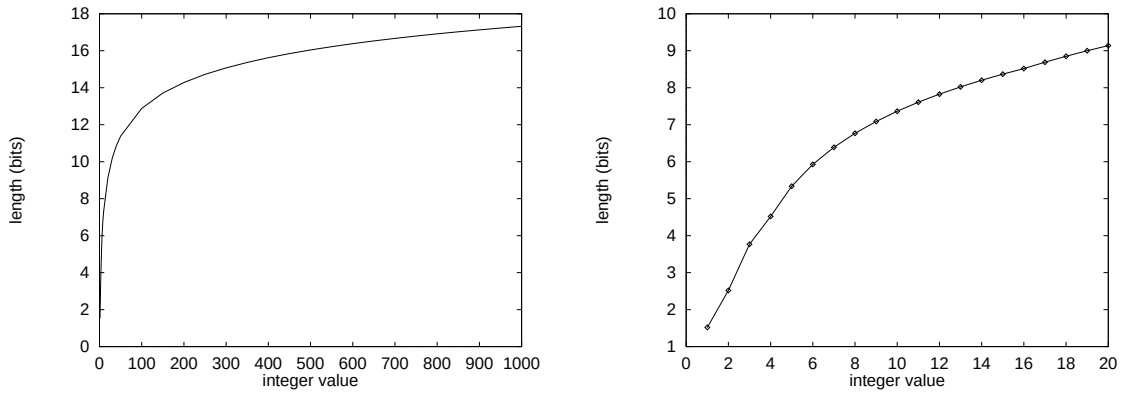
Finally, we note that to transmit the set of natural numbers including zero, it is only necessary for the sender and the receiver to agree that the sender shall always add one to the required value prior to encoding and transmission.

Broadly speaking a code is *universal* if it can be used to transmit a sufficiently long string without knowledge of the initial probability distribution of the source alphabet. Similarly, a code is *asymptotically optimal* if after some initial value the average code word length is minimised. $\mathcal{L}^*$ satisfies these requirements, (see Li and Vitányi, 1997, p.75–80 for details).

Figure 5.11(a) shows the graph of $\log_2^*$, as we can see the graph tails off slowly, as required for an asymptotically optimal code. However the initial segment climbs rapidly, see Figure 5.11(b). This may not be the *most appropriate* shape for a function encoding small integers which seem to dominate the entity counts of designs. The code, $\mathcal{L}^*$, is not the only asymptotically optimal universal code for integers, but "[$\mathcal{L}^*$] has had a virtual monopoly in MDL applications" (Baxter, 1996, p.81). There are many possible alternatives, some assuming more prior knowledge, (see Baxter, 1996, p.81–95). However as Baxter notes, we need to consider the choice of integer coding schemes:

> Although their code lengths converge asymptotically, feasible inductive inference is never done with an asymptotically large amount of data. So the difference in code lengths may affect the inference.
>
> Baxter (1996, p.83)

[4] Technical note. Since the function $\lfloor \log_2 n + 1 \rfloor$ has a fixed point at $n = 2$, we actually use 'length -1' in place of length. However, this change has no impact on the remainder of this thesis.



(a) Moderate Integers

(b) Small Integers

Figure 5.11: Graph of $\log_2^*$

5.2.4 Length of Code Words with Known Probabilities

In Chapter 6, we will see that we need to find the length of a message describing the structure of a graph. Since we are only interested in the structure, a flat (multi-)graph can be described by just transmitting the details of its edges, where a node is identified by some arbitrary symbol from a (finite) alphabet. Hence we need some way of estimating the minimum message length of a sequence of symbols drawn from a known frequency distribution.

For a message describing a (given) graph (as above), we can determine the *a priori* probability of a node being chosen at random from the node sequence describing the graph. Let such a node be n_i, and let $\Pr n_i$ be the probability of n_i being selected from the node sequence. For a practical example of how to describe a graph, see Section 6.2.1.

We need a way of estimating the corresponding minimum message length for such a node sequence. Shannon (1948) showed that the message length of a sequence of symbols chosen from a fixed finite alphabet is equal to the sum of the corresponding symbol length for each symbol, i.e.,

$$\text{length}(s_1 \ldots s_n) = \sum_i \text{length}\, s_i$$

Hence we need an estimate for the length of each symbol. Shannon further showed that the minimum length of such a symbol (from a finite alphabet) can be estimated by $-\log_2(\Pr s_i)$ where $\Pr s_i$ is the probability of the symbol s_i occurring.

Example Consider the message, *aabab*, consisting of just two distinct symbols a and b. The message has a total of five symbols, hence $\Pr a = \frac{3}{5}$ and $\Pr b = \frac{2}{5}$; since a occurs three times and b occurs twice. Hence the minimum length for representing a is 0.74 bits and b has a minimum length representation of 1.32 bits. Thus the total (minimum) length for this message is $3 \times 0.74 + 2 \times 1.32 = 4.86$ bits. Observe that if we had encoded a and b using the obvious code of $a = 0$ and $b = 1$ (say), then the length would have been $5 \times 1 = 5$ bits; this length is greater than Shannon's minimum message length.

It must be stressed that the above is a theoretical lower bound and that in general no practical codes achieve this limit. The current best is Arithmetic Coding (see Witten et al., 1987). Further it should be borne in mind that if the original *a priori* estimates of the probabilities are wrong, the resulting message length will be suboptimal (Oliver and Hand, 1994). In our case this problem does not arise with respect to edge endpoints because they are derived from the required graph. Therefore, this estimate serves our purpose.

A flat graph can be described by a sequence of edges;[5] each edge being described by *exactly*

[5]Plus some additional information for decoding which can be ignored for the present.

two nodes. For our purposes, an isolated node need not be described, since (by implication) it does not contribute anything to the system, otherwise it would be connected to another node.

Hence in our case, we can estimate the minimum message length of a graph's edge description, for the edge (n_{e_1}, n_{e_2}) by

$$-\sum_{e \in \mathcal{E}} (\log_2(\Pr n_{e_1}) + \log_2(\Pr n_{e_2}))$$

In our extended model for complexity (see Section 6.2.3) we represent the endpoint of an edge in a hierarchical graph by a sequence of node symbols. (How such a sequence of nodes is determined and how its last element is detected is explained in Chapter 6.) To find the (partial) minimum message length corresponding to a sequence of node symbols, we simply take the sum of the minimum length of each individual node's symbol in the sequence. For example if we have the node sequence $n_1 n_2 \ldots n_\xi$ this has a corresponding minimum message length of $-\left(\log_2 \Pr n_1 + \log_2 \Pr n_2 + \cdots + \log_2 \Pr n_\xi\right)$ bits.

5.2.5 Further Reading

Li and Vitányi (1997) is probably the definitive reference on Kolmogorov Complexity.

Each problem that I solved became a rule
which served afterwards to solve other problems.

RENÉ DESCARTES (1596–1650)
Discours de la Méthode (1637)

Chapter 6

Describing a Graph

Synopsis

This chapter presents our complexity measure, Ψ, in detail, explaining how it is calculated and why it is a complexity measure. We also show that Ψ satisfies Weyuker's (1988) proposed property set for complexity measures.

In this chapter we describe our complexity measure, Ψ, in detail, and show that it satisfies Weyuker (1988) proposed property set for complexity measures.

6.1 The Message Passing Metaphor

Our message passing paradigm is very simple. We imagine that we need to communicate the structure of a design graph between a transmitter and a receiver along a *perfect* transmission medium. That is, the receiver receives exactly what is transmitted, without any errors, duplication, or data loss. The receiver must be able to recreate an equivalent graph from the received message, plus knowledge of the message's structure.

We hypothesise that the length of the resultant message is a measure of the structural complexity of the proposed design. Further, by application of Occam's razor,[1] a smaller message length indicates a better design.

6.2 Ψ: The Complexity of a Design Graph

In this section we describe our proposed complexity measure. This is a recursive definition, so we start by describing a simplified base case, and building upwards.

6.2.1 Describing the Edges in a Graph

In this section, we are going to develop a partial message for describing the edges in a standard graph. We ignore (for now) issues of how this message might be decoded by the receiver. This is only done to simplify the exposition, a decodable message will be covered in Section 6.2.2.

Given a multi-graph $G^*(\mathcal{N}, \mathcal{E}^*)$ with ξ nodes $\{n_1, \ldots, n_\xi\}$. A directed link between n_i and n_j can be represented by the sequence $P(i)\ P(j)$, where P is some permutation of the set of nodes. We need know nothing about P except that it is a permutation of the index set $\{1, \ldots, \xi\}$. Since however, we do not know how the original index set was assigned, we may as well take P as the

[1] "Entities should not be multiplied unnecessarily.", William of Occam (c.1320).

identity function, and hence ignore P. This modification reduces the complexity of the mathematical notation below, without any loss of generality; this can be formally justified by the definition of isomorphic graph equality (see Definition 5.18).

Now consider a message, which consists of an arbitrary sequence of symbols drawn from some finite alphabet. From Shannon's (1948) information theory we know that the message length[2] required for each symbol in the message so that the message's length is minimised is $\log_2(\frac{1}{\Pr\mu}) = -\log_2(\Pr\mu)$, where $\Pr(\mu)$ is the probability of some symbol μ appearing in the message (see Section 5.2.4).

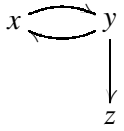


Figure 6.1: Simple graph

Example Consider the small graph shown in Figure 6.1. This graph consists of three nodes $\{x, y, z\}$ and three edges $\{(x, y), (y, x), (y, z)\}$. Recall that the order of the nodes in an edge is significant. Hence, a message describing this graph could be:

$$xy \quad yx \quad yz$$

It is obvious that the above message contains three distinct symbols, and contains a total of six symbols. Therefore the respective probabilities of x, y, z are $\frac{1}{3}$, $\frac{1}{2}$ and $\frac{1}{6}$. From Section 5.2.4, the corresponding message lengths (for each symbol) are respectively given by $\log_2 3$, $\log_2 2$ and $\log_2 6$. Multiplying the message length of each symbol by the number of times it occurs, yields a total message length of

$$2 \times \log_2 3 + 3 \times \log_2 2 + 1 \times \log_2 6 \approx 8.75 \text{ bits}$$

This can be formalised as follows. We can see that the length of a message describing the graph G^* with ξ nodes $\{n_1, \ldots, n_\xi\}$, is

$$-\sum_{i \in \mathcal{N}} c(i) \log_2 (\Pr i)$$

Where c_i is a count of the number of times node n_i appears in the description. All that remains is to find some function for c and an estimate of the probability of a node occurring in the description.

We know that for any multi-graph, each edge impinges on *exactly* two nodes. In a message describing the structure of a multi-graph, each node appears once for each edge that impinges on it. Therefore, the number of times each node appears in our message is equal to the number of edges impinging on it. In other words, the function c is really the *degree* of the corresponding node.

Moreover, in any multi-graph, the sum of the number of edges impinging on any node over the whole graph is twice the number of edges in the graph, i.e.,

$$\sum_{i \in \mathcal{N}} c_i = 2E$$

where c_i is the number of edges impinging on node n_i, and E is the number of edges in the graph. Now from Theorem 5.1 we know that $\sum_i d_i = 2E$. Hence

$$\sum_{i \in \mathcal{N}} c_i = 2E = \sum_{i \in \mathcal{N}} d_i$$

[2]We use $\log_2$ in what follows, which yields a result in bits. It is possible to use logarithms to any base, but the results will differ by a multiplicate constant.

where d_i is the degree of node n_i. The message describing the graph has $2E$ symbols. Therefore, the probability of a node occurring in a message is the node's degree divided by the sum of degree of all nodes.

Hence, we can conclude that the message length of an arbitrary node n_i in a message is:

$$-\log_2\left(\frac{d_i}{D}\right)$$

Where d_i is the degree of node n_i, and D is $\sum_{i\in\mathcal{N}} d_i$. Therefore, the total length of a message describing the structure of a multi-graph is

$$-\sum_{i\in\mathcal{N}} d_i \log_2\left(\frac{d_i}{D}\right) \tag{6.1}$$

We assume that $0\times\log_2\left(\frac{0}{x}\right)=0$ for $x>0$.

Such a (partial) message is sufficient to describe the edges in a multi-graph. This result holds unchanged for a graph, $G(\mathcal{N},\mathcal{E})$.

6.2.1.1 Example: Chain Graph

To demonstrate the above, consider the small graph, shown in Figure 6.2. This graph consists of n nodes arranged in a chain, such that, there is an edge from node 1 to node 2, node 2 to node 3, etc. until finally node $n-1$ has an edge to node n, and node n has no other edges impinging on it.

Figure 6.2: Chain Graph

Degree	Num. Nodes	Degree × Nodes	1/Pr	Length
1	2	2	$2(n-1)$	$2\log_2(2(n-1))$
2	$n-2$	$2(n-2)$	$n-1$	$2(n-2)\log_2(n-1)$
Total		$2(n-1)$		$2(n-1)\log_2(n-1)+2$

Table 6.1: Calculation of Chain Graph's Message Length

The calculation of the resultant message length is shown in Table 6.1. For ease of presentation, the reciprocals of the probabilities are shown, thus yielding positive lengths. The first row gives the calculation of the contribution of the two end nodes to the message length. The second row, the contribution of the $n-2$ middle nodes. The last row shows the column totals for columns 3 and 5. Column 3, "Degree × Nodes", shows the product of the first two columns. Column 4, "1/Pr", shows the reciprocal of the probability of a node appearing in a message, and is just the total of column 3 divided by the entry in column 1. Finally, column 5, "Length", shows the length attributable to each type of node, and is just the entry in column 3 times the log of the entry in column 4.

Hence, for a chain graph of n nodes, the message length is $2(n-1)\log_2(n-1)+2$.

6.2.1.2 Example: Star Graph

Now, consider another small graph, shown in Figure 6.3. This graph consists of n nodes arranged in a star-like configuration, such that, node n has edges to every other node, and the remaining nodes $(1,\ldots,n-1)$ have no other edge connections dependencies. Note that the same result would be achieved if the direction of every arrow was reversed.

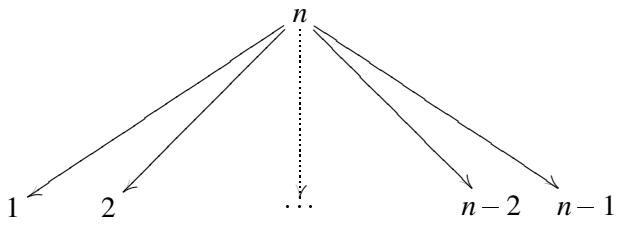


Figure 6.3: Star Graph

Degree	Num. Nodes	Degree × Nodes	$1/\Pr$	Length
1	$n-1$	$n-1$	$2(n-1)$	$(n-1)\log_2(2(n-1))$
$n-1$	1	$n-1$	2	$(n-1)\log_2 2$
Total		$2(n-1)$		$2(n-1)+(n-1)\log_2(n-1)$

Table 6.2: Calculation of Star Graph's Message Length

The calculation of the resultant message length is shown in Table 6.2. Hence, for a star object of n nodes, the message length is $2(n-1)+(n-1)\log_2(n-1)$.

6.2.1.3 Discussion

Considering the two previous examples, we see that each has the same number of nodes, and the same number of edges. However, each yields a different message length. Further, the star configuration has a message length less than or equal to the same sized chain configuration for most n.[3]

6.2.2 Connections in a Single Object

The previous section showed how to create a partial message for describing the edges in a standard (multi-)graph. However, we ignored the issue of how this message might be decoded by the receiver. In this section we expand our message definition to create decodable messages.

In order for our message to be understandable by the receiver, each message needs to include information about the size of the symbol set required for encoding the graph's structure, i.e., the cardinality of the node set for the graph. If we assume some form of arithmetic coding scheme, this information allows the receiver to establish an *a priori* probability for each symbol, see MacKay (1997), and hence decode the message. It must be stressed that our message lengths are theoretical results, and cannot be achieved in practice.

This begs the question of how to calculate the message length of the representation of the cardinality of a set, in our message. There are a number of possible representations. We have chosen to use the $\mathcal{L}^*$ function (see Section 5.2.3), which yields an optimal universal prefix encoding for all positive integers.[4] Recall that the code length of this coding scheme is given by $\log_2^*$. We need

[3] Clearly, the formulae make no sense for $n=0$ or $n=1$, and there is equality only for the special cases of 2 and 3 nodes, when the respective graphs are identical.

[4] See the discussion at the end of Section 5.2.3.

to add one to the natural number being transmitted so that zero can be sent. Note that this function is strictly greater than zero, for all positive integers, and is strictly monotonically increasing.[5]

Hence the length of a message describing a (multi-)graph is given by

$$\log_2^*(E+1) - \sum_{i \in \mathcal{N}} d_i \log_2 \left(\frac{d_i}{D} \right) \tag{6.2}$$

This extension now allows us to transmit a standard graph, which corresponds to a design graph with *no objects*.

It follows that for a single object, such a message is sufficient to describe its structure; and that for a given object, set of basic entities and connections, the message length is fixed.

Recalling our discussion in Section 6.2.1.3, it is easy to see that each of the two sample graphs could be (contrived) designs. Further each design contains the same number of edges. Hence as we saw each design yields a different message length.

6.2.3 Extensions for Multiple Objects

In the previous section we showed how the structure of a single object could be represented by a message,[6] and how the message's length[7] could be calculated.

In this section, we shall extend these ideas to permit multiple (and nested) objects. The significant difference is that now a basic entity in one object may need the services of a basic entity in another object. Hence there are now two kinds of links, intra-object and inter-object. Inter-object links also include links between entities nested inside different enclosing objects; as well as the more common meaning of between objects.

Our message now needs to include information about object nesting and basic entity connections. It is convenient to assume that there is always a single top-level object, even if one has to be artificially created. Hence the object nesting forms a tree with a single root.

To make this discussion clearer, we consider an example, see Figure 6.4. The design is purely artificial and has no implied meaning. The design has the modular tree structure shown in Figure 6.5. To make this exposition clearer, basic entities are shown as lowercase letters and objects are shown as uppercase letters. Further, each entity is given a unique letter.

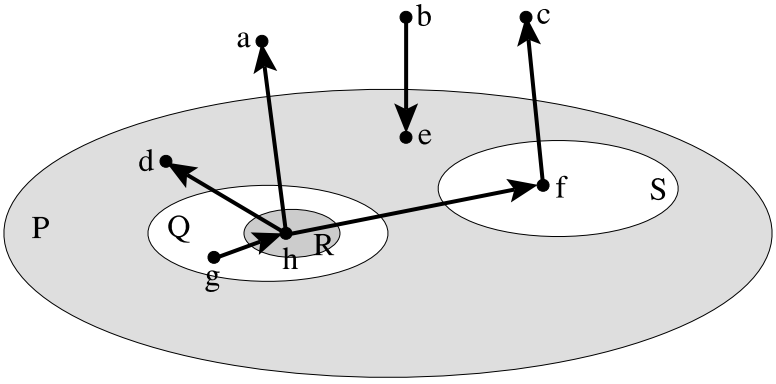


Figure 6.4: A Nested Design Graph

Once again, we need to describe the edges between two basic entities. Intra-module edges can be handled as before. We need a method for describing inter-module edges, i.e., edges which pass through a module's boundary. In a tree, there is a unique minimum length path between any two

[5] That is, $\forall a, b \in \Re, \quad a > b \implies f(a) > f(b)$.

[6] Such messages are not unique, because the edge descriptions in the message can be reordered.

[7] The length for a given structure is constant.

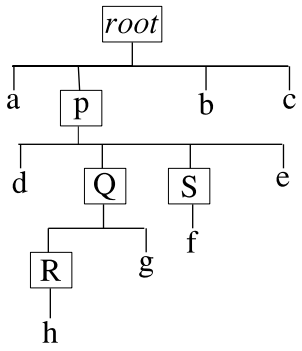


Figure 6.5: Modular Tree Structure of Nested Design Graph

nodes, and since we want to minimise message length this is a good starting point. So for example the edge between h and f passes upwards from R to Q to P, where it peaks, and passes downwards to S. It reduces message length to describe an edge in the highest object containing that edge, in this case P. Hence the edge between h and f can be described by the message fragment

$$QRh-Sf$$

as an edge description in object P.

Applying this principle to our example design, for the edge descriptions we have

$$\begin{array}{lcl}
\text{edges(root)} & = & PQRh-a \quad b-Pe \quad PSf-c \\
\text{edges(P)} & = & QRh-d \quad QRh-Sf \\
\text{edges(Q)} & = & g-Rh \\
\text{edges(R)} & = & \text{none} \\
\text{edges(S)} & = & \text{none}
\end{array}$$

The entities in several objects can be labelled by using distinct (small) alphabets for each object, provided, that within the top-level of each object we can distinguish objects from basic entities. In other words, as long as we can tell that the last symbol in an edge description is a basic entity rather than an object. Now each part of an edge's description only needs to be processed in the context of its encapsulating object. Therefore, our message must provide information about which entities in an object are themselves other objects. The use of such small alphabets, reduces the number of distinct symbols required in each alphabet, and hence the resultant message length.

Hence in our example, root contains the entities a, b, c and P, and hence only needs an alphabet with these four symbols.

In view of the above, we use the following message structure, for each object recursively:

$$\begin{array}{lll}
\text{object_description} = & \text{object_id} + & \\
& |\text{nodes}| + & \\
& |\text{objects}| \ \{\text{object_description}_1, \ldots, \text{object_description}_m\} + & \\
& |\text{edges}| \ \{\text{edge}_1, \ldots, \text{edge}_e\} & \\
 & & \\
\text{edge} = & \text{point}_1 - \text{point}_2 & \\
 & & \\
\text{point} = & \{\text{object_id}\}^* \ \text{basic_entity_id} &
\end{array}$$

Where $|x|$ encodes the cardinality of the indicated set, and the symbols '+', '–', '{', and '}' are not part of the message, but are shown only to aid human comprehension of the message's structure. The symbol '+' represents concatenation, and '*' indicates a (possibly empty) repeating set.

Note that in the above message description, we distinguish between objects and basic entities, but they are drawn from the same alphabet. They can only be distinguished by the receiver because of the included list of nested object_descriptions, each of which starts with its own object_id.

Continuing our example, we can describe our hypothetical design, (see Figures 6.4 and 6.5), by the following five message fragments:

$$\begin{aligned}
\text{obj(root)} &= 1+4+1(\text{obj(P)})+3(PQRh-a,b-Pe,PSf-c)\\
\text{obj(P)} &= P+4+2(\text{obj(Q)},\text{obj(S)})+2(QRh-d,QRh-Sf)\\
\text{obj(Q)} &= Q+2+1(\text{obj(R)})+1(g-Rh)\\
\text{obj(R)} &= R+1+0()+0()\\
\text{obj(S)} &= S+1+0()+0()
\end{aligned}$$

Where the symbols ‘,’, ‘+’, ‘–’, ‘(’, and ‘)’ are not part of the message, but are shown only to aid human comprehension of the message’s structure. Note that the ‘1’ at the start of the root’s object description is only to maintain symmetry between the object descriptions and has no other purpose.

The full message is created by textually substituting the four sub-messages in place of each ‘obj(x)’. Hence the full message becomes:

$$\begin{aligned}
\text{design} = {} & 1+4+1(\\
& P+4+2(\\
& Q+2+1(\\
& R+1+0()+0()\\
&)+1(g-Rh),\\
& S+1+0()+0()\\
&)+2(QRh-d,QRh-Sf))+\\
& 3(PQRh-a,b-Pe,PSf-c)
\end{aligned}$$

We know from Section 6.2.1 that, the message length of an arbitrary node n_i in an object can be calculated as $-\log_2(d_i/D)$. However, how do we calculate the message length of an arbitrary object n_i? Counting the occurrences of each basic entity in the above descriptions will show that they occur with the same frequency as their corresponding node’s degree.

If we contract an object, keeping the edges which pass through its boundary, to a simple node, we can create the ‘degree’ of an object; and we see that each object occurs with the same frequency as its simple node’s degree. For example, the object Q has three edges passing through its boundary, and occurs three times in the above description. Unfortunately, this yields an incorrect probability for the total occurrences of the object_id in the message, and hence to a longer message length. The number of occurrences of the object_id in the message is out by one, because of its occurrence at the start of its own object_description. Hence, when calculating a object’s degree, we need to add one to the number of edges which cross the object’s boundary.

In our example, root contains the entities a, b, c and P. The degree of each entity is (respectively) 1, 1, 1, 3, plus another occurrence of P at the start of its object_description, so the corresponding probabilities are $\frac{1}{7}$, $\frac{1}{7}$, $\frac{1}{7}$ and $\frac{4}{7}$.

6.2.4 Further Extensions for HOOD

HOOD environmental objects do not fit into the model so far presented. Environmental objects can be referenced from *anywhere* in a design. Therefore encoding such inter-object access as above, would unjustifiably increase the message length for what is in fact outside of the designer’s control. We therefore decided to modify the edge encoding of the previous section to indicate if an environmental object was being accessed. This was done by adding a single Boolean flag in front of the second point of each edge’s description, to indicate whether or not the next point

is an environmental object. Environmental objects can only be referenced from an object in the current design tree, an environmental object can never reference an object in the design tree. This modification should not distort message length comparisons as it uniformly increases all edges by one bit, and we never change the number of edges in a design graph.

Hence our final model looks like:

Design Description Message

object_description = object_id +
|nodes| +
|objects| {object_description$_1$, ... , object_description$_m$} +
|edges| {edge$_1$, ... , edge$_e$}

edge = point$_1$ – env_object point$_2$

point = {object_id}* basic_entity_id

env_object = 1 iff next object is an environmental object, 0 otherwise

Length of Full Design Description Message

The length of a message describing a general design using this encoding has the form:

$$\text{length} = \sum_{m\in\mathcal{M}} \left(\log_2^*(N_m+1) + \log_2^*(N_m^{'}+1) + \log_2^*(E_m+1) - \sum_{n\in\mathcal{N}_m} f_n \log_2\left(\frac{f_n}{F_n}\right) + E_m \right) + 1 \tag{6.3}$$

Where $\mathcal{M}$ is the set of all modules (objects), N_m is the number of entities (objects + basic entities) in module m, $N_m^{'}$ is the number of objects in module m ($N_m \geq N_m^{'}$), E_m is the number of edges described in module m and $\mathcal{N}_m$ is the set of entities in module m. f_n is the frequency of entity n in module m, which for basic entities is given by the degree of the corresponding node, and for objects is the degree of the contracted node plus 1 and $F_n = \sum_n f_n$.

Length of Single Object Design Description Message

Hence for a design consisting of a single object, we have the following equation for its message length

$$\text{length} = \log_2^*(N+1) + \log_2^*(E+1) - \sum_{i\in\mathcal{N}} d_i \log_2\left(\frac{d_i}{2E}\right) + E + C \tag{6.4}$$

Where $C = 1 + \log_2^* 1$, E is the number of edges, N is the number of nodes in the graph, d_i is the degree of the node i.

6.3 A Complexity Measure?

Having defined a complexity measure, we should demonstrate, at least informally, that it captures notions of coupling and cohesion. Further that the complexity measure permits some form of trade-off between these two concepts. We will give some more complete examples of this in Chapter 8. For now we will just use an intuitive model.

Consider a design consisting of 10 basic entities, and three objects arranged in a balanced binary tree. Further, let the 10 basic entities form two highly-cohesive groups, with no (or very little) coupling between the groups. Intuitively, it seems reasonable that each group should be placed in a leaf node of the binary tree.

Now consider moving one basic entity from one group, L into the other R. The `object_description` of the group L will get smaller, it contains fewer entities and fewer links. The `object_description` of the group R will get larger, it now contains more entities. Additionally, the `object_description` of the root-group T will get larger, it contains several links from L to R. The original design has a complexity of 155 bits, whilst the second design has a complexity of 166 bits. Forming a single object results in a complexity of 172 bits.

However, if the design only contains 6 basic entities, the design has a complexity of 58 bits, but moving all the entities into a single object has a complexity of 52 bits.

6.4 Theoretical Validation

Many proposed software complexity metrics have been criticised for not having been theoretically validated. In this thesis we present a new complexity measure based on a message passing paradigm, and this section will show that this metric satisfies Weyuker's Properties (Weyuker, 1988) for a complexity measure. Her proposal is widely accepted (Shepperd and Ince, 1993) as providing a basis for theoretical validation.

In Section 6.2, we proposed a complexity measure based on the length of a message describing the structure of the underlying graph. We showed how such a message could be constructed, and how the length of such a message could be calculated. In this section, we shall examine how our proposed complexity measure performs against Weyuker's Properties for any complexity measure.

6.4.1 Weyuker's Properties

Weyuker's Properties requirements are mainly taken from Shepperd and Ince (1993, p.68–69), with the systematic replacement of program by system, and a few rewordings from Briand et al. (1996).

Let Ψ be our complexity measure (as described above), and let $\mathcal{S}$ be the set of all software systems.

Before proceeding to look at Weyuker's Properties, it is worth pausing to prove five theorems and two small lemmas, which are useful in the following proofs of Weyuker's Properties.

Definition 6.1. *Define* $0\log_2 0 = 0$*, since* $\lim_{\varepsilon\to 0}\varepsilon\log_2\varepsilon = 0$.

Lemma 6.1.

$$\sum_i x_i \log\left(\frac{x_i}{X}\right) = \sum_i x_i \log x_i - X\log X$$

where $X = \sum_i x_i$.

Proof.

$$\begin{aligned}\sum_i x_i \log\left(\frac{x_i}{X}\right) &= \sum_i x_i \left(\log x_i - \log X\right) \\ &= \sum_i x_i \log x_i - \log X \sum_i x_i \\ &= \sum_i x_i \log x_i - X \log X\end{aligned}$$

□

Lemma 6.2. *The function $g(x) = (x+1)\log_2(x+1) - x\log_2 x$ is always non-negative and strictly monotonically increasing for all $x \geq 0$.*

Proof.

$$\begin{aligned}\frac{dg}{dx} &= \frac{1}{\ln 2}\left(1 + \ln(x+1) - (1+\ln x)\right) \qquad x > 0 \\ &= \frac{1}{\ln 2} \ln\left(1 + \frac{1}{x}\right) \\ &> 0\end{aligned}$$

Hence $g(x)$ is strictly monotonically increasing for $x > 0$, and

$$\begin{aligned}\lim_{\varepsilon \to 0} g(\varepsilon) &= \frac{1}{\ln 2} \ln\left(1 + \frac{1}{\varepsilon}\right) \\ &> 0\end{aligned}$$

Hence g is strictly monotonically increasing for all $x \geq 0$.

To prove $g(x) \geq 0$, for $x \geq 0$ just rearrange the terms in g, thus

$$\begin{aligned}g(x) &= (x+1)\log_2(x+1) - x\log_2 x \\ &= \log_2(1+x) + x\log_2\left(1 + \frac{1}{x}\right) \\ &\geq 0\end{aligned}$$

Since both summands on the right are positive. □

Our thanks to Peter Williams for help with the above proof.

Corollary 6.1. *Let $g(x) = (x+1)\log_2(x+1) - x\log_2 x$ then $g(X) - g(x) \geq 0$ for all $X \geq x \geq 0$, with equality only when $X = x$.*

Proof. By Lemma 6.2 $G(X) \geq g(x)$ and both terms are positive. □

Theorem 6.1. *Assuming optimal encoding for all messages, adding an extra symbol to a message cannot decrease the size of the message.*

Proof. If a message has symbols from an arbitrary set $x_1, \ldots, x_n$ such that each symbol occurs with frequency $f_1, \ldots, f_n$ respectively, where some f_i may be zero, then the probability of picking symbol x_i at random from the message's content is f_i/F where $F = \sum_i f_i$. From Shannon's information theory, we know that the minimum message length for symbol x_i is $-\log_2(f_i/F)$, and therefore the minimum message is given by

$$\begin{aligned}l &= -\sum_i f_i \log_2\left(\frac{f_i}{F}\right) \\ &= F\log_2 F - \sum_i f_i \log_2 f_i \qquad \text{by Lemma 6.1}\end{aligned}$$

Hence we need to show that l is monotonically increasing for all members of X.

Let $\hat{z}$ denote z the contents of the message after it has been changed by adding one symbol. Hence

$$\hat{l} = \hat{F}\log_2\hat{F} - \sum_i \hat{f}_i \log_2 \hat{f}_i$$

All $\hat{f}_i$ are equal to f_i except one (denote this special f_i by f_i', say), which is $f_i + 1$, and hence $\hat{F} = F + 1$. Then

$$\begin{aligned}
\hat{l} - l &= \hat{F}\log_2\hat{F} - \sum_i \hat{f}_i \log_2 \hat{f}_i - F\log_2 F + \sum_i f_i \log_2 f_i \\
&= \hat{F}\log_2\hat{F} - F\log_2 F + \sum_i f_i \log_2 f_i - \sum_i \hat{f}_i \log_2 \hat{f}_i \\
&= \hat{F}\log_2\hat{F} - F\log_2 F + f_i'\log_2 f_i' - \hat{f}_i'\log_2 \hat{f}_i' && \text{since all other } f_i = \hat{f}_i \\
&= g(F) - g(f') && \text{where } g(x) = (x+1)\log_2(x+1) - x\log_2 x \\
&\geq 0 && \text{by Corollary 6.1}
\end{aligned}$$

□

The above theorem may not hold under conditions of sub-optimal encodings.

Corollary 6.2. *Adding a connection to any part of a design graph cannot decrease the length of partial message describing the object's connections.*

Proof. Adding an extra connection to an object, either involves adding an extra degree to a node in an object or adding an extra edge crossing the boundary of an encapsulated object. In either case, an extra symbol is added to the partial message describing the object's connections, and hence by Theorem 6.1 the length of this partial message cannot decrease. □

Theorem 6.2. *Adding an additional edge to a design graph, increases the design's complexity.*

Proof. For any design graph, adding an edge is equivalent to adding 1 to the total number of connections in at least one object. From Equation 6.3, we have

$$l = \log_2^*(N+1) + \log_2^*(N'+1) + \log_2^*(E+1) - \sum_{n\in\mathcal{N}} f_n \log_2\left(\frac{f_n}{F}\right) + E \quad \text{for the affected object}$$

and

$$\hat{l} = \log_2^*(N+1) + \log_2^*(N'+1) + \log_2^*(\hat{E}+1) - \sum_{n\in\mathcal{N}} \hat{f}_n \log_2\left(\frac{\hat{f}_n}{\hat{F}}\right) + \hat{E}$$

All the terms of which are obviously strictly monotonically increasing, except $-\sum_{n\in\mathcal{N}} \hat{f}_n \log_2\left(\frac{\hat{f}_n}{\hat{F}}\right)$ which is monotonically increasing by Corollary 6.2. Hence the length must increase. □

Theorem 6.3. *Adding an additional unconnected node to a design graph, increases the design's complexity.*

Proof. For any design graph, adding a node to an object is equivalent to adding 1 to the total number of nodes in the encapsulating object. From Equation 6.3, we have

$$l = \log_2^*(N+1) + \log_2^*(N'+1) + \log_2^*(E+1) - \sum_{n\in\mathcal{N}} f_n \log_2\left(\frac{f_n}{F}\right) + E \quad \text{for the affected object}$$

and

$$\hat{l} = \log_2^*(\hat{N}+1) + \log_2^*(N^{'}+1) + \log_2^*(E+1) - \sum_{n\in\mathcal{N}} f_n \log_2\left(\frac{f_n}{F}\right) + E$$

The node is unconnected, so that all the terms in the above sum are unchanged, except $\log_2^*(\hat{N}+1)$. Since $\log_2^* x$ is a strictly monotonically increasing function the length of the module's description must increase. □

Theorem 6.4. *Adding an additional object to a design graph, increases the design's complexity.*

Proof. Follows immediately from Theorem 6.3 since the node count in the encapsulating object rises and the object count in the encapsulating object rises. □

Theorem 6.5. *Adding an additional node with degree at least 1 to a design graph, increases the design's complexity.*

Proof. Obvious from proofs of Theorems 6.2 and 6.3. □

W-Property 6.1. *The measure must not assign the same number to all systems:*

$$\exists p, q \in \mathcal{S} \bullet \Psi(p) \neq \Psi(q)$$

Proof. Immediately follows from Theorems 6.2–6.5. □

W-Property 6.2. *There exist only a countable number of systems for a given measurement value.*

> The stated purpose of this axiom is to 'strengthen' the [previous] axiom, as violation suggests that the measure is comparatively insensitive.
>
> Shepperd and Ince (1993, p.68)

Proof. A graph, $G(\mathcal{N}, \mathcal{E})$, consists of two countable sets, namely: nodes and edges. It follows immediately that the number of graphs is countable since we have only countable unions of countable sets. □

W-Property 6.3. *There are systems drawn from the same equivalence class:*

$$\exists p, q \in \mathcal{S} \bullet \Psi(p) = \Psi(q)$$

Proof. Our proof is by constructing two system with the same measure. Let p be an arbitrary system with an underlying graph such that all nodes do not have identical degree. Let q have exactly the same graph, but with the direction of each edge reversed. Since our complexity measure is derived from a message describing the structure of a system; it follows that reversing the edge direction cannot change the complexity. Since, p and q are not isomorphically equivalent graphs, this concludes our proof. □

W-Property 6.4. *There must exist systems that compute the same function but have different numbers attached to them.*

Since we are discussing systems not programs, it seems more general to replace 'compute the same function' by 'serve the same purpose'.

Proof. Construct a new system q from the original system p by adding an unconnected operation. Then from Theorem 6.3 the complexity of q must be greater than p. □

W-Property 6.5. *The measure must be monotonic, wrt. adding components:*

$$\forall p, q \in \mathcal{S} \bullet \Psi(p) \leq \Psi(p \circ q) \ \wedge \ \Psi(q) \leq \Psi(p \circ q)$$

Where $\circ$ denotes the concatenation operation (see Section 5.1.6).

Proof. Our proof is by induction on the structure of the graph. Recall that design concatenation is derived from graph union, which is in turn derived from set union. Therefore $p \circ q$ has at least as many nodes as $\max(|\mathcal{N}_p|, |\mathcal{N}_q|)$ and has at least as many edges as $\max(|\mathcal{E}_p|, |\mathcal{E}_q|)$. Hence by structural induction using Theorems 6.2–6.5, the design's complexity cannot decrease, as required. □

This result holds, even if there are no links between the constituent designs p and q.

Corollary 6.3. *The complexity of a design concatenated with itself has the same complexity as the original:*

$$\forall p \in \mathcal{S} \bullet \Psi(p) = \Psi(p \circ p)$$

Proof. Immediately follows from the definition of design concatenation (see Section 5.1.6), since for all sets X, $X \cup X = X$. □

W-Property 6.6. *Concatenation of a system r to another system must not always yield a constant increment to the total complexity measure:*

$$\exists p, q, r \in \mathcal{S} \bullet \Psi(p) = \Psi(q) \ \wedge \ \Psi(r \circ p) \neq \Psi(r \circ q)$$

Also:

$$\exists p, q, r \in \mathcal{S} \bullet \Psi(p) = \Psi(q) \ \wedge \ \Psi(p \circ r) \neq \Psi(q \circ r)$$

Proof. Since design concatenation is a commutative operation, we have $p \circ q = q \circ p$ for all p and q. From Corollary 6.3 concatenating a design with itself yields a design of the same complexity. Hence let p and q be design graphs as in the proof of W-Property 6.3, and let $r = p$, then $\Psi(p) = \Psi(q)$ and $\Psi(p \circ r) = \Psi(p)$ but $\Psi(q \circ r) \geq \Psi(p \circ r)$ since $q \circ r$ contains more edges than p and by Theorem 6.2 this increase the design's complexity, as required. □

Actually all that is required is that p and r have nodes/edges in common, whilst q has nothing in common with p.

W-Property 6.7. *The measure must be sensitive to the ordering of the system components. Let* ρ *be a permutation function, then:*

$$\exists p \in \mathcal{S} \bullet \Psi(p) \neq \Psi(\rho(p))$$

We interpret ordering to refer to moving entities around the hierarchical structure. For example creating a new object or moving a basic entity from one object to another. Weyuker, was discussing moving program fragments, and we regard moving entities as similar for designs.

Proof. See Section 6.3 for an example. □

W-Property 6.8. *The measure must be insensitive to renaming changes of system components. Let* τ *be a renaming function, then:*

$$\forall p \in \mathcal{S} \bullet \Psi(p) = \Psi(\tau(p))$$

Proof. In Section 6.2.2, we saw that node index numbers are not significant in determining the length of a message describing the graph's structure. □

W-Property 6.9. *Module monotonicity*

$$\exists p, q \in \mathcal{S} \bullet \Psi(p) + \Psi(q) < \Psi(p \circ q)$$

> This property reflects the fact that there may be interaction between the concatenated [designs]. ... [W-Properties 6.5 and 6.9] allow for the possibility that as a [system] grows from its component bodies, additional complexity is introduced due to the potential interaction among these parts.
>
> Weyuker (1988, p.1363)

Proof. The proof is by construction of an example. Let p be a single object system, containing two basic entities a_1 and a_2, say. Hence $\Psi(p) \approx 11.8$. Now let q be another single object, containing the basic entities b_1 and b_2, with the restriction that none of the basic entities are the same. Now form $r = p \circ q$, then $\Psi(r) \approx 33.6$. Therefore $2 \times 11.8 = 23.6 < 33.6$ as required. □

Note that there are designs which do not satisfy this property (e.g., set $q = p$ giving $\Psi(p) + \Psi(q) = 2\Psi(p) > \Psi(p \circ q) = \Psi(p)$), which is of course, acceptable since the property only requires an instance to exist. Which is what Weyuker intended from her logic.

6.5 Conclusion

In this chapter we have described a new software design complexity measure, based on a message passing metaphor, and shown that Weyuker's Properties hold for this measure. Having established reasonable theoretical grounds for believing that Ψ is a complexity measure, we need to conduct an empirical validation. Before doing so we need a prototype tool to both evaluate Ψ, and look for improved designs. Such a system is the subject of the next chapter.

> Mathematics is not a deductive science—that's a cliche.
> When you try to prove a theorem, you don't just list the hypotheses,
> and then start to reason.
> What you do is trial and error, experimentation, guesswork.
>
> PAUL R. HALMOS
> *I Want to be a Mathematician* (1985)

> A thing is obvious mathematically after you see it.
>
> R. D. Carmichael
> *Mathematical Maxims and Minims* (1988)

Part III

Morpheus

Chapter 7

Morpheus: A Prototype System

Synopsis

This chapter describes both our extensions to HOOD for capturing a more detailed description of the proposed software architecture and the implementation of our prototype system, *Morpheus*, for improving designs. *Morpheus* compares designs based on our previously defined complexity measure Ψ.

Chapter 6 showed that our complexity measure, Ψ, had some desirable theoretical properties for a complexity measure. Chapter 6 also described the computation of our complexity measure from a hierarchical design graph.

This chapter describes *Morpheus*, a prototype tool, for finding Ψ from an initial design expressed in HOOD. *Morpheus* reads in an initial design and then searches for modifications to this initial design which yield a lower numerical value for Ψ, and hence are we believe less complex; and therefore *ceteris paribus* better designs.

Chapter 8 will show some practical examples of using *Morpheus*.

7.1 Extensions to HOOD - Augmented HOOD

HOOD was described in Chapter 3. In this section we describe our extensions to HOOD. These extensions are necessary because standard HOOD does not provide sufficient detail on the relationships between entities, to permit the calculation of our complexity measure or the effect of changing the modular structure.

In particular, the description of an operation in an ODS provides no information on the entities it needs to perform its function. However, an object's description has to provide this information. So the designer must have given this matter some consideration. Not only is this information necessary for our present purposes, but it seems useful to record these details as part of the development process.

In considering how to extend HOOD, it was important to make only minor modifications to the existing ODS—for compatibility with existing tool-sets. The changes had to be in line with HOOD's philosophy, and only the changes necessary for this research should be included. These limitations were imposed so that we could (in principle) collect and analysis 'real world' HOOD designs, for validation of *Morpheus*.

In line with the above objectives, the only modification to the existing ODS is in the pseudo-code section of the description of operations. This introduced an optional new section called a

code_linkage section. This new section contains information on the types and variables[1] used by the operation. The formal changes to the ODS's syntax are documented in Appendix A.

It was also necessary to make an extension to HOOD's semantics. We permitted the identification of used operations in an operation's definition to include constants as well as operations. This could have been done by adding a further field to the existing ODS for operations. However, since from a general semantic perspective there is little difference between a constant and a procedure, this seems a reasonable change. Additionally, not distinguishing these categories makes mechanical collection easier; a point we shall return to shortly.

It may be objected that these changes impose more housekeeping on the designer and an extra workload for *Morpheus*'s validation phase. The first objection is in a small way justified, but since the designer must consider these issues, it does not seem a significant problem. Further, since HOOD does not permit variables to be accessed outside of the enclosing object, and the current ODS provides no indication of why the variables are present, this modification seems desirable just from a design perspective.

The second objection, additional workload, (and indeed the first objection to some extent), can both be overcome by creating a tool to parse the Ada pseudo-code, and generate the necessary information automatically.

7.2 Implementation

This section describes the implementation of *Morpheus*. It is not intended to be a design document as such, but rather to highlight the main features of *Morpheus*, and explain how *Morpheus* generates its advice from an initial design. The entire *Morpheus* system is written in Pop-11.[2]

7.2.1 Basic Structure

The basic structure of *Morpheus* is shown in Figure 7.1. A proposed design, expressed in Augmented HOOD's ODS format is entered into *Morpheus*. *Morpheus* parses the entered design along with a predefined environment to produce a parse tree of the initial design.

The parse tree is then processed by the Data Analyser to resolve cross references and determine the modular structure of the entered design. The Data Analyser produces a hierarchical design graph representing the structure of the initial design (see Section 5.1.5).

The design graph is then passed to the final phase, the Improvement Engine. The Improvement Engine conducts a hill climbing search, by constructing possible (valid) alternative graphs; and comparing the complexity values for each proposed graph. Finally, the Improvement Engine produces a report describing the graph with the lowest computed complexity.

The following sections describe each of these three phases in more detail.

7.2.2 Parser

As we saw earlier, the purpose of this phase is to read in the proposed design and generate a parse tree. A parse tree shows the composition of the initial design without regard for the text used to represent the design. So far, we have described the Parser as if it was a single item, however, the Parser consists of two components, namely: a lexical analyser and a syntax analyser. The purpose of the lexical analyser is to process the input file and form the individual input characters into discrete symbols according to the lexical conventions of the input language. The syntax analyser

[1] HOOD's data entities.

[2] Pop-11 is a procedural language originally designed for artificial intelligence applications. Pop-11 shares a number of features with modern Lisp, and provides a rich type system, which is dynamically typed, with facilities for adding new user-defined types, and automatic storage management. For further information see, for example, Barrett et al. (1985) or Laventhol (1987). Anderson (1989) provides a collection of papers on some of the more modern features of Pop-11 not covered in the previous books. (This description of Pop-11 is adapted from POPLOG's on-line help file `help pop11`.)

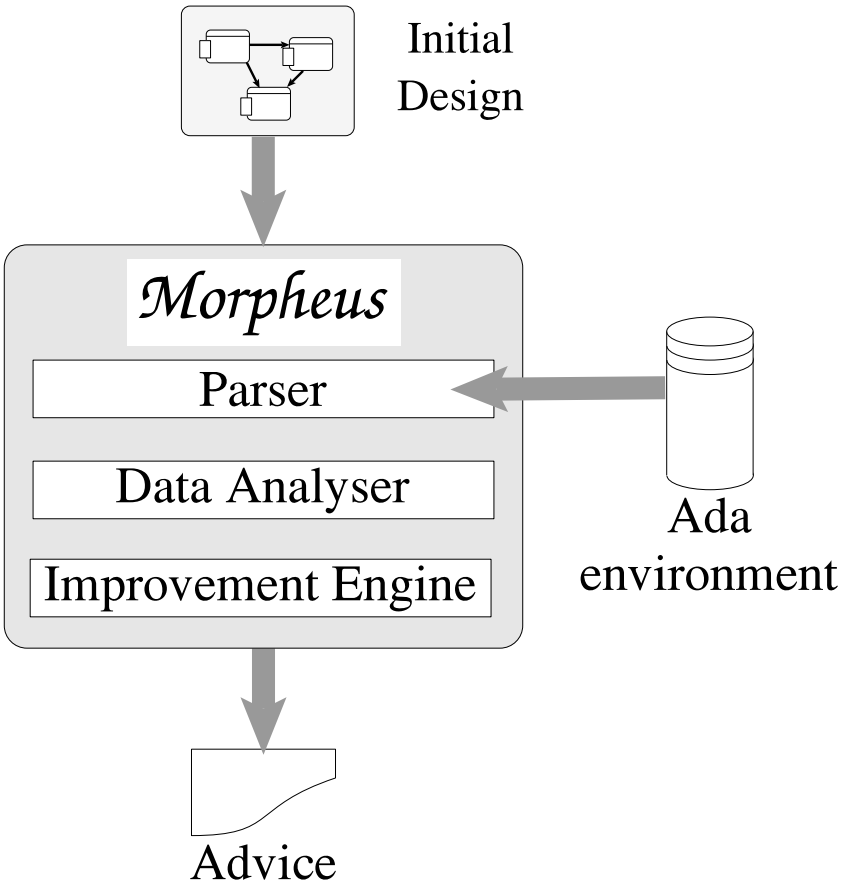


Figure 7.1: Architecture of *Morpheus*

takes each symbol produced by the lexical analyser and constructs a parse tree according to the syntax rules of the language.

The advantage of using a parse tree is that only this phase of *Morpheus* needs to handle input and know the concrete syntax of HOOD.[3] The construction of a parse tree from a source file which has a well-defined syntax is, of course, a standard phase of all compilers—and is well understood.[4] Appendix D of Delatte et al. (1993) contains a sample `YACC` description of HOOD's ODS using Standard Interchange Format (SIF). This was used as the basis for our input grammar to the Parser. We did, of course, add our new syntax rules for Augmented HOOD.[5]

The work carried out by a parser as syntax phrases are recognised can vary considerably. We choose to make our parser do very little work—just construct a new node and update the parse tree accordingly. This decision was made because further processing of a HOOD design requires access to information outside of the current syntax phrase. In passing it should be noted that HOOD does not require the declaration of an entity before it is referenced. Hence the information for further processing might not be available until after the whole file has been read in.

The design of any real parser has to address the problem of errors in the supplied input. We

[3]It would be quite possible for such a parse tree to be produced from some other tool, perhaps whilst a design was being sketched out on a drawing tablet, but see Section 7.3.2.

[4]See for example Bornat (1979) or Watson (1989). From theoretical work in this area, a number of so-called 'parser-generators' have evolved; most notably `YACC` under UNIX. Our implementation does not use `YACC` but rather the `LR_parser` written by Robert Duncan for Pop-11. `LR_parser` is similar to `YACC`, but with some additional meta-rules for handling optional and repeating syntax phrases.

[5]For further information on the construction of `YACC` style grammar descriptions from BNF, see Schreiner and Friedman, Jr (1985).

choose to assume, for ease of construction, that the input was essentially error free. If a syntax error is discovered an error message is output and *Morpheus* halts. Unfortunately, HOOD's syntax is not well suited to clear error messages because the ODS sections do not have well-defined closers.

The Parser is also responsible for reading in the 'Ada environment'. This environment is currently based on a subset of Annex C of Ichbiah et al. (1983). The purpose of this environment is to define standard types (such as integer and Boolean) and operations (such as arithmetic operations). With one minor exception, to be explained shortly, this environment is treated just like any part of the supplied design. Therefore, in principle, it is possible to replace the chosen Ada environment with another.[6]

We noted earlier that the environment was handled with one exception. The environment is processed, by the Parser as normal, whilst however permitting a dollar symbol to be part of an identifier.[7] This feature of the Parser is optional, but its use permits the creation of 'private' objects which cannot be accessed outside of the environment file. Since, with this facility disabled (the normal case), attempting to use identifiers containing dollar symbols causes an error. This feature is currently used to define `$STANDARD` as the base environment object.

7.2.2.1 The Parse Tree

This section briefly considers the produced parse tree. The precise structure of a parse tree node is a design decision, and is not wholly dictated by the underlying grammar. *Morpheus* contains around 70 distinct node kinds. For example, the node for **required objects** consists of four fields, namely: `line_number`, `object_header`, `note` and `required_entities`. Where the last field may have an arbitrary set of (non-empty) members. This layout corresponds (almost) exactly with the underlying syntax phrase, **required_object** (see BNF production 13 in Delatte et al. (1993, Appendix D)). The one exception being the addition of the line number field which is used for error reports. A similar approach is used for all the required node types.

7.2.3 Data Analyser

The Data Analysis phase of *Morpheus* takes the parse tree constructed by the Parser (see Section 7.2.2) and generates a graph detailing the structure of the supplied design. In a standard compiler this phase would correspond to semantic analysis. HOOD's semantic rules are described in Appendices A and E of Delatte et al. (1993). The overall logic of the Data Analysis phase is shown in Figure 7.2.

In essence, the graph required by the Improvement Engine (see Section 7.2.4) only requires information on the hierarchical structure of the design and information on which entities are connected. The former describes how entities are nested with respect to objects. The latter shows which other entities an individual entity depends on in order to perform its function. Since the Improvement Engine never changes the linkage information,[8] (only the hierarchical structure) we choose to use two separate data structures for these two different kinds of information.

In order for the Improvement Engine to process the supplied design graph in the context of HOOD, a couple of secondary pieces of information are required, namely: which objects are environmental objects and which entities in an object are variables. These are described after we have covered the principal requirements (see Section 7.2.3.5).

Before describing the construction of the design graph, we must detour slightly and explain the symbol table and lazy references.

---

[6] Perhaps, more usefully, on a large project, we could envisage the project having its own basic environment, probably built on top of the Ada environment.

[7] Standard HOOD requires identifies to start with a letter and only contain letters, digits and underscores.

[8] Changing the linkage information would require knowledge about *why* such links exists, which *Morpheus* does not possess.

```
data_analysis ( parse_tree )
begin

    walk parse_tree constructing
        symbol_table, entity_structure_table
        and object_structure_table ;

    iterate over symbol_table to resolve lazy references ;

    complete parent information in object_structure_table ;

    construct entity_tree from object_structure_table ;
    populate entity_tree by walking symbol_table ;

    construct linkage_table from entity_structure_table ;

    walk symbol_table and construct
        set of environmental_objects and
        set of data_entities ;

    return entity_tree, linkage_table, environmental_objects
        and data_entities ;

end
```

Figure 7.2: Data Analysis Phase

7.2.3.1 The Symbol Table

Central to Data Analysis is the construction of a symbol table. The structure of the symbol table is shown in Figure 7.2.3.1.

Symbol-Table = *Symbol-Table-Entry*-**set**

Symbol-Table-Entry :: *kind* : *Entity-Kind*
object : *Object-Name*
name : [*Entity-Name*]
signature : [*Signature*]
full-name : *Full-Name*

Signature :: *parameters* : [*Type**]
returns : [*Type*]

Entity-Kind = OBJECT|TYPE|DATA|CONSTANT|OPERATION|OPERATION-SET

Type :: *object* : *Object-Name*
name : *Entity-Name*

Full-Name = String: full name of entity including kind and signature

Object-Name = *Simple-Name*

Entity-Name = *Simple-Name*

Simple-Name = String of alphanumeric characters starting with a letter

Figure 7.3: Symbol Table

The symbol table contains information about all the symbols in the supplied design. The `entity_kind` field is required because HOOD permits 'overloading' of an objects's namespace, using the kind of entity to resolve conflicts. `Object` is just the name of the object in which the entity is declared.[9] `Name` is the simple name of the entity. `Full_name` is the fully expanded name of the entity, and is used to give (unique) full names to the Improvement Engine. The `signature` field, only applicable to constants and operations, contains a list of parameters and the return type.

Entities are added to the symbol table as they are found during a walk of the parse tree. Since Data Analysis supports lazy references (see Section 7.2.3.2) only entities defined in the **provided_interface** and **internals** are added, references in the **required_interface** are ignored.

Operations and constants cannot always be fully processed during the parse tree walk because their signatures may involve lazy references. Therefore there is a completion process for the symbol table at the conclusion of the tree walk to resolve fully all symbol table details.

7.2.3.2 Lazy References

Strictly speaking, according to the ODS's definition, any reference to an entity should always provide the object's name, and for operations the full signature. Whilst this is probably true for automatically generated designs, the authors' own experience of writing designs suggests that people are lazy and just provide sufficient information to uniquely identify an entity. Such lazy references can be resolved via the symbol table (see Section 7.2.3.1). This facility is not taken to its full limit, because (for example) *Morpheus* does not resolve (overloaded) operations on the basis of partial signatures. Either the signature is present or it is absent.

There is one problem with this approach caused by the ODS's definition. If an operation is overloaded with one form having a signature and another having an empty signature, it is not possible to determine if a reference is lazy or is in fact a reference to the operation with an empty signature.[10]

If *Morpheus* encounters ambiguous references or cannot find a matching symbol, it halts and expects the designer to clarify their intent.

7.2.3.3 Deriving the Object Hierarchy

As we have seen, the Improvement Engine requires knowledge about the hierarchical nature of the design. According to HOOD's semantic rules only objects and operation_sets can encapsulate other entities. In particular, operations cannot be nested.[11]

Operation_sets pose a problem for Data Analysis. They are only really a notational convenience for the graphical subset of HOOD. The difficulty is how should they be treated with respect to the object hierarchy. Two possibilities arise, either they are ignored or treated as an object. In the first case, the designer (potentially) loses cognitive information because the set that he has created is thrown away. In the latter case, operation_sets clearly are not real objects because they cannot encapsulate every kind of entity (e.g., objects and variables). Therefore regarding them as real objects gives them a greater significance than they really merit. Currently *Morpheus* does not accept operation_sets, and ignoring them seems the best practical solution. Such an approach would not significantly detract from the value of *Morpheus*'s suggestions; as operation_sets are purely a notational convenience.

HOOD's object structure is just a standard tree where a non-leaf node can have as many children as required. Three issues remain to be explained, namely: deriving the hierarchy from the supplied design, populating the constructed tree with entities other than objects, and handling object names.

As we have already noted HOOD does not require an entity to be declared before being referenced. In particular, an object can refer to entities in another object's **provided_interface** before the referenced object is specified. Additionally a non-terminal object must identify its children,

[9] Objects declare themselves.

[10] Perhaps HOOD should require a reference to an operation with an empty signature to specify an empty pair of parenthesis. At this time, we rejected this modification because of invalidating existing HOOD designs.

[11] *We believe that this should be seen as a limitation of* HOOD *because it does not support decomposition of (potentially) large operations.*

but a child object does not identify its parent. Fortunately, HOOD requires all object names to be unique within a design tree. This means that as objects are seen, we can construct a table showing the children (if any) of each object. Figure 7.4 shows the structure of the object structure table. On completion of walking the parse tree, we can therefore identify the parent (if one exists) of each object.

Object-Structure = *Object-Structure-Entry*-**set**

Object-Structure-Entry :: *object-name* : *Object-Name*
parent : *Object-Name*
children : *Object-Name*-**set**
siblings : *Object-Name*-**set**

Figure 7.4: Object Structure Table

This leaves us with two problems. Firstly, we may not have an object tree but rather an object forest, and secondly, we do not know the identity of the root object. Both of these problems can be overcome by creating a pseudo-object (called `$top_object$`) and making its children, those objects which do not have parents.

Entity-Tree = *Entity-Tree-Node**
Entity-Tree-Node = *Full-Name* | *Entity-Tree*

Figure 7.5: Entity Tree

The entity tree (see Figure 7.5) can now be constructed from the object structure table. The basic-entities can be inserted into the entity tree by scanning the symbol table, and placing each basic-entity declared in a particular object into the corresponding place in the entity tree.

In the supplied design each object has a user-specified name. In principle it is easy to pass this information into the Improvement Engine. However, the activity of the Improvement Engine will create new objects and destroy some existing objects and move entities between objects. Thus rendering the original object name misleading. It was therefore decided to ignore the supplied object name.

7.2.3.4 Deriving the Linkage Information

The second major component of the graph for the Improvement Engine is the set of links. For each entity these links show all the other entities upon which this entity directly depends. As the parse tree is being walked an entity structure record is created for each entity as it is encountered. Figure 7.2.3.4 shows the structure of the entity structure table.

Entity-Structure-Table = *Entity-Details*-**set**

Entity-Details :: *full-name* : *Full-Name*
kind : *Entity-Kind*
provides : *Entities*
requires : *Entities*
components : *Entities*

Entities = *Full-Name*-**set**

Figure 7.6: Entity Structure Table

At the very least each entity provides its own services. The `components` field is only really used by objects and operation_sets, since they are the only encapsulation entities in HOOD. The `requires` field identifies only those other entities directly required by the current entity.

$$\begin{array}{lcl} \textit{Linkage-Table} & = & \textit{Entity} \xrightarrow{m} \textit{Depends-On} \\ \\ \textit{Entity} & = & \textit{Full-Name} \\ \textit{Depends-On} & = & \textit{Full-Name}\text{-}\mathbf{set} \end{array}$$

Figure 7.7: Linkage Table

Once no more changes to the set of entity details is required, construction of the graph's linkage information (see Figure 7.7) is easy. Just use the `requires` field of each entity detail record. No data is generated for entities that have no dependencies, since its node has no outward edges in the underlying graph.

7.2.3.5 Secondary Information

As we noted earlier, there are two minor information requirements due to the nature of HOOD and the Improvement Engine.

$$\begin{array}{lcl} \textit{Environmental-Objects} & = & \textit{Objects} \\ \textit{Variables} & = & \textit{Entities} \\ \\ \textit{Objects} & = & \textit{Object-Name}\text{-}\mathbf{set} \\ \textit{Entities} & = & \textit{Full-Name}\text{-}\mathbf{set} \end{array}$$

Figure 7.8: Secondary Information

A HOOD design must be closed and part of the design philosophy of HOOD is to permit the separate development of individual design components by independent designers (see HOOD HUM (1996)). HOOD provides environmental objects (see Section 3.4.4) for describing the interfaces of objects not currently being designed. Since the content of such objects is outside the control of the designer of the current component, it was considered necessary for *Morpheus* to preserve the integrity of environmental objects and not move entities across such object boundaries. Therefore in addition to the information describing the design's hierarchical graph, the Data Analysis phase also passes a list of environmental objects (see Figure 7.8) to the Improvement Engine. This information can easily be derived from the parse tree, since each object's kind must be specified in the supplied design.

HOOD requires that variables[12] are only accessed by entities in their enclosing object, see rules V-6 on p. 87 and ODS-18 in Delatte et al. (1993, p.133). In order to *encourage* the observation of this requirement, *Morpheus* penalises violations by doubling up the description of an edge which requires a data item from another object. This has no impact on the message's meaning - but does increase the length (cost) of such a link. This approach is, of course, ugly—but serves its intended purpose. Hence, the Data Analysis phase must also pass a list of variables (see Figure 7.8) to the Improvement Engine. This list can be collected by scanning the symbol table generated during Data Analysis.

7.2.4 Improvement Engine

The Improvement Engine takes the graph generated by Data Analysis (see Section 7.2.3) and performs a search for better designs. A design is better if it has a shorter message length, constructed according to the rules in Section 6.2.3. The overall logic is shown in Figure 7.9.

The design of the Improvement Engine has three main problems to address:

1. Computing the message length.

[12] HOOD's data entities.

```
improvement_engine ( entity_tree, linkage_table,
                     environmental_objects, data_entities )
begin

    assign [linkage_table] to history_list ;

    copy linkage_table to best_design ;

    loop
        filter history_list giving active_list ;

        generate new designs from active_list giving new_designs ;

        calculate message lengths of new_designs ;

        append new_designs to history_list ;

        sort history_list by message length ;

        update best_design if top of history_list is best ;
    endloop ;

    report best_design ;

end
```

Figure 7.9: Improvement Engine Phase

2. Generating alternative designs.

3. Type of search strategy to employ.

The first of these is simple to implement based on the work in Chapter 6. However it is expensive[13] in processing time, which given the large number of design alternatives to consider is a burden and some form of approximation would be of significant benefit.

7.2.4.1 Generating Alternative Designs

The Improvement Engine has a set of rules for constructing alternative designs from an initial design. There are four creation rules and two style rules.

The creation rules are:

Move Node which looks for a basic entity with a high external degree and attempts to move the entity into another object. Hence trying to reduce the number of inter-object links.

Raise Node which like the previous rule, looks for a basic entity with a high external degree and attempts to move the entity up the hierarchy one level. Hence trying to reduce the number of intervening objects through which links must pass.

Split Object which looks for an object with a largish number of components, and splits the object. Thus reducing the size of large objects.

Join Objects which merges the contents of sub-objects into their parent. Thus reducing the object count.

The HOOD style rules are:

[13] This can be improved by use of an incremental model, see Section 7.3.1.

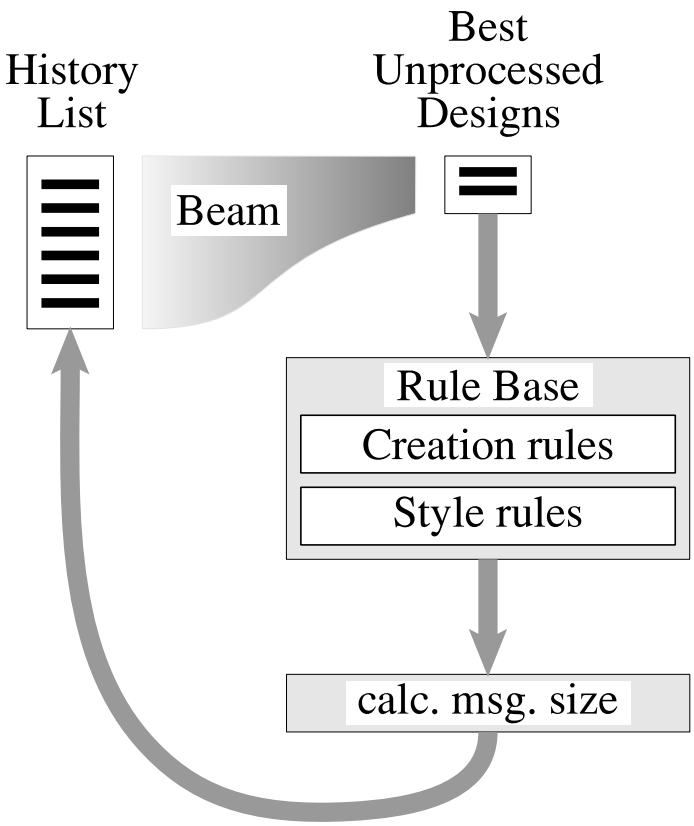


Figure 7.10: *Morpheus*'s Search Strategy

Remove Singletons Objects with only one element, are removed, by merging the single child into the object's parent.

No Mixed Objects HOOD does not permit an object to contain basic entities and other objects. Currently, this rule is not used.

For each search iteration, the creation rules provide as many alternative designs as possible. These designs are then (potentially) modified by the style rules to remove unacceptable designs. This generates a new set of designs for consideration.

7.2.4.2 Search Strategy

Morpheus's search strategy is simple, see Figure 7.10; *Morpheus* hill climbs with a narrow search beam (Thornton and du Boulay, 1992). That is it keeps a small active list (about 8–20, see Figure 7.11) of the currently best designs, and applies the rule base (see Section 7.2.4.1) to generate new designs. The best designs from each iteration are kept as the input into the next iteration.

$$\textit{Active-List} = \textit{Active-List-Entries}^*$$

$$\begin{array}{lll} \textit{Active-List-Entries} :: & \textit{message-length} & : \mathbb{R} \\ & \textit{design} & : \textit{Design} \end{array}$$

$$\textit{Design} = \textit{Linkage-Table}$$

Figure 7.11: Active List

This approach guarantees that the current best is always kept, but does run the risk of cycles and blind allies. To help alleviate this problem, a larger history list (about 30–45, see Figure 7.12)

$$\textit{History-List} \;=\; \textit{History-List-Entries}^{*}$$

$$\begin{array}{lll} \textit{History-List-Entries} :: & \textit{expanded} & : \mathbb{B} \\ & \textit{active-entry} & : \textit{Active-List-Entry} \end{array}$$

Figure 7.12: History List

of the best seen designs is kept. Newly generated designs are compared to the history list, and previously seen designs are discarded. Designs in the history list which have previously been used as input to the rule base are marked, and at each iteration a new active list is generated from the best unused members of the history list.

The search stops either after a predefined number of iterations or when the history list contains no more unused entries. At which stage a report on the *best* design is produced. This report identifies the proposed structure of the system.

7.3 Limitations

Having examined the implementation of *Morpheus*, it is useful to stand back a little, and look at the main limitations of the current prototype.

7.3.1 Physical Resources

The biggest problem that *Morpheus* has at the moment[14] is *speed*. It requires about 14 hours of elapsed time[15] to process the *TriviCalc* design (see Section 8.3). *TriviCalc* has some 300 basic entities, distributed over 17 objects with in excess of 1050 links.

There are three significant time consumers in *Morpheus*, all in the Improvement Engine, namely: generation of new designs, message length calculation[16] and comparing different designs for equality. The latter two principally arise because of the sheer number of designs considered.

Closely related to the problem of speed, is the consumption of memory. *Morpheus* requires large quantities of memory for what can only be considered moderate sized designs. Whilst this problem has not been fully explored, its origin appears to be closely related to the number of new designs generated by the rule base.

Major benefits could be realised by having some heuristics for immediately rejecting 'poor' designs, and having a method for finding the approximate message length. However, this requires further work. An obvious side-effect of such improvements would be that a larger number of designs could be considered on each iteration of the search cycle.

Postscript Recently, an improved theoretical model for calculating message length has been developed. This model is fully described in Section 6.3. This model should enable an incremental calculation of message length based on changes in the underlying design rather than the current implementation's full calculation of message length for each change.

It is anticipated that this change will also reduce memory requirements, since only the differences for each iteration might need to be kept.

Currently, *Morpheus*, does not incorporate this new theoretical model or incremental message length evaluation and future work should include the introduction of these changes. We believe that these modifications will have significant practical benefits.

[14] Since these timings were taken, we have developed a better model of how message length is related to node degree. With this improved model, we believe that the time taken to calculate a message's length would be cut significantly.

[15] Using a Sun Ultra Enterprise 2 server, employing one 300MHz UltraSPARC processor, with a 1Mb cache.

[16] This will be improved by the application of the theoretical model in Section 6.3.

7.3.2 Missing Information

As we have noted before *Morpheus* cannot handle partial designs or designs with missing information. This problem has implications for the deployment of *Morpheus* in the early stages of design, when its suggestions might be most useful.

It is unworthy of excellent men to lose hours like slaves in the labour of calculation which could be relegated to anyone else if machines were used.

GOTTFRIED WILHELM VON LEIBNITZ (1646–1716)
German philosopher and mathematician

Chapter 8

Empirical Evidence in Support of Ψ

Synopsis

This chapter presents some empirical evidence in support of our hypothesis that Ψ is a reasonable complexity measure; which permits trade-offs between various engineering properties. We present a few small examples to illustrate this claim. Then a few small real designs are considered. We end with an application of *Morpheus* to *TriviCalc*, a working spreadsheet, and consider if the design has been improved.

In this chapter we examine empirical evidence to support Ψ as a complexity measure.

Note 8.1. *All the graphs in this chapter were produced by* `dotty` *(Koutsofios and North, 1994) which attempts to minimise edge crossings.*

8.1 Initial Experiments

We start by looking at a few small (contrived) systems, to see how Ψ behaves. We start all these designs off consisting of just a single object (unless otherwise noted), i.e., all basic entities in the same object. We can regard this as representing a null hypothesis of no hierarchical structure. There are no data or environmental entities in the experiments in this section.

8.1.1 Varying Group Size

In this experiment, we look at the effect of increasing group size. The results are shown in Table 8.1.

The system consists of two sets, each of n basic entities. Each set has high cohesion. Specifically, each basic entity in the set has one link to or from every other basic entity; the links are from low numbered entities to higher numbered entities. There is only one link between the two sets; from $A1$ to $B1$.

The first most striking observation from Table 8.1 is how fast Ψ increases as new basic entities are added. The reduction achieved by forming a hierarchical structure is quite impressive, for example the case of 10 basic elements achieves a reduction of 21% which is significant.

The corresponding figures are interesting, particularly Figure 8.3 as it seems unlikely that a human designer would have created this arrangement. It also shows just how complicated small systems are to understand. Observe how the structures of the A and B groups are similar except for the top-level entity in B (labelled $B1$). We can explain this difference because of the single inter-object link between $A1$ and $B1$.

Figure 8.4 shows an example of the structure under our null hypothesis.

number of nodes	Ψ under null hypothesis	final Ψ	object structure	see figure
3	58.5	58.5	(A1,A2,A3,B1,B2,B3)	8.1
5	179.8	160.8	((B1,B2,B3,B4,B5) A1,A2,A3,A4,A5)	8.2
10	901.9	706.8	((((B0,B4,B5,B6,B7,B8,B9) B2,B3) B1) ((A0,A4,A5,A6,A7,A8,A9) A2,A3) A1)	8.3

Table 8.1: Effect of Increasing Group Size

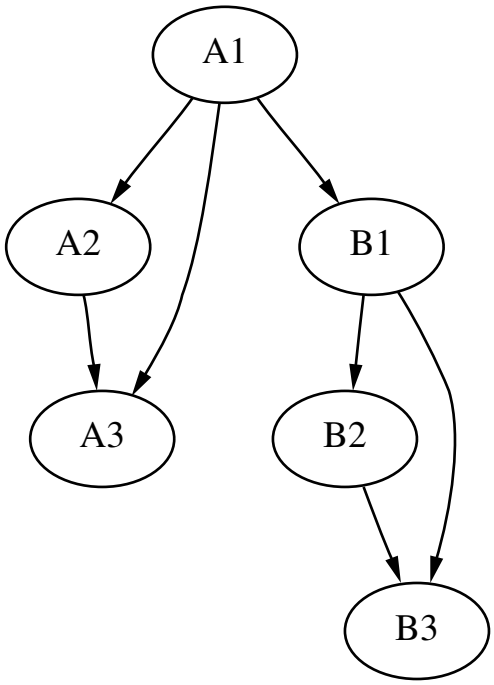


Figure 8.1: Grouping with 2 groups of 3 entities

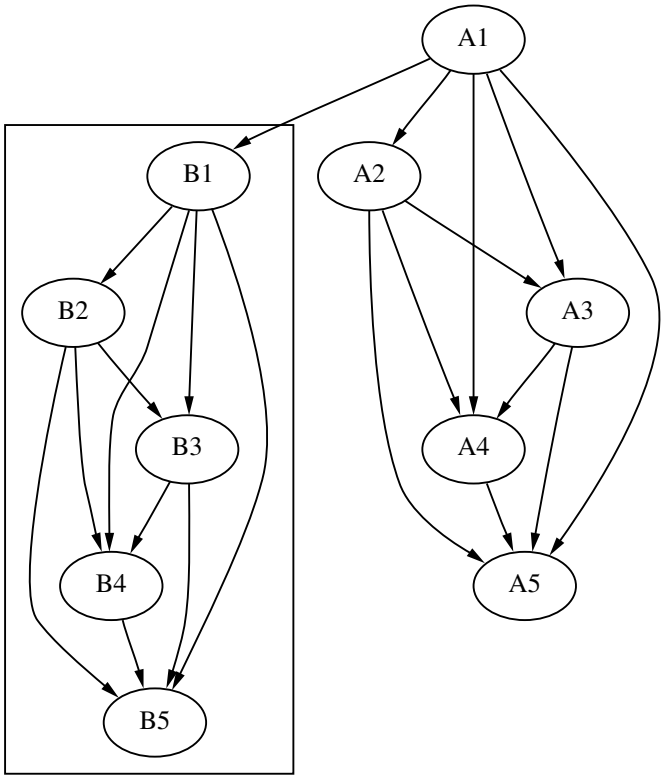


Figure 8.2: Grouping with 2 groups of 5 entities

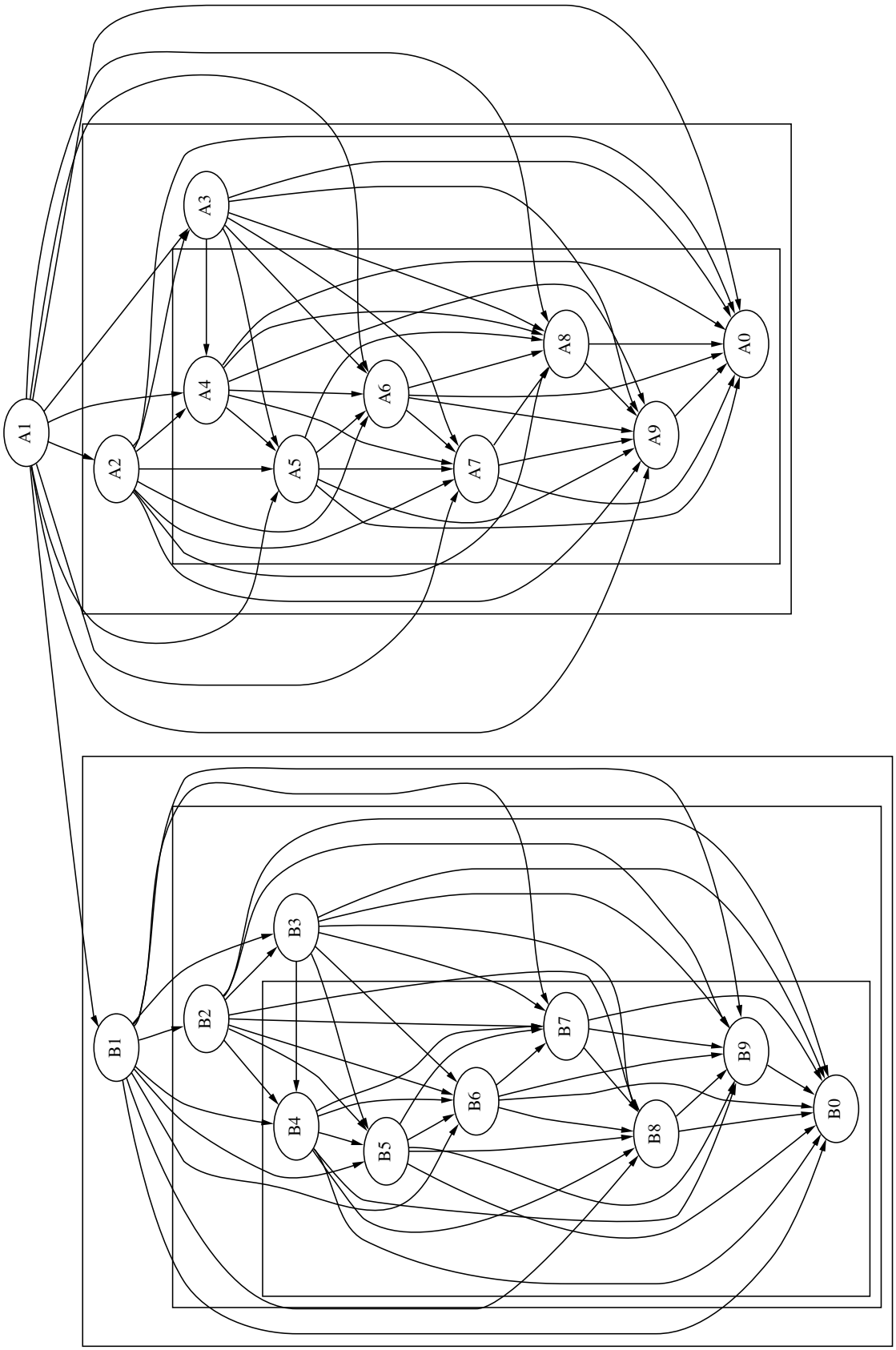


Figure 8.3: Grouping with 2 groups of 10 entities

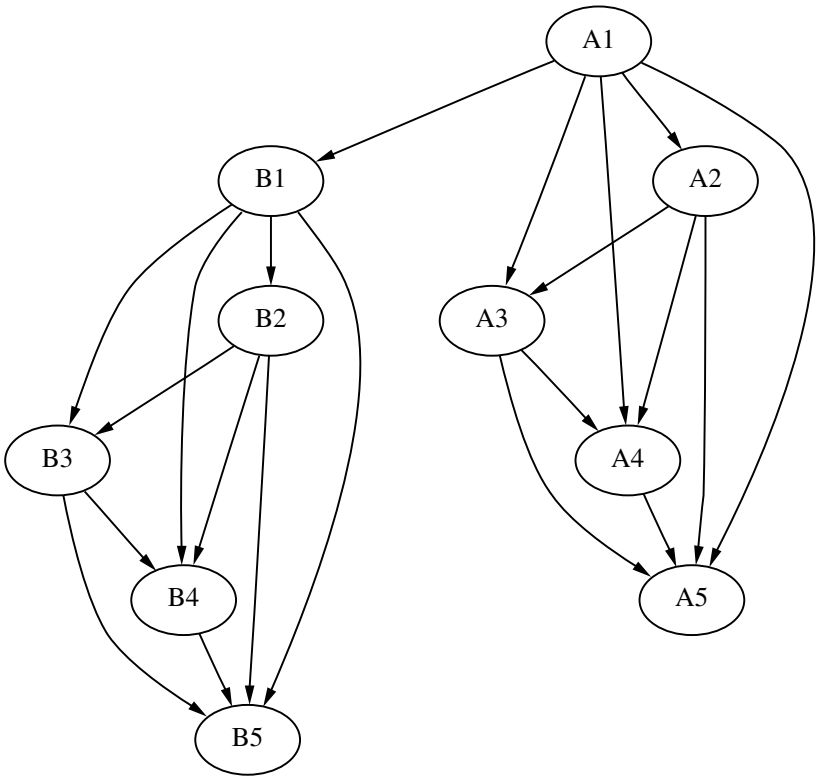


Figure 8.4: No structure with 2 groups of 5 entities

8.1.2 Moving Basic Entities

From Section 8.1.1 we know that for groups of 5 with high internal cohesion and low coupling that the Ψ favours the structure shown in Figure 8.2. In these two experiments we moved basic entities from group *A* to group *B* to confirm that *Morpheus* was making the right choices.

In the first experiment, we moved *A*1 into the *B* group. This produced a complexity of 172.8 bits, compared to the previous best of 160.8 bits.

In the second experiment, we moved *A*5 into the *B* group. This produced a complexity of 174.6 bits.

We can explain both of these results, by observing that the two moves increased the inter-object coupling, and hence we would regard it as sub-optimal.

8.1.3 Reducing Cohesion

Starting from the same base as the previous experiment, we wanted to study the effect of reducing intra-object cohesion. We did this by progressively removing internal links inside the *B* group, whilst leaving the *A* group unchanged.

We had to be careful not to remove all the links from one basic entity in succession, otherwise *Morpheus* would just treat the basic entity as unattached, and place it in the best position, without regard for our desire to keep a 'cohesive' group.

The results of this experiment are shown in Table 8.2.

number of links	Ψ under null hypothesis	final Ψ	object structure	see figure
10	179.8	160.8	((B1,B2,B3,B4,B5) A1,A2,A3,A4,A5)	8.5
9	171.5	154.6	as above	8.6
8	163.0	148.1	as above	8.7
7	154.0	141.4	as above	8.8
6	145.0	134.7	as above	8.9
5	135.6	128.0	as above	8.10
4	125.6	120.9	as above	8.11
3	115.5	113.6	((B1,B2,B3,B4) A1,A2,A3,A4,A5,B5)	8.12
2	105.2	105.2	(A1,A2,A3,A4,A5,B1,B2,B3,B4,B5)	8.13
1	94.6	94.6	as above	8.14
0	83.4	83.4	as above	8.15

Table 8.2: Effect of Reducing Cohesion within a Group

We can clearly see from Table 8.2 that the first seven experiments all retained the same group structure. It was not until the number of links in group *B* dropped below four, that the basic structure was changed. This occurred when the last link to *B*5 was broken. We can understand *Morpheus*'s decision to place *B*5 into the *A*, by realising that whilst $\log_2^* x$ is a strictly monotonically increasing function, its rate of growth decreases as x increases.

Once the number of links has dropped below three, there is no justification for keeping *B* as a separate group. Not unreasonable, as its internal cohesion has virtually gone.

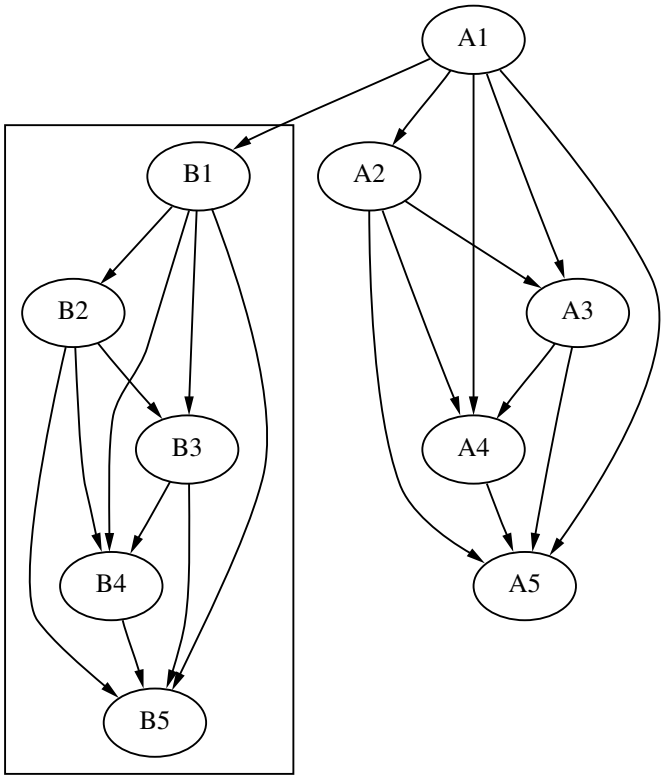


Figure 8.5: Cohesion with 2 groups, one with 10 links

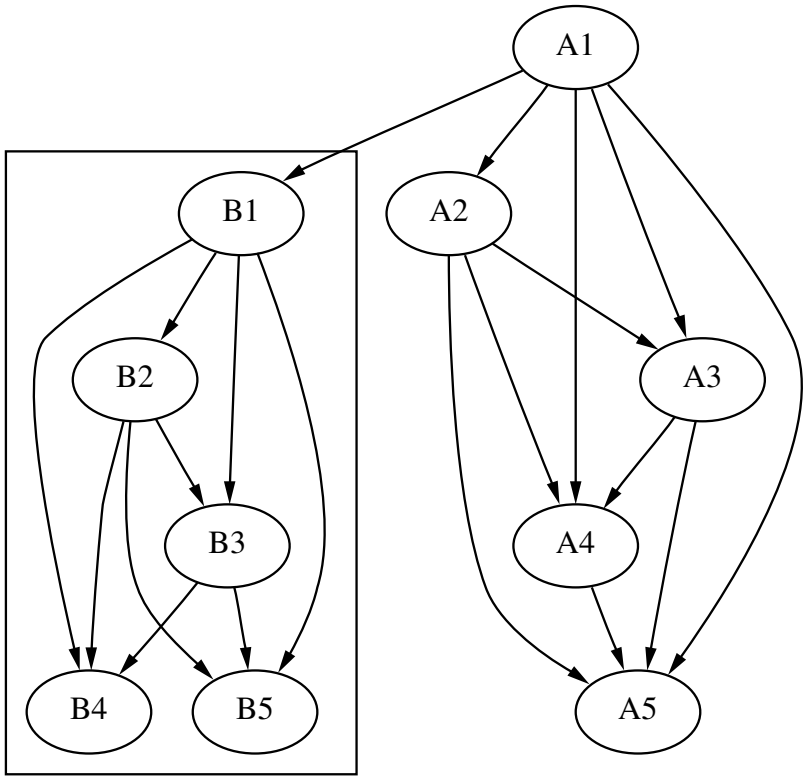


Figure 8.6: Cohesion with 2 groups, one with 9 links

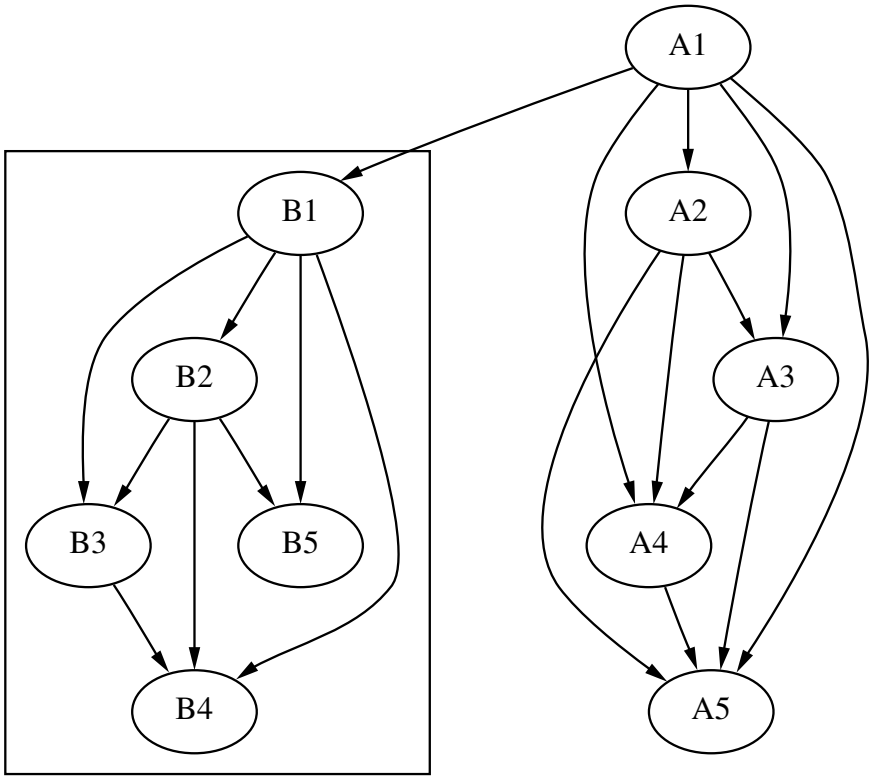


Figure 8.7: Cohesion with 2 groups, one with 8 links

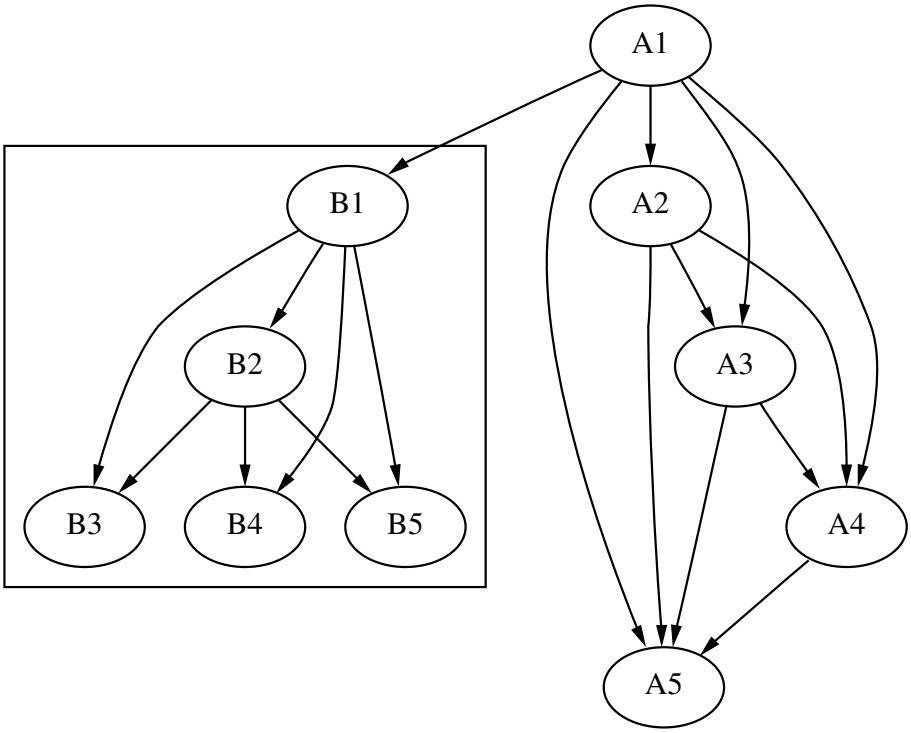


Figure 8.8: Cohesion with 2 groups, one with 7 links

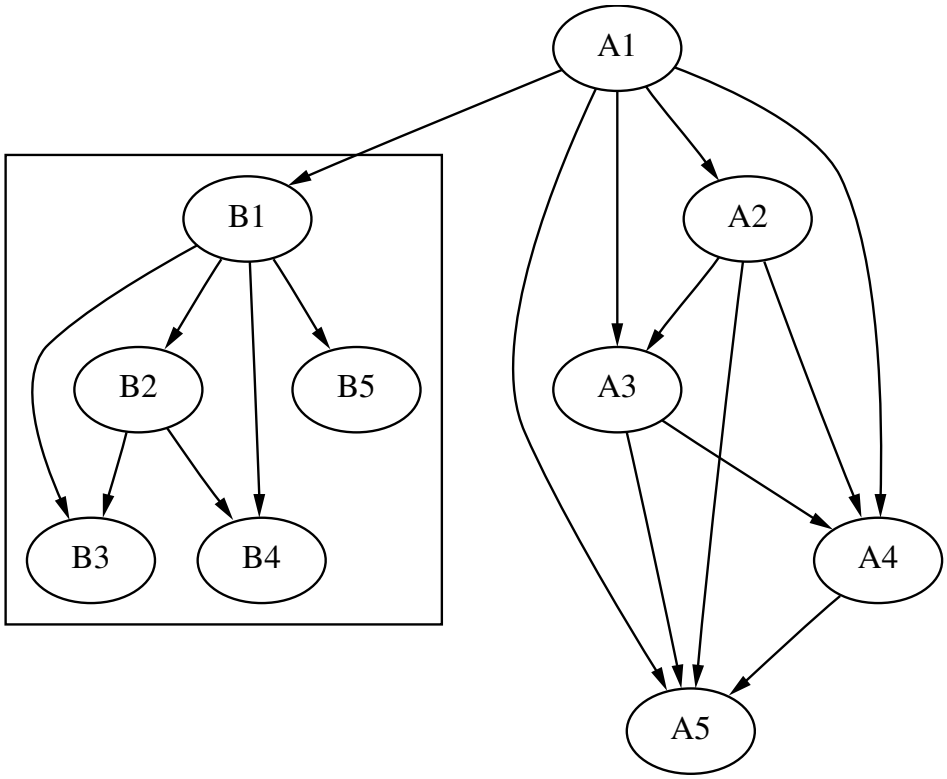


Figure 8.9: Cohesion with 2 groups, one with 6 links

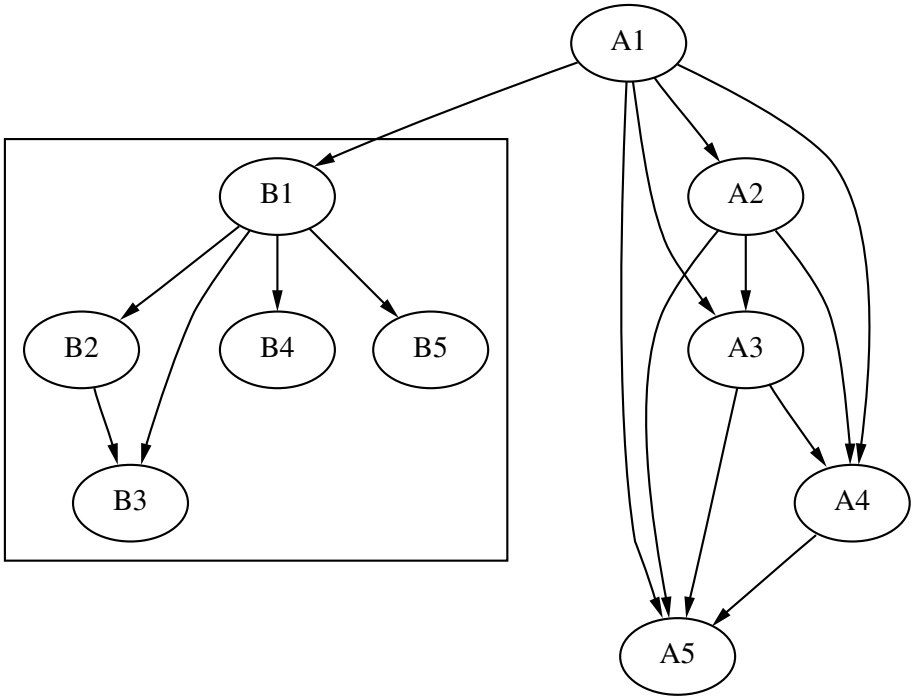


Figure 8.10: Cohesion with 2 groups, one with 5 links

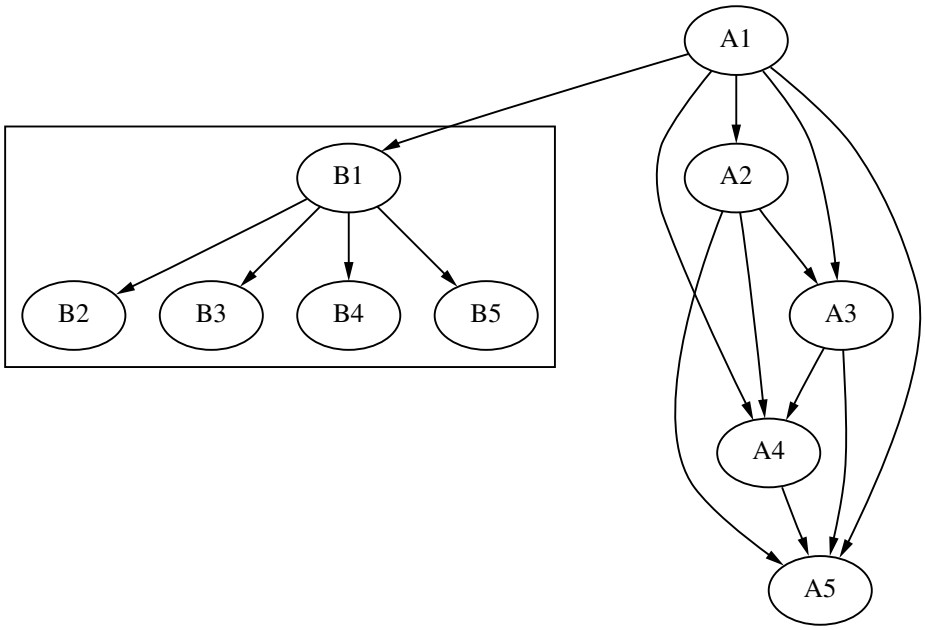


Figure 8.11: Cohesion with 2 groups, one with 4 links

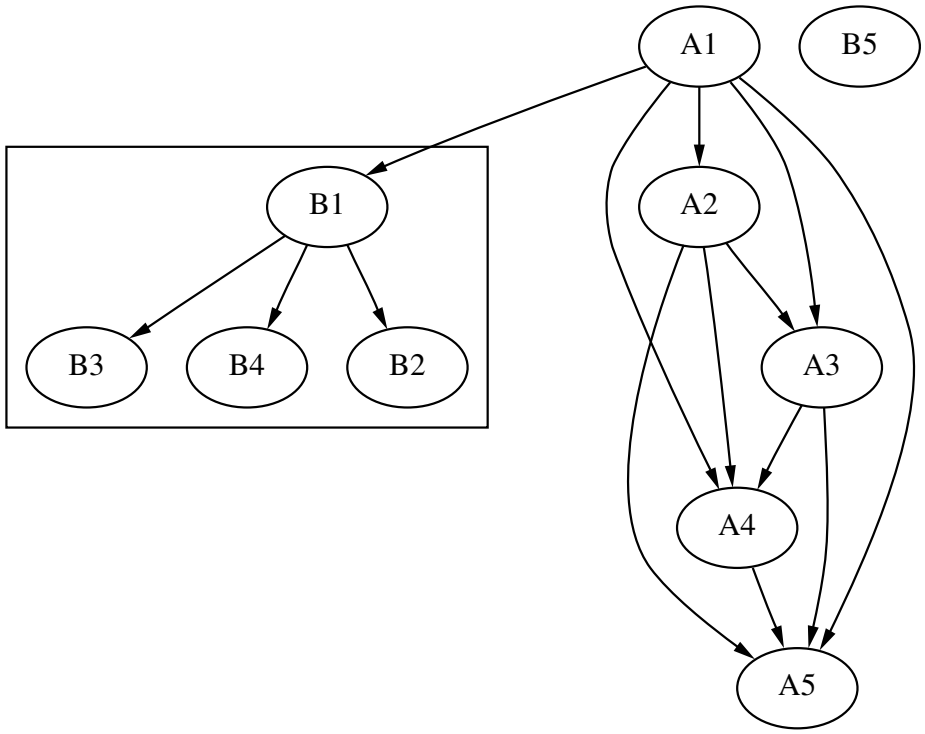


Figure 8.12: Cohesion with 2 groups, one with 3 links

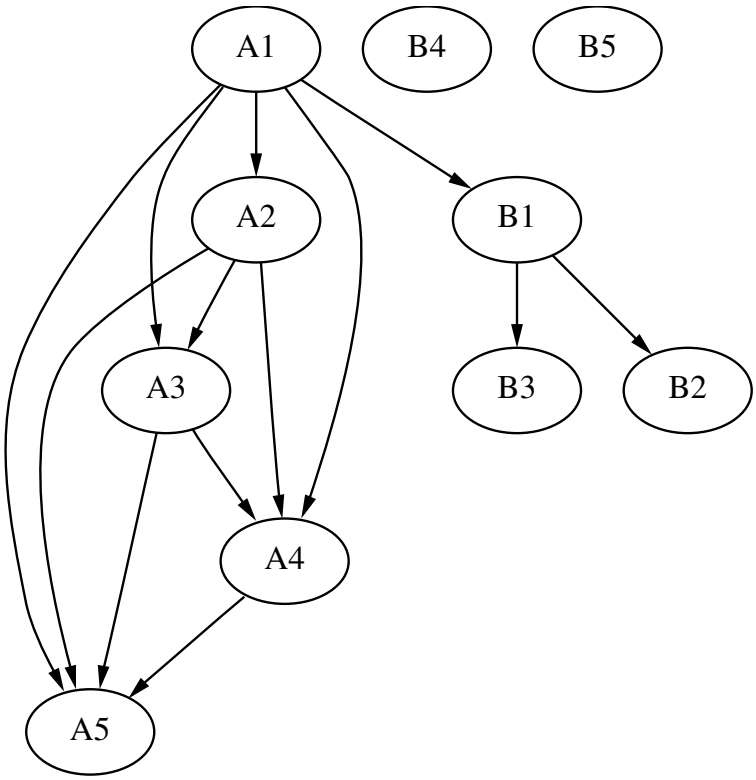


Figure 8.13: Cohesion with 2 groups, one with 2 links

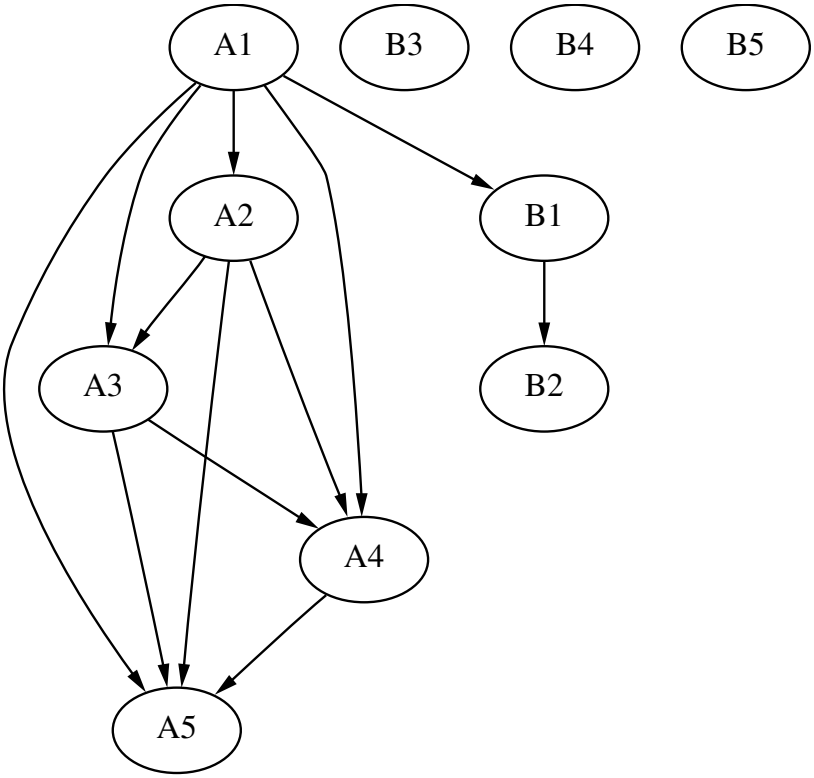


Figure 8.14: Cohesion with 2 groups, one with 1 link

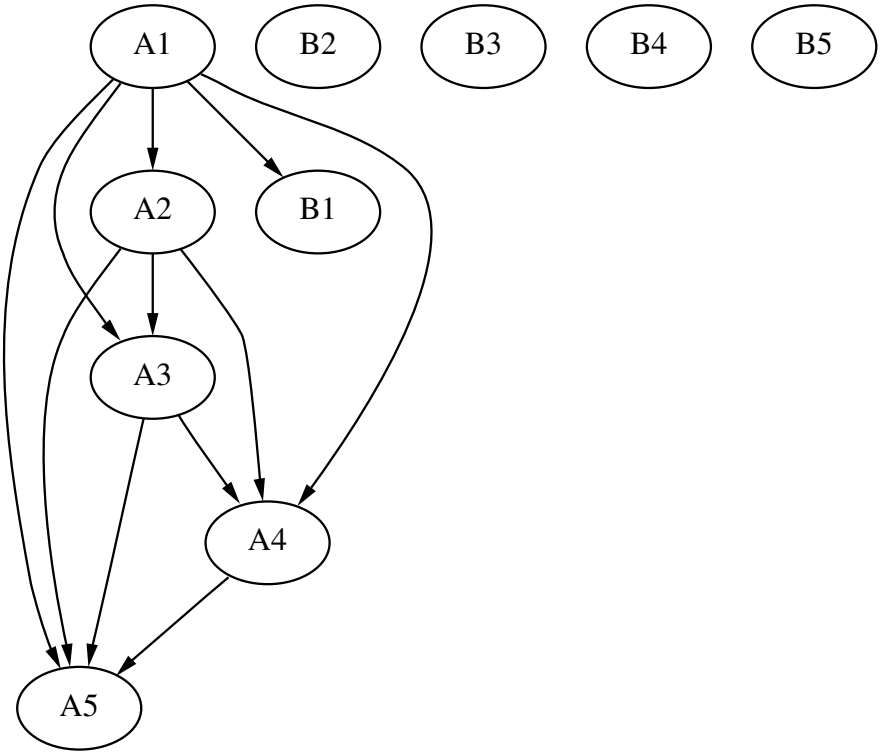


Figure 8.15: Cohesion with 2 groups, one with no links

8.1.4 Increasing Coupling

Once again starting from the same base as the previous experiment, we wanted to study the effect of increasing inter-object coupling. We did this by progressively adding links between the A and B groups.

We did not want all the new links to go from one basic entity to another basic entity or this would have resulted in a system with a few highly cohesive basic entities rather than just increasing the links from A to B.

The results are shown in Table 8.3. The double line under the entry starting 5, is a reminder that the linkage structure underwent a change. Between 0 and 5 (inclusive), we formed links from group A to B, by just adding a link from A_i to B_i. However, after 5 we formed additional links by adding links from B_i to A_i, the intention being to avoid just making B a sub-group of A. In retrospect, this idea was correct in principle, but we would have been better to add links from B_i to A_{6-i} thus avoiding too much cohesion between specific basic entities.

number of links	Ψ under null hypothesis	final Ψ	object structure	see figure
0	172.3	157.2	((B1,B2,B3,B4,B5) A1,A2,A3,A4,A5)	8.16
1	179.8	160.8	as above	8.17
2	187.4	170.0	as above	8.18
3	195.2	178.9	as above	8.19
4	203.1	187.8	as above	8.20
5	211.0	196.5	as above	8.21
6	218.8	205.8	((A1,A2,A3,A4,A5) (B1,B2,B3,B4,B5))	8.22
7	226.5	213.8	as above	8.23
8	234.	225.9	(((A3,A4,A5,B3,B4,B5) A2,B2) A1,B1)	8.24
9	242.0	232.3	as above	8.25
10	249.7	238.7	as above	8.26

Table 8.3: Effect of Increasing Coupling between Groups

Looking at Table 8.3, we see that complexity rises as new inter-group links are added, as expected. More interestingly, at 6 links, the two groups are formed into two 'equal' sub-objects, which we speculate is caused by trade-offs on the complexity of adding a new object containing 6/7 edges versus the additional complexity of more links between the an encapsulated object and its parent. We can explain this by observing that the (partial) message length needed to distinguish between only two objects is 1 bit, as opposed to adding more bits to the message to distinguish between nearly equal node degrees.[1]

Interestingly, after 7 links between the groups, the strategy changes again, this time to having objects nested 3 deep. We hypothesis that this arrangement is arrived at by the interaction of object size (number of encapsulated entities) versus the message length for describing edges. Unfortunately at this time we cannot offer a more detailed explanation.

[1] It is well known that the length of a message is maximised when all the symbols have equal probabilities.

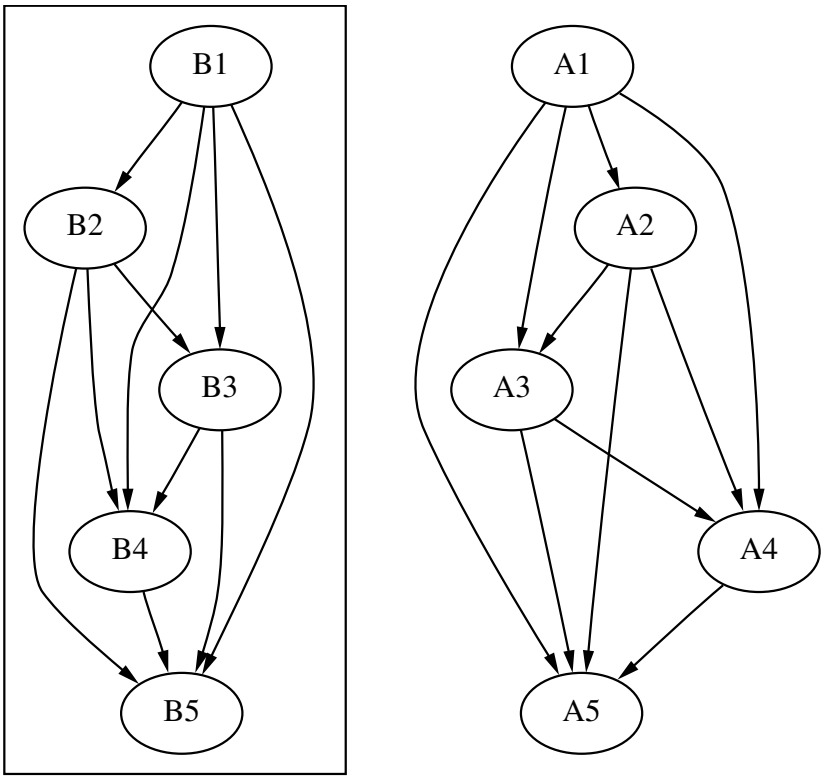


Figure 8.16: Coupling with 2 groups, and no links between groups

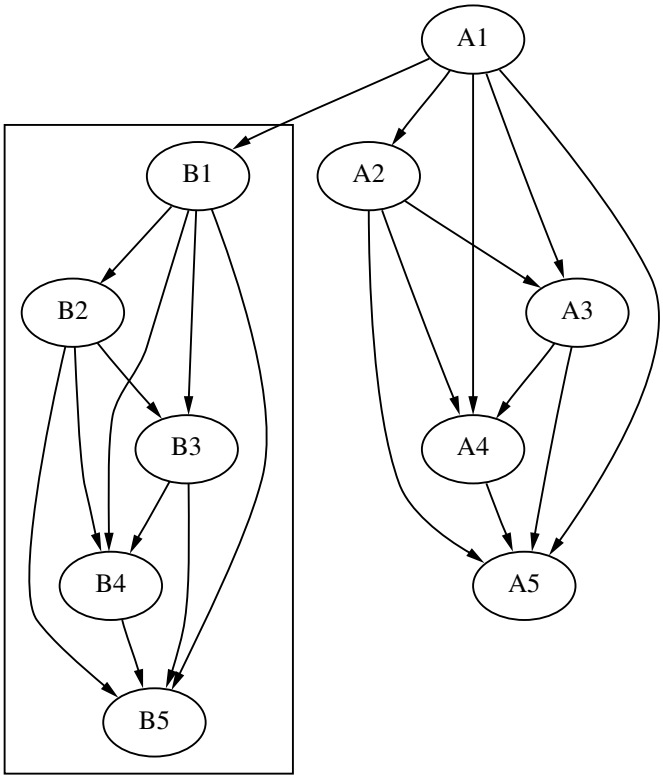


Figure 8.17: Coupling with 2 groups, and 1 link between groups

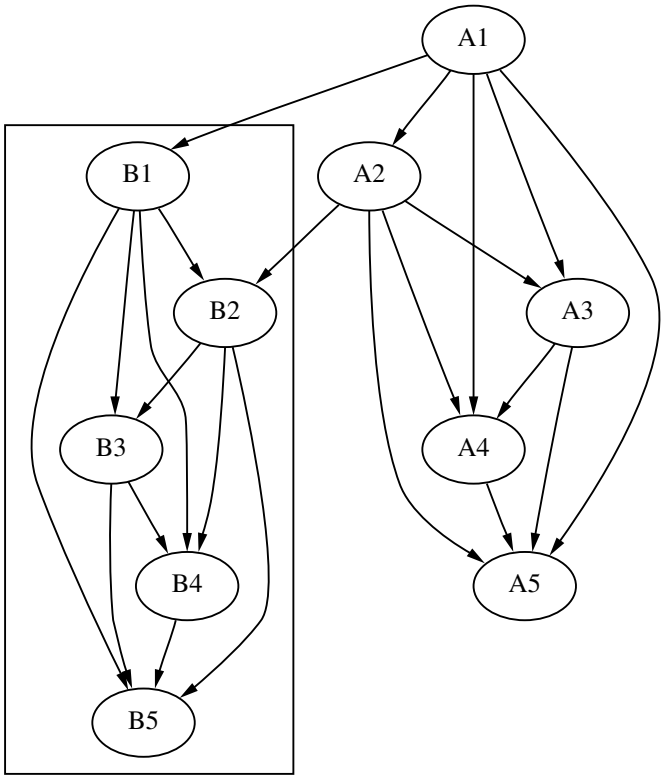


Figure 8.18: Coupling with 2 groups, and 2 links between groups

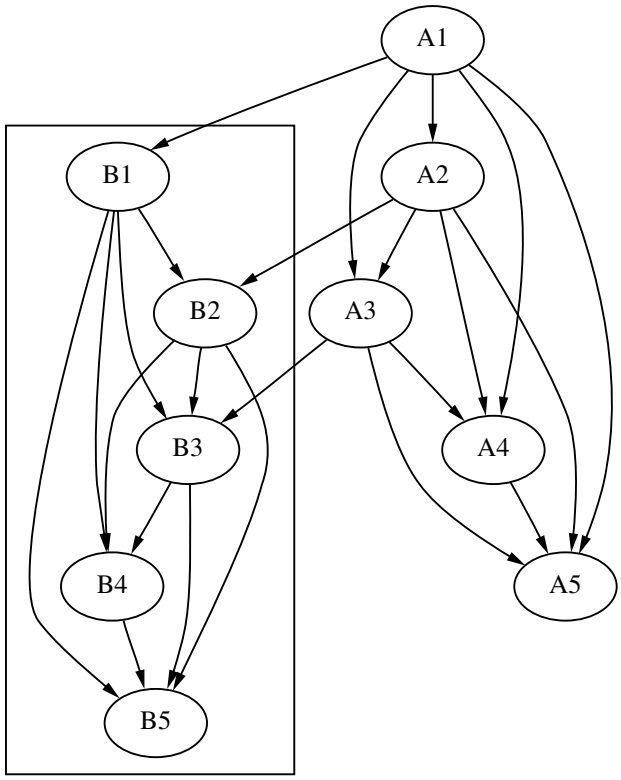


Figure 8.19: Coupling with 2 groups, and 3 links between groups

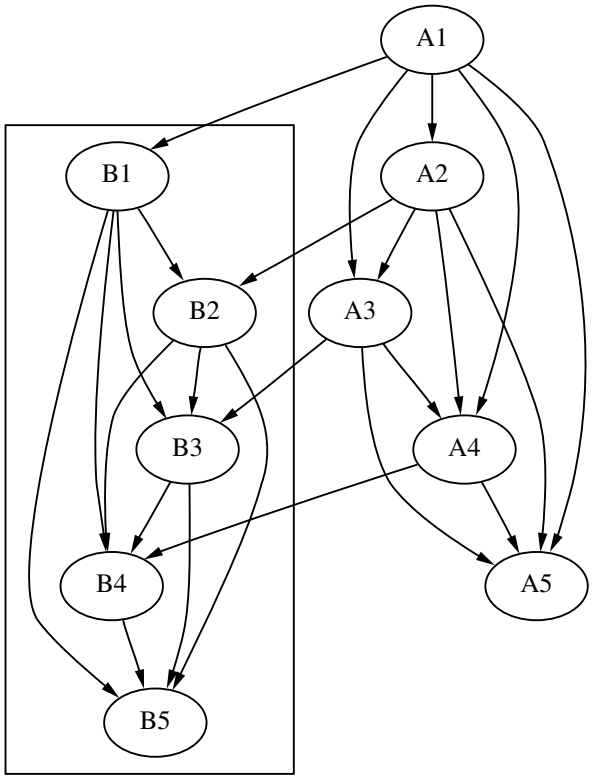


Figure 8.20: Coupling with 2 groups, and 4 links between groups

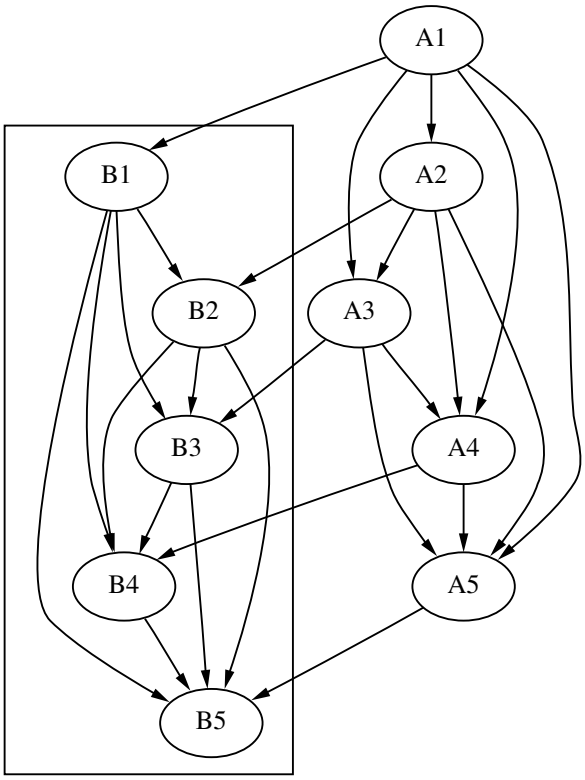


Figure 8.21: Coupling with 2 groups, and 5 links between groups

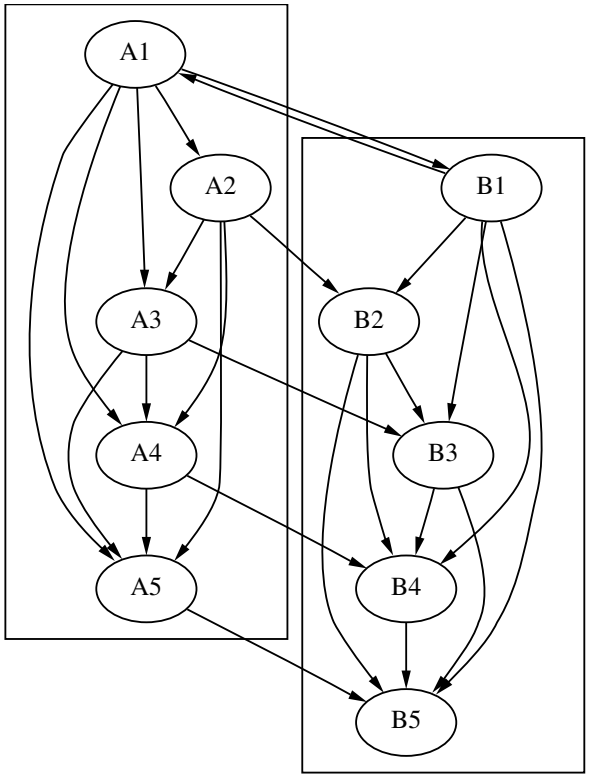


Figure 8.22: Coupling with 2 groups, and 6 links between groups

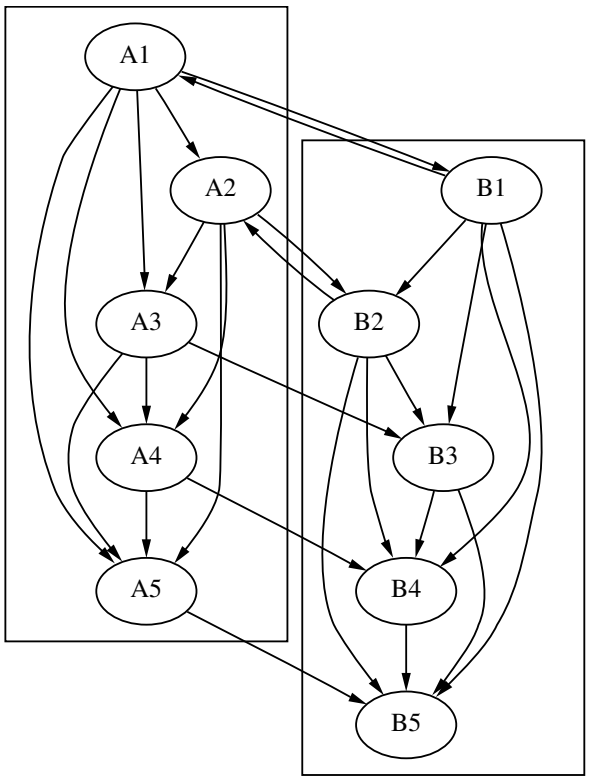


Figure 8.23: Coupling with 2 groups, and 7 links between groups

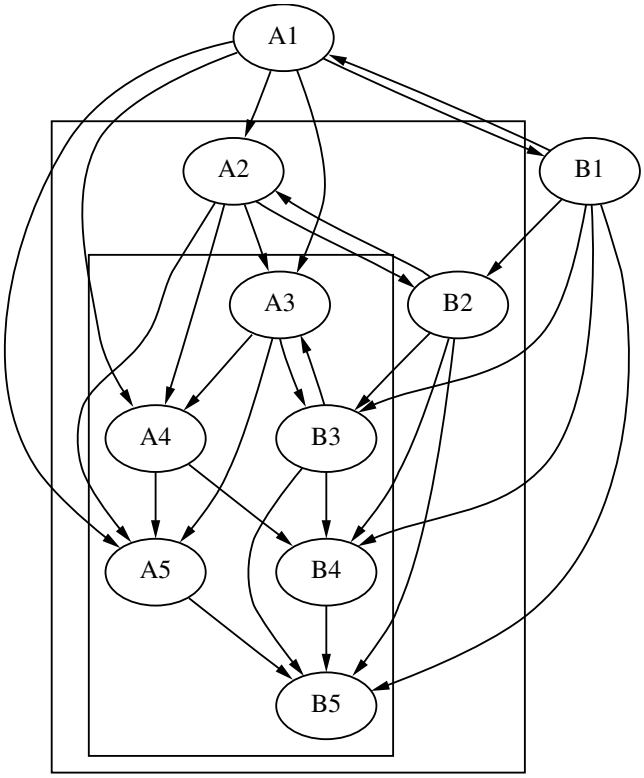


Figure 8.24: Coupling with 2 groups, and 8 links between groups

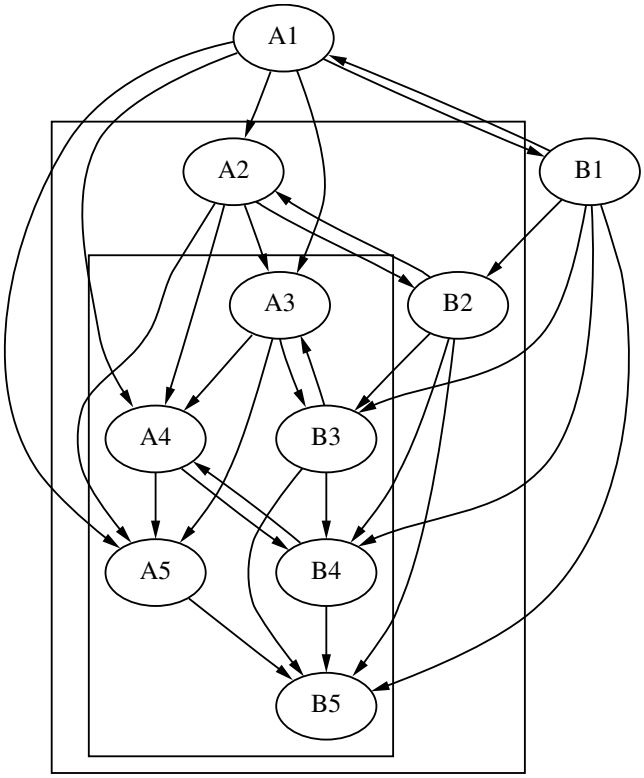


Figure 8.25: Coupling with 2 groups, and 9 links between groups

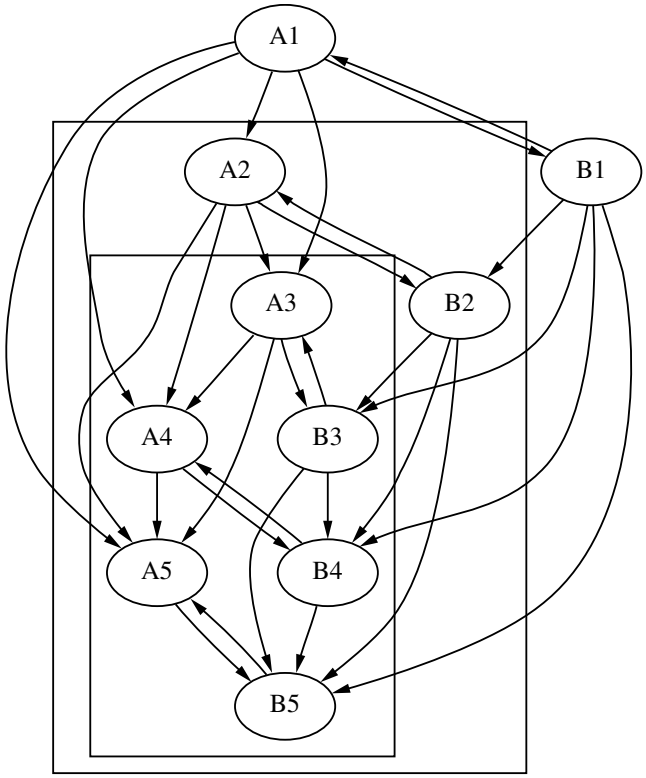


Figure 8.26: Coupling with 2 groups, and 10 links between groups

8.2 A Small Example: Traffic Lights

In this section we return to the example of traffic lights which we started in Section 3.2. The ODS was updated to capture the extra design information provided by Augmented HOOD (see Section 7.1 and Appendix A). The exact changes are described below.

Operation `lights.change` had an **operation_requirements** section added to its **pseudo_code** section, showing that `lights.change` updates the variable `other_road`.

```
OPERATION_REQUIREMENTS
  WRITES_TO
    other_road ;
END_OPERATION_REQUIREMENTS
```

Operation `seconds.count` had a **pseudo_code** section added, showing that a number of internal variables were updated during the course of its operation.

```
PSEUDO_CODE
  OPERATION_REQUIREMENTS
    WRITES_TO
      elapsed ;
      ac_present ;
      bd_present ;
      current_green_pair ;
  END_OPERATION_REQUIREMENTS
```

Operation `traffic_sensors.check` had a **pseudo_code** section added, showing that a hardware latches were read during the course of its operation.

```
PSEUDO_CODE
  OPERATION_REQUIREMENTS
    READS_FROM
      ac_sensors ;
      bd_sensors ;
  END_OPERATION_REQUIREMENTS
```

Objects `traffic_lights` and `text_io` were left unchanged.

We conducted a number of experiments with this design, the results are summarised in Table 8.4

design	no environment		with environment	
	Ψ	see figure	Ψ	see figure
Original	346.97	8.27	326.04	8.27
Flat	306.88	8.28	306.79	8.29
Morpheus's proposal	276.78	8.30	283.36	8.31

Table 8.4: Experiments with Traffic Light Design

Recall that in these design graphs, trapeziums represent datatypes and boxes represent variables.

We took the original design as proposed by Robinson (1992a), and calculated it complexity, both with `text_io` as an environmental object and just as part of the design. This was contrasted

with a flat design containing no objects other than those needed when `text_io` was made an environmental object.

We then ran *Morpheus* on Robinson's design (with and without `text_io` as an environmental object). Not surprisingly the complexity of *Morpheus*'s chosen solution with an environmental object is larger than that without an environmental object. We attribute this to having a greater freedom to rearrange the design without environmental objects.

More interestingly, the two designs produced by *Morpheus* are markedly similar, which gives us more confidence in *Morpheus*'s behaviour. The principal changes between the original design and *Morpheus*'s proposal were

- Moving datatype `traffic_lights.road` into object `lights`. This reduces some of the coupling between objects. Further, this is technically better HOOD, because a parent object should not contain basic entities.

- Moving datatype `traffic_sensors.present` into object `seconds`. This reduces the coupling between `traffic_sensors` and `seconds`, since `seconds` has two variables declared as type `present` and an operation requiring type `present`, whereas `traffic_sensors` only has two operations depending on `present`. Note however that our complexity measure does not distinguish between different kinds of connections.

  Perhaps examining the kind of connection is something that should happen in future, but this would detract from the generality of our complexity measure Ψ. We believe that if such comparisons are required, they would be better handled by changes to the style section of *Morpheus*'s rule base.

- The operation `traffic_lights.second` (labelled `tl.second` in our diagrams, for uniqueness) is moved into object `seconds`.

We can well understand why this last move would occur (reducing coupling between `traffic_lights` and `seconds`). Unfortunately, it highlights two potential difficulties. First, how to handle HOOD's implementation link, and second, keeping the design's entry-point (if there is one) at the top of the object hierarchy.

The first problem should, we believe, be handled by modification to *Morpheus*'s Data Analyser, such that references to basic entities which have an implementation link should be redirected to the entity responsible for actually providing the required service. This would only be a relatively minor change to the Data Analyser, but might have repercussions for the presentation of *Morpheus*'s results—a subject which is outside the scope of this thesis.

The second problem is as yet an open issue.

8.2.1 Discussion on the Traffic Lights Design

In this section we will look at the software design underlying the Traffic Lights example, comparing the before (see Figure 8.27) with the after (see Figure 8.31). The changes are described in the above section. A quick count of the number of service requirements (links) flowing out of all objects, shows a drop from 15 to 10 which supports the authors' intuitive claim that the overall system coupling has fallen. Looking at each object in turn (see Section 3.6).

Traffic_Lights Basically this object has not changed (certainly from its outward appearance). The operation `tl.second` and datatype `road` have moved into internal objects, but to some extent `tl.second` is just a design artifact to provide the object `traffic_lights` with a callable interface (via an interrupt). The datatype `road` is interesting because it is a global datatype used across `traffic_lights`. We think that from a design perspective `road` would be better as a global datatype rather than as part of the object `lights`; but we accept that strictly this is poor HOOD. In a similar view we would have made the datatype `present` global because it pervades several objects.

We believe that this difference between *Morpheus*'s suggestion and our opinion is due to the use of $\log_2^*$ to represent natural numbers (see Section 5.2.3).

Lights The `lights` object is (apart from the datatype `road`) a well-defined conceptual object, just providing services to manipulate the colour of the roads' traffic lights. Obviously such an object requires the operation `change` and the datatype `colour` to be external attributes.

The data `other_road` is an oddity, but it is part of the design of the `change` operation and so cannot be removed from the object `lights`. We would have preferred some form of dedicated `road` object providing a datatype `road` with services for selecting which road was allowing traffic to flow and its opposite road. This however is not what Robinson (1992a) specified. This illustrates the significant limitation of not having knowledge of *purpose* of a design, but we see this as a major piece of future work.

Traffic_Sensors The object `traffic_sensors` has remained well-defined, encapsulating private data, operations and datatypes. The only real issue with `traffic_sensors` is the location of the datatype `present`. However, as we have already said, we believe `present` would be better as a global datatype. We note that moving `present` from `traffic_sensors` to the object `seconds` reduces its external link count from 3 to 2 and does reduce the object's coupling.

This raises a general question, should links to datatypes count as strongly as other kinds of links? This is an open question, but in view of this example we are inclined to think they should be a special case and have a length reduction factor. This however seriously complicates the underlying theoretical model.

Seconds The object `seconds` is the largest object in the design and has not unreasonable had the most changes. We have already discussed the datatype `present`, so we shall ignore it in discussing `seconds`. The majority of `seconds` consists of internal data and operations, so it clearly has a satisfactory structure. Moving the operation `tl.second` into `seconds` clearly reduces `seconds`' coupling, but does raise questions about the availability of external functions.

There is no obvious way to inform *Morpheus* the the operation `tl.second` is an externally required function. In this example, of course, this is not a serious problem because we can easily make `tl.second` external. The concern that functions may get 'lost' inside internal objects rather than been clearly visible.

Conclusion This discussion has highlighted some weaknesses in the application of Ψ and *Morpheus* to this small example. Some of these matters clearly require more research. Whilst some are partly due to this being a small example; because it is easy for humans to see better approaches in such small systems. As Müller et al. (1993) have observed, humans are good at identifying building blocks, given sufficient time, but such time is often not available; this is of course especially true in large complex systems.

We believe that *Morpheus* should be seen as a design aid, to make suggestions and hence raise issues, rather than as a replacement for human judgement and intuition.

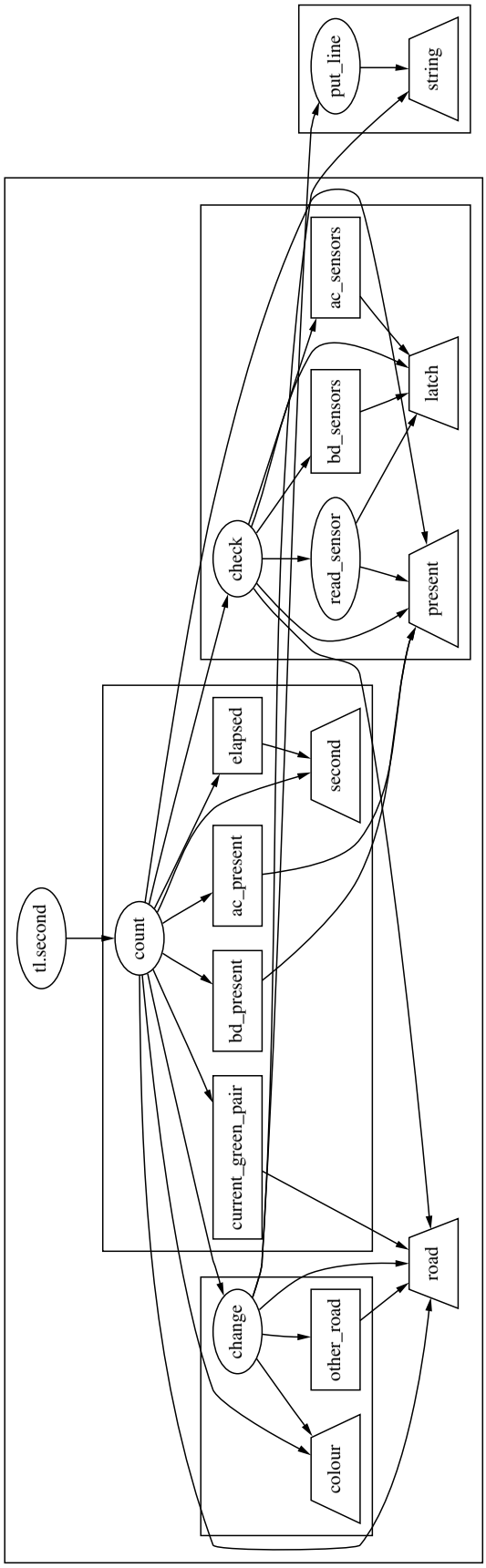


Figure 8.27: Graph of original Traffic Light design

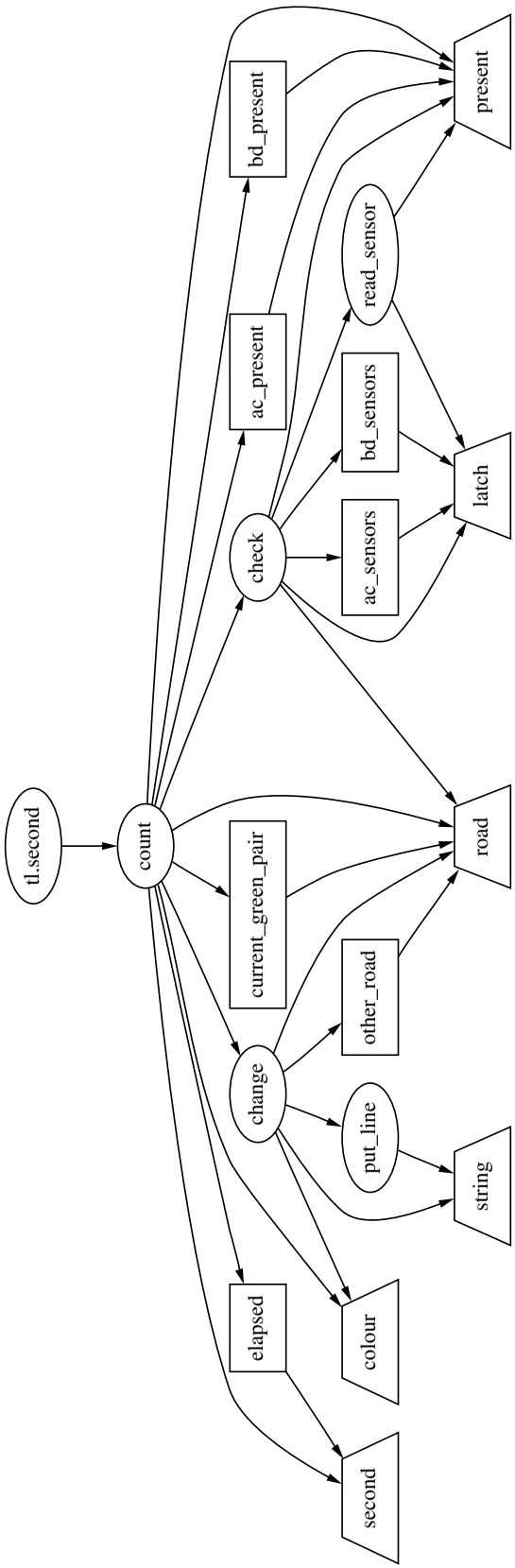


Figure 8.28: Graph of flat Traffic Light design without environment

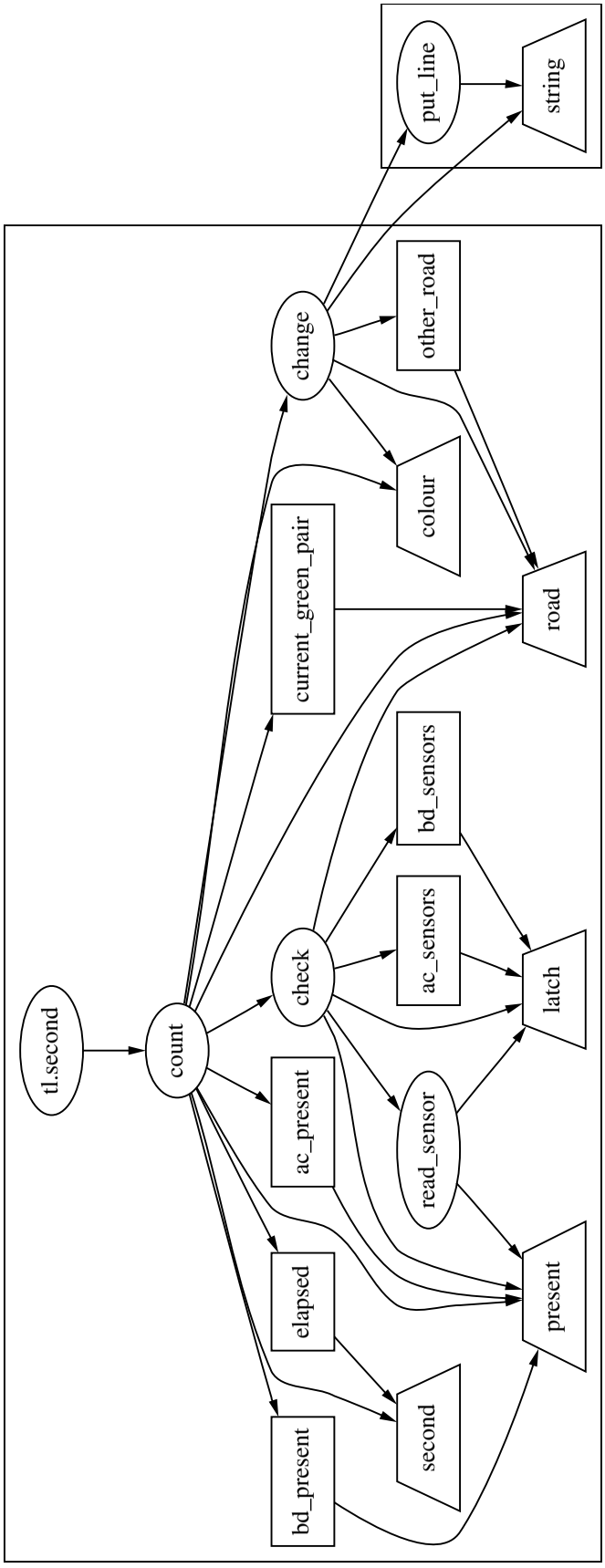


Figure 8.29: Graph of flat Traffic Light design with environment

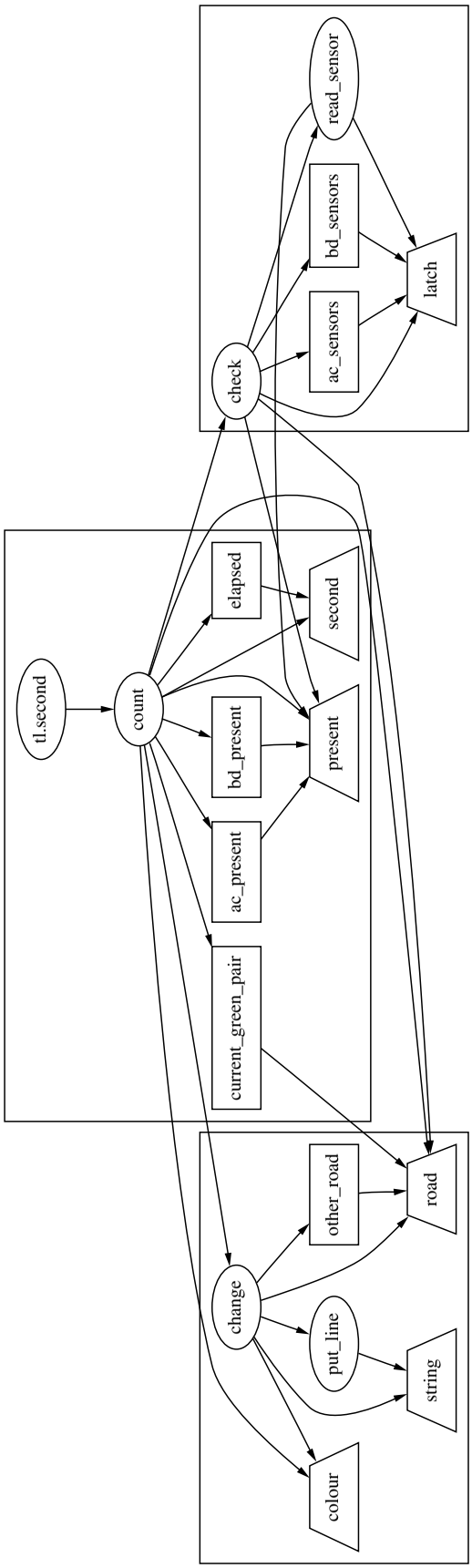


Figure 8.30: Graph of *Morpheus*'s Traffic Light design without environment

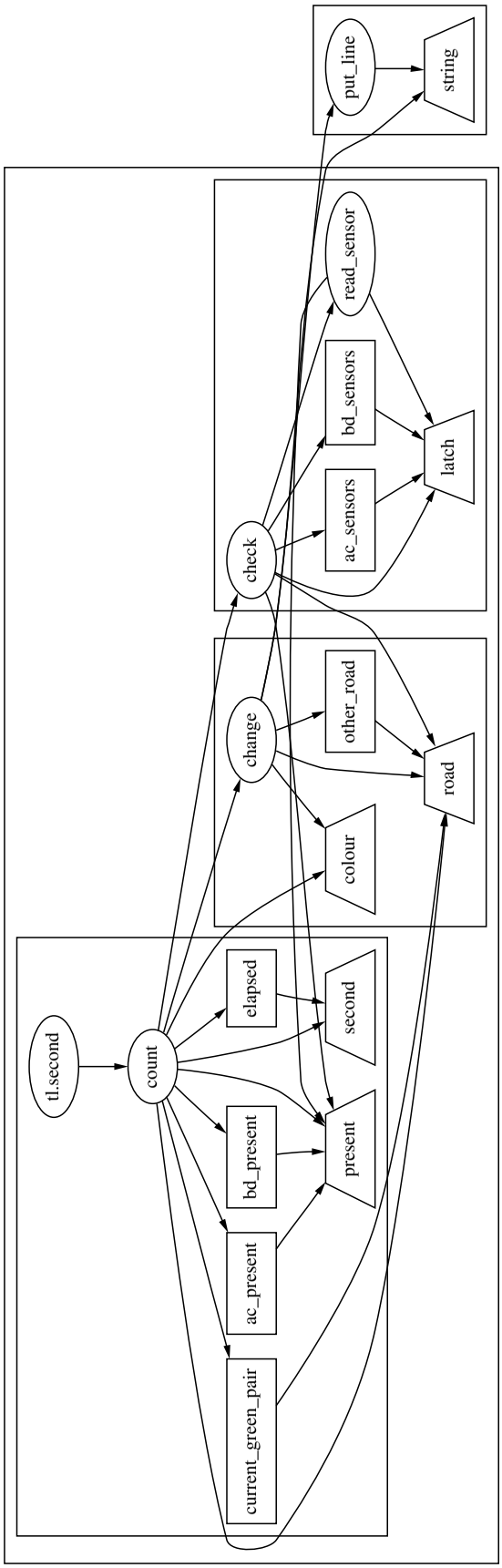


Figure 8.31: Graph of *Morpheus*'s Traffic Light design with environment

8.3 *TriviCalc* - A System to Design

Note 8.2. *Unfortunately, it is not possible to include design graph samples in this section because they are quite unreadable on any reasonable scale. Creating several disjoint sheets of a graph is unfortunately not likely to aid the reader's comprehension any better.*

This section describes the application of *Morpheus* to a moderately sized software design. As our sample project we have chosen a simple spreadsheet (called *TriviCalc*), which the authors designed and then processed through *Morpheus*. We examine *Morpheus*'s proposed changes and consider whether the design has been improved.

TriviCalc is a small spreadsheet, first described by Listov and Guttag (1986). A full description of *TriviCalc* is contained in Section B.1. It permits only the four basic arithmetic operations, and at most two dependencies into a third slot (e.g., we can say $A1 + A2 \rightarrow A5$, and indeed $A1 + A2 \rightarrow A3$ with $A1 + A3 \rightarrow A5$, *but not* $A1 + A2 + A3 \rightarrow A5$). Having created a dependency, *TriviCalc* behaves like a normal spreadsheet in that changes to $A1$ (say) are reflected into other slots (e.g., $A5$).

A small single line editor is provided for entering commands, along with a limited macro facility. The contents of a spreadsheet may be saved to a file and later restored, this restoration includes the current macro set.

Initial Design The guiding principal, was information hiding and the separation of concerns. It can be seen that each module maps onto objects in the description of the problem. With the exception of the editor, all the other main objects in the problem domain map to single objects in the design. The editor encapsulates several objects because it has a number of distinct sub-components of the display to maintain. It should be stressed that the original design was of reasonable quality, and did not contain any examples of strong coupling. The vast majority of the system only contained *data coupling*, however the Display Manager[2] does exhibit *external coupling* because it has knowledge of control sequences for display terminals.

Morpheus's Suggestions *Morpheus* was run on the authors' initial design for *TriviCalc*. Although *Morpheus* generates a log of its changes, describing these in detail would be counter-productive because the broad picture would be lost in the mass of detail. So instead we will just outline *Morpheus* main changes from a modular level.

Morpheus's changes can be summarised as follows:

- The most obvious set of changes was the removal of the separate data-types module, with the datatypes being redistributed around the other modules.

  This clearly has a significant impact on system coupling, because datatypes only used in one object can be localised. This is the ideal, but we must recognise that in a development situation, it is not always possible to predict when some datatype will be required by another *new* object. This may lead to unnecessary coupling between objects so that a single datatype may be shared. Such a situation, generally indicates that the object structure should be reviewed since there may be a need to introduce another object to handle the shared datatype (e.g., an ADT).

- The other significant change was to break the system into two high-level objects, one responsible for input of commands and command identification, the other responsible for obeying the commands and maintaining the state of the spreadsheet.

  This modification helps to reduce coupling and hence complexity by a clearer separation of concerns, i.e., by increasing cohesion. This change also suggests that in future some form of reusable editor and a reusable command despatch system might be worth developing, especially if *TriviCalc* was to form part of a family of spreadsheets with different levels of sophistication.

[2] Responsible for displaying the current state of the spreadsheet and the user's input.

- The other interesting change at the module level was the splitting of the spreadsheet calculation pad into two separate but equal objects, one responsible for validating potential commands, the other for actually performing the required changes.

  Much of the internal complexity of the calculation pad comes from the need to propagate changes to dependent slots and to save the spreadsheet in a suitable order for later restoration. By splitting the validation from the implementation, the mechanics for propagating changes can be further encapsulated.

8.3.1 Improvement?

Full details of both the initial and post-processing modular structures are shown in Section B.3. The original design had a complexity of 12038.9 bits, whilst *Morpheus*'s suggestion had a complexity of 9618.8 bits, a saving of about 20 percent.

Redistributing the datatypes is clearly a good thing, as the majority of types are only used locally inside a module. We choose to separate them partly for ease of construction and partly because initially we had no firm idea of where each type would be required. We suspect that the latter is a common problem when starting a design. This highlights the tendency to create a design and leave bits matters unchanged even when our ideas have solidified over the course of the design. This is, of course, a common problem with legacy software—bits get added and removed to solve the 'immediate problem' but without regard for the integrity of the overall design or future changes. We believe that following a major update to a system, the design should be reexamined, and a tool such as *Morpheus* would aid this process.

Rearranging the system into two high-level modules is at first sight odd. However on reflection such an arrangement serves to better separate concerns. One object can manage input and command identification, a relatively self-contained problem. Whilst the other object is responsible for carrying the entered commands without regard for how they arrived.

Breaking the spreadsheet's calculation pad into two modules can also be seen as a good idea, since validation of potential commands is not strictly part of obeying a set of commands. Obviously there is some overlap, since validation may depend on the current state of the calculation pad.

We conclude therefore that *Morpheus* has produced a simpler design.

8.3.2 Support for Future Changes?

In this section we want to examine *Morpheus*'s changes in the light of a possible future change. Currently *TriviCalc* does not support common (unary) mathematical functions,[3] for example logarithms and trigonometric functions. It would clearly extend *TriviCalc*'s usefulness if such a facility was added allowing a wider range of problems to be addressed.

Such a modification could easily be seen as a next step in enhancing *TriviCalc*'s capabilities. Obviously there would need to be individual operations in the calculation pad to support each required function and some form of validation mechanism to prevent illegal functions or operation on inappropriate data (for example taking the log of a string).

Basically, *TriviCalc* would require two groups of changes; a new set of operations in the calculation pad to support the additional functionality and modification of the command processor to despatch the new command as required.

There are several possible approaches to this problem - but from a future enhancement perspective the best is to support some form of generic unary function. Under this scheme the command identification sub-system would just recognise the 'basic shape' of a unary function and the calculation sub-system would be responsible for implementing the required function and propagating the changes as required. Such an approach would allow further unary functions to be added as

[3] In this section we use the term *function* to refer to a mathematical function and reserve *operation* for HOOD operations. There need not be a one-one mapping between a mathematical function and the equivalent implementation.

required. The editor would not require any changes and basic validation would remain unchanged since it can already check for valid slot identities and slot containing strings rather than arithmetical expressions.

The calculation pad would need a look-up table for validation and despatch to the appropriate operation. However, the basic entry point for updating the calculation pad would remain unchanged.

The biggest change to any object's interface would be a new operation in the calculation pad's validation object to support identification of valid unary functions. However, this would only be an example of *data coupling* and hence have no significant effect on the system's overall coupling.

These changes could be easily incorporated into the revised *TriviCalc* without causing any major perturbations. We therefore believe that the changes which *Morpheus* proposed for *TriviCalc* are well founded.

The strongest arguments prove nothing so long as the conclusions are not
verified by experience.
Experimental science is the queen of sciences
and the goal of all speculation.

ROGER BACON (1214?–94?)
English philosopher, scientist

Chapter 9

Summary and Conclusion

Synopsis

In this chapter, we present a critical evaluation of this thesis and its contribution to knowledge. Section 9.1 provides a brief summary of this thesis. Section 9.2 discusses what this thesis has achieved, and how far its objectives have been met. Section 9.3 looks forward to future work, as a result of this research. Finally, Section 9.4 provides a brief overall conclusion.

9.1 Summary of this Thesis

This thesis has studied the problem of providing an intelligent system to aid software designers improve the quality of their designs. We have limited ourselves to the objective evaluation of a design's modular complexity. Little previous work has been done in producing systems for even this limited objective.

Although there are clearly many other factors which influence quality; many of which are subjective and involve value judgements. Further, some of these factors are subconscious and not fully articulated or understood by human experts. We believe that reducing software complexity makes a significant contribution to improving software quality, by reducing the amount of complexity that engineers have to handle. Hence their time and effort can be spent on other areas of software development.

We have rigorously defined modular complexity by developing a method to represent the structure of a design's graph as a message. By reference to Kolmogorov complexity and in particular the Minimum Description Length principle, the length of such a message is an objective measure of the complexity of a proposed design.

We have examined some of the theoretical properties of our proposed complexity measure against criteria proposed by other researchers, in particular Weyuker (1988). On this basis we have shown that our measure satisfies these criteria and deserves further study. A key feature of the design of our complexity measure is that it provides an approach for finding alternative designs with reduced complexity.

We have created a prototype tool, *Morpheus*, which reads in a design expressed in HOOD, and searches for a better design. *Morpheus* uses a narrow beam hill-climbing search strategy to look for less complex designs using our complexity measure for evaluating potential designs. We have applied *Morpheus* to a design which we created for a spreadsheet package. *Morpheus* successfully found a less complex architecture.

9.2 Evaluation

The previous section provided a brief summary of this thesis. In this section we shall look at how well this work meets our original objective.

9.2.1 Achievements

There can be no doubt that we have created a prototype tool which takes in an architectural design expressed in HOOD, and finds alternative designs with less complex structure. Complexity has been defined in terms of the length of a message describing the structure of the design. The use of message length as a measure of complexity is founded on Kolmogorov complexity, which gives us an objective basis for comparing the structural complexity of designs.

We have shown that our complexity metric satisfies criteria that other researchers have suggested are good properties for complexity measures. This has been rigorously proved for the most general hierarchical modular structure consisting of inter-connected basic entities and objects (modules). The mathematical proofs make no assumptions about the nature of basic entities, and only assume that modules can encapsulate basic entities and other modules.

The most significant achievement of this work is the development of an objective complexity metric, which permits the evaluation of structural complexity. This claim to objectivity is not based on other people's notions of complexity, which others may wish to argue about; but is based on the Minimum Length Description principle related to Kolmogorov complexity, and is embodies Occam's razor. Occam's razor basically says that given the choice between two explanations, of the same situation, the simplest should be preferred. This is exactly what our tool does for software designs.

We have met the following objectives

- Development of a better complexity measure, which permits the complexity of alternative designs to be objectively compared. The measure is based on well-established mathematical principles.
- Highlighted weaknesses in the the HOOD design notation, and proposed some small extensions to HOOD which correct some of these deficiencies. The designer must have already considered the information missing from the design, so we are only tightening the description of the design and not adding substantially to the designer's workload.
- Identification of a method for finding alternative architectural designs. Given that our complexity measure is a message describing a graph, it is easy to see that manipulating the graph will change graph's complexity and hence the message length.
- Identification of a basic set of rules for manipulating design graphs.
- Production of an automatic system for finding simpler designs (if possible) to a proposed design.

Potentially this may assist in

- Developing simpler designs.
- Identification of more defects in the design stage, which should reduce the cost of software production.
- Reducing the costs of maintenance both through the development of more reliable software with less bugs and conceptually easier designs to understand whilst making modifications.
- Simplifying software maintenance.

- A better understanding of the relationship between coupling and cohesion. Particularly in situations were the worst forms of coupling are not permitted.

As part of the development of *Morpheus*, we have identified a number of deficiencies in HOOD as currently defined. Some of these deficiencies we have addressed in our extensions to HOOD. The others (e.g., nested operations) would be quite simple to add, but require further consideration because they alter what may be regarded as the philosophy of HOOD.

Although *Morpheus* is based on HOOD, in principle there is no reason why (with suitable changes) it could not be applied to other modular based architecture design languages, for example VDM-SL.

9.2.2 Industrial Application

Morpheus has the potential to be used commercially, but before this happens some remaining problems need to be addressed, namely:

- The resource requirements of *Morpheus* are currently excessive, and would prohibit the system being used on a regular basis.
- The global structure of the rule base in *Morpheus* needs to be improved to make it easier to modify the style rules to cater for different methodologies and handling of proposed design changes for stylistic reasons.
- Industrial validation of *Morpheus* is required to boost confidence in its suggestions.

9.3 Further Work

The previous section considered the limitations with the present system. In this section, we shall consider possible future work

- The modifications to the implementation of *Morpheus* referred to in Section 7.3 need to be incorporated. This should significantly improve *Morpheus*'s run-time performance both in terms of speed and memory utilisation.
- The introduction of a type theory and object-oriented design need to be investigated. Currently, this work is based upon HOOD's typing system (name equivalence) rather than decomposing a type into its constitution parts (if any). Further the underlying theoretical model used in this thesis is object-based and hence has no support for a class concept. The introduction of classes would broaden the application of this approach.
- A faster method for pruning *Morpheus*'s search space needs to be developed, probably by refining our understanding of the relationship between coupling and cohesion. Although we believe that *Morpheus*'s speed can be improved by use of an incremental model, some mechanism[1] for discarding 'unsatisfactory' changes to the design is still desirable.
- The effect of different coding schemes for natural numbers needs to be investigated.[2] There are a number of asymptotically optimal universal codes for integers (see Section 5.2.3). More interestingly would be the investigation of a coding scheme which gave less weight to small integers and had a smaller first derivate.
- Different ways of generating alternative designs should be investigated, possibilities include genetic algorithms and simulated annealing. This thesis has only considered creating new designs by manipulating the graph representation of a design. Clearly, there are alternative approaches to creating 'new' designs, and these should be investigated.

[1] It is not sufficient to drop changes which increase message length.

[2] Our mathematical model only requires a monotonically increasing function.

- Alternative theories for modelling a hierarchical graph need to be investigated, and their impact on message lengths determined. This thesis assumes a single class of theories for describing a hierarchical graph. There are undoubtedly others, some of which may yield smaller message lengths and thus more closely approximate the true[3] Kolmogorov Complexity of the underlying design.

- Providing a clearer method for reporting *Morpheus*'s results, in a manner readily understandable to the end-user. *Morpheus*'s output is currently rather cryptic, and not obviously related to the initial design; particularly as module names are not preserved. To make *Morpheus* acceptable in an industrial setting, a simple to understand output is required. Even better would be to reverse engineer *Morpheus*'s output into a design notation; ideally using the same notation as the original design.

- Integrating *Morpheus* into other CASE tools. *Morpheus* is really intended as the back-end of a CASE tool, and not for direct use by a designer. We need to merge *Morpheus* into a CASE tool so that it has access to other facilities (in particular a database for storing large designs) and supports industrial use.

9.4 Contribution of This Thesis

The research reported in this thesis has developed a metric for measuring the absolute complexity of a software design's architecture. Complexity is measured in an objective manner and does not require any human judgement. This complexity measure has been theoretically validated against Weyuker's (1988) properties for a complexity measure. Using this metric a tool has been developed which takes an existing design and finds a simpler alternative. Initial empirical validation shows potential.

Research is to see what everybody else has seen,
and to think what nobody else has thought.

ALBERT SZENT-GYÖRGI (1893–1986)
U.S. biochemist

Hofstadter's Law: It always takes longer than you expect,
even when you take into account Hofstadter's Law.

DOUGLAS R. HOFSTADTER (1945–)
Gödel, Escher, Bach (1979)

[3]Remember that Kolmogorov Complexity is non-computable.

Bibliography

Aceto, L. (1992). *TriviCalc Reference Manual*. COGS, University of Sussex, Brighton, England.

Anderson, J., editor (1989). *POP-11 Comes of Age*. Ellis Horwood, Chicester, England.

ANSI C (1989). *The American National Standard for Information Systems - Programming Language C*. American National Standards Institute, New York, NY. ANSI X3.159-1989.

Atkinsom, G. (1977). The non-desirability of structured programming in user languages. *ACM Sigplan Notices*, 12(7):43–50.

Bancroft (1969). *English Dictionary*. Bancroft and Co. (publishers) Ltd., London, England, revised edition.

Barrett, R., Ramsey, A., and Sloman, A. (1985). *POP-11: A Practical Language for Artifical Intelligence*. Ellis Horwood, Chicester, England.

Baxter, R. A. (1996). *Minimum Message Length Inference: Theory and Applications*. PhD thesis, School of Computer Science, Monash University.

Benedusi, P., Cimitile, A., and de Carlini, U. (1992). Reverse engineering processes, design documents production, and structure charts. *Journal of Systems and Software*, 19(3):225–245.

Bieman, J. M. and Ott, L. M. (1994). Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657.

Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E. (1994). Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83.

Binder, R. V. (1996). Testing object-oriented systems: A status report. First published in *American Programmer* April 1994.

Birtwistle, G. M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1973). *SIMULA Begin*. Petrocelli/Charter, New York, NY.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ.

Booch, G. (1987). *Software Engineering with Ada*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 2nd edition.

Booch, G. (1991). *Object Oriented Design with Applications*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA.

Bornat, R. (1979). *Understanding and Writing Compilers: A do-it-yourself guide*. Macmillan Computer Science Series. Macmilian Education Ltd., Basingstoke, England.

Briand, L. C., Morasca, S., and Basili, V. R. (1996). Property-based software engineering measurement: Refining the additivity properties. *IEEE Transactions on Software Engineering*, 22(1):68–86.

Bujnowski, J. (1993). Knowledge elicitation report. Technical report, British Telecommunications Research Laboratory, Martlesham Heath, Ipswich, Suffolk. draft.

Bundy, A. and MacQueen, H. (1994). The new software copyright law. *The Computer Journal*, 37(2):79–82.

Calliss, F. W. (1989). *Inter-Module Code Analysis Techniques for Software Maintenance*. PhD thesis, School of Engineering and Applied Science (Computer Science), University of Durham.

Calliss, F. W. and Cornelius, B. J. (1989). A case study of module factoring. Computer Science Technical Report 13/1989, School of Engineering and Applied Science (Computer Science), University of Durham, England.

Carré, B. (1979). *Graphs and Networks*. Oxford applied mathematics and computing science series. Clarendon Press, Oxford, England.

Casey, C. (1994). *A Programming Approach to Formal Methods*. McGraw-Hill international series in software engineering. McGraw-Hill Book Company, London, England.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

Choi, S. C. and Scacchi, W. (1990). Extracting and restructing the design of large systems. *IEEE Software*, 7(1):66–71.

Churcher, N. I. and Shepperd, M. J. (1995). Towards a conceptual framework for OO software metrics. *ACM Sigsoft Notices*, 20(2):69–75.

Cox, B. J. (1986). *Object-oriented Programming - An evolutionary Approach*. Addison-Wesley Publishing Company, Reading, MA.

Curtis, B., Krasner, H., and Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1288.

Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured Programming*. Academic Press, New York, NY.

Dasgupta, S. (1991). *Design Theory and Computer Science: Processes and Methodology of Computer Science Design*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England.

Dawes, J. (1991). *The VDM-SL Reference Guide*. Pitman Publishing, London, England.

de Marco, T. (1979). *Structured Analysis and System Specification*. Prentice-Hall, Englewood Cliffs, NJ.

de Remer, F. and Kron, H. H. (1976). Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):80–86.

Deitel, H. M. (1984). *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, Reading, MA, revised first edition.

Delatte, B., Heitz, M., and Muller, J. F. (1993). *HOOD Reference Manual 3.1*. Prentice-Hall, London, England.

Denvir, T. (1986). *Introduction to Discrete Mathematics for Software Engineers*. Macmillan Computer Science Series. Macmilian Education Ltd., Basingstoke, England.

Dijkstra, E. W. (1968). Go to statements considered harmful. *Communications of the ACM*, 11(3):147–148. (letter to the editor).

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10):TBD. (Turing Award Lecture).

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.

Embley, D. W. and Woodfield, S. N. (1987). Cohesion and coupling for abstract data types. In *Sixth Annual International Phoenix Conference on Computer Communications*, pages 229–234, Washington, D.C. IEEE Computer Society.

Embley, D. W. and Woodfield, S. N. (1988). Assessing the quality of abstract data types written in Ada. In *Tenth International Conference on Software Engineering*, pages 144–153, Washington, D.C. IEEE Computer Society.

Englemore, R. and Morgan, T. (1988). *Blackboard Systems*. Addison-Wesley Publishing Company, New York, NY.

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

Fenton, N. E. (1994). Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206.

Fetzer, J. H. (1988). Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1064.

Fowler, M. and Scott, K. (1997). *UML Distilled: Applying the Standard Object Modeling Language*. The Addison-Wesley Object Technology Series. Addison Wesley Longman, Inc., Reading, MA.

France, R. B. (1992). Semantically extended data flow diagrams: A formal specification tool. *IEEE Transactions on Software Engineering*, 18(4):329–346.

Fuggetta, A., Ghezzi, C., Mandrioli, D., and Morzenti, A. (1993). Executable specifications with dataflow diagrams. *Software Practice and Experience*, 23(6):629–653.

Galton, A. (1992). On the notions of specification and implementation. Research Report 246, Department of Computer Science, University of Exeter, England.

Gane, C. and Sarson, T. (1979). *Structured Systems Analysis*. Prentice-Hall, Englewood Cliffs, NJ.

George, C., Haff, P., Havelund, K., Haxthausen, A. E., Milne, R., Nielsen, C. B., Prehn, S., and Wagner, K. R. (1992). *The RAISE Specification Language*. Prentice-Hall, Hemel Hempstead, England.

George, C., Haxthausen, A. E., Huges, S., Milne, R., Prehn, S., and Pedersen, J. S. (1995). *The RAISE Development Method.* Prentice-Hall, Hemel Hempstead, England.

Goguen, J. A., Jouannaud, J.-P., and Meseguer, J. (1985). Operational semantics of order-sorted algebra. In Brauer, W., editor, *12th International Conference on Automata, Languages and Programming*, pages 221–231. Springer-Verlag, Berlin, Germany. LNCS:194.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its implementation.* Addison-Wesley Publishing Company, Reading, MA.

Grogono, P. (1980). *Programming in Pascal.* Addison-Wesley Publishing Company, Reading, MA, revised edition.

Halstead, M. H. (1977). *Elements of Software Science*. Elsevier North-Holland, New York, NY.

Hardy, G. H. (1947). *A Mathematican's Apology*. Cambridge University Press.

Harrison, W. (1992). An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering*, 18(11):1025–1029.

Harrison, W. and Ossher, H. (1993). Subject-oriented programming (a critique of pure objects). In *OOPSLA '93: Eightth annual conference on Object-Oriented Programming Systems, Languages and Applications*, pages 411–428, Washington, DC. ACM Press. published in ACM SIGPLAN Notices 28(10).

Henry, S. and Kafura, D. (1993). The evaluation of software systems' structure using quantitative software metrics. In Shepperd, M. J., editor, *Software Engineering Metrics, Volume 1: Measures and Validations*, McGraw-Hill international series in software engineering, chapter 6, pages 99–111. McGraw-Hill Book Company, Maidenhead, England. Reprinted from Software Practice and Experience, 14(6); 561–573, 1984.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

HOOD HRM (1995). *HOOD Reference Manual, Release 4*. HOOD Technical Group. HRM4–6/29/95.

HOOD HUM (1996). *HOOD User Manual*. HOOD User Groop. HUM-1.0.

Hops, J. M. and Sherif, J. S. (1995). Development and application of composite complexity models and a relative complexity metric in a software maintenance environment. *Journal of Systems and Software*, 31(2):157–169.

Howell, A. J. (1996). personal communications.

Hutchens, D. H. and Basili, V. R. (1993). System structure analysis: Clustering with data bindings. In Shepperd, M. J., editor, *Software Engineering Metrics, Volume 1: Measures and Validations*, McGraw-Hill international series in software engineering, chapter 5, pages 83–98. McGraw-Hill Book Company, Maidenhead, England. Reprinted from IEEE Transactions on Software Engineering, 11(8); 749–757, 1985.

Ichbiah, J. D. et al. (1983). *Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A*. Castle House Publications Ltd.

Jackson, M. A. (1975). *Principles of Program Design*. Academic Press, Orlando, FL.

Jackson, M. A. (1983). *System Development*. Prentice-Hall, Hemel Hempstead, England.

Jackson, M. A. (1995). *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press series. Addison-Wesley Publishing Company, Wokingham, England.

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1994). *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press series. Addison-Wesley Publishing Company, Wokingham, England. Revised printing.

Jones, C. B. (1986). *Systematic Software Development using VDM*. Prentice-Hall, Englewood Cliffs, NJ.

Jones, D. S. (1979). *Elementary Information Theory*. Oxford applied mathematics and computing science series. Clarendon Press, Oxford, England.

Khoshgoftaar, T. M. and Allen, E. B. (1994). Applications of information theory to software engineering measurement. *Software Quality Journal*, 3(2):79–103.

Koutsofios, E. and North, S. C. (1994). *Editing graphs with* dotty. AT&T Bell Laboratories, Murray Hill, NJ.

Laventhol, J. (1987). *Programming in POP-11*. Blackwell Scientific Publications Ltd.

Lew, K. S., Dillon, T. S., and Forward, K. E. (1988). Software complexity and its impact on software reliability. *IEEE Transactions on Software Engineering*, 14(11):1645–1655.

Li, M. and Vitányi, P. (1993). *An Introduction to Kolmogorov Complexity and its Applications*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY.

Li, M. and Vitányi, P. (1997). *An Introduction to Kolmogorov Complexity and its Applications*. Graduate texts in Computer Science. Springer-Verlag, New York, NY, 2nd edition.

Listov, B. and Guttag, J. (1986). *Abstraction and Specification in Program Development*, pages 433–444. The MIT Press, Cambridge, MA.

Lor, K. W. E. and Berry, D. M. (1991). Automatic synthesis of SARA design models from system requirements. *IEEE Transactions on Software Engineering*, 17(12):1229–1240.

MacKay, D. J. C. (1997). Information theory, pattern recognition and neural networks. forthcoming, available in http://wol.ra.phy.cam.ac.uk/mackay/itprnn.

Marca, D. A. and McGowan, C. L. (1988). *SADT—Structured Analysis and Design Technique*. McGraw-Hill.

MASCOT (1987). *The Official Handbook of MASCOT: Version 3.1*. Her Majesty's Stationery Office, London, England. Joint IECCA and MUF Committee on MASCOT.

McCabe, T. J. (1976). A software complexity measure. *IEEE Transactions on Software Engineering*, 2(6):308–320.

McCabe, T. J. and Butler, C. W. (1989). Design complexity measurements and testing. *Communications of the ACM*, 32(12):1415–1425.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Hemel Hempstead, England.

Mohanty, S. N. (1981). Entrophy metrics for software design evaluation. *Journal of Systems and Software*, 2:39–46.

Müller, H. A., Orgun, M. A., Yilley, S. R., and Uhl, J. S. (1993). A reverse-engineering approach to subsystem structure identification. *Software Maintenance Research and Practice*, 5(4):181–204.

Munro, J. E. (1992). *Discrete Mathematics for Computing*. Chapman & Hall, London, England.

Mynatt, B. T. (1990). *Software Engineering with Student Project Guidance*. Prentice-Hall.

Nakajo, T. and Kume, H. (1991). A case history analysis of software error cause-effect relationships. *IEEE Transactions on Software Engineering*, 17(8):830–838.

Neil, M. and Bache, R. (1993). Data linkage maps. *Software Maintenance: Research and Practice*, 5:155–164.

Oliver, J. J. and Hand, D. J. (1994). Introduction to minimum encoding inference. Technical Report 4–94, Department of Statistics, Open University, England. revised Dec. 1996.

Orr, K. T. (1971). *Structured System Development*. Yourdon Press, New York, NY.

Page-Jones, M. (1988). *The Practical Guide to Structured Systems Design*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition.

Page-Jones, M. (1992). Comparing techniques by means of encapsulation and connascence. *Communications of the ACM*, 35(9):147–151.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Parnas, D. L. (1997). Software engineering: An unconsummated marriage. *Communications of the ACM*, 40(9):128.

Popper, K. R. (1968). *The Logic of Scientific Discovery*. Harper and Row, New York, NY.

Pressman, R. S. (1992). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Inc., New York, NY, 3rd edition.

Rayward-Smith, V. J. (1983). *A First Course in Formal Language Theory*. Blackwell Scientific Publications, Oxford, England.

Reisig, W. (1985). *Petri nets: An Introduction*. Springer-Verlag.

Rich, C. and Waters, R. C. (1990). *The Programmer's Apprentice*. ACM Press frontier series. Addison-Wesley Publishing Company.

Rissanen, J. (1978). Modelling by shortest data description. *Automatica*, 14:465–471.

Robinson, P. J. (1992a). *HOOD: Hierarchical Object-Oriented Design*. Prentice-Hall object-oriented series. Prentice-Hall, Hemel Hempstead, England.

Robinson, P. J., editor (1992b). *Object-oriented Design*. UNICOM Applied Information Technology. Chapman & Hall, London, England.

Room, A. (1990). *NTC's Classical Dictionary: The Origins of the Names of Characters in Classical Mythology*. National Textbook Company, Lincolnwood, IL.

Rosen, J.-P. (1997). *HOOD: An Industrial Approach for Software Design*. HOOD Technical Group, Spacebel, Belgium.

Ross, K. H. and Moore, R. R. (1995). *XY-pic Reference Manual*. DIKU, University of Copenhagen, Denmark. Version 3.1.

Rotenstreich, S. (1994). Toward measuring potential coupling. *Software Engineering Journal*, 9(2):83–90.

Schreiner, A. T. and Friedman, Jr, H. G. (1985). *Introduction to Compiler Construction with UNIX*. Prentice-Hall, Englewood Cliffs, NJ.

Shannon, C. E. (1948). The mathematical theory of communications. *Bell System Technical Journal*, 27:379–423,623–656.

Shepperd, M. J., editor (1993). *Software Engineering Metrics, Volume 1: Measures and Validations*. McGraw-Hill international series in software engineering. McGraw-Hill Book Company, Maidenhead, England.

Shepperd, M. J. and Ince, D. C. (1993). *Derivation and Validation of Software Metrics*. The International Series of Monographs on Computer Science. Clarendon Press, Oxford, England.

Simon, H. A. (1973). The structure of ill structured problems. *Artifical Intelligence*, 4:181–200.

Simon, H. A. (1976). *Administrative Behaviour*. The Free Press, New York, NY, 3rd edition.

Simon, H. A. (1981). *The Sciences of the Artifical*. The MIT Press, Cambridge, MA, 2nd edition.

Spivey, J. M. (1989). *The Z Notation: A Reference Manual*. Prentice-Hall, Hemel Hempstead, England.

Stroustrup, B. (1994). *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 2nd edition.

Thornton, C. J. and du Boulay, B. (1992). *Artificial Intelligence Through Search*. Intellect, Oxford, England.

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society (Series 2)*, 42:230–265.

UML (1997). Unified modeling language, version 1. http://www.rational.com/uml.

Visser, W. and Hoc, J.-M. (1990). Expert software design strategies. In Hoc, J.-M., Green, T. R. G., Samurçay, R., and Gilmore, D. J., editors, *Psychology of Programming*, Computers and People series, chapter 3.3, pages 233–249. Academic Press Ltd., London, England.

Ward, P. T. and Mellor, S. J. (1985). *Structured Development for Real-Time Systems*. Yourdon Press, New York, NY. 3 volumes.

Watson, D. (1989). *High-Level Languages and Their Compilers*. Addison-Wesley Publishing Company.

Weinberg, G. M., Geller, D. P., and Plum, T. W. (1975). If-then-else considered harmful. *ACM Sigplan Notices*, 10(8):34–43.

Weyuker, E. J. (1988). Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227.

Wirth, N. (1974). On the construction of well structured programs. *Computing Surveys*, 6(4):247–259.

Witten, I. H., Neal, R. M., and Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Wulf, W. A., Shaw, M., Hilfinger, P. N., and Flon, L. (1981). *Fundamental Structures of Computer Science*. Addison-Wesley Publishing Company, Reading, MA.

Yourdon, E. N. (1986). *Structured Walkthroughs*. Prentice-Hall, 3rd edition.

Yourdon, E. N. (1989). *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, NJ.

Yourdon, E. N. and Constantine, L. L. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Englewood Cliffs, NJ.

Appendix A

Augmented HOOD

This appendix presents the changes to HOOD's Standard Interchange Format, as documented in Delatte et al. (1993, appendix D). The syntax is presented using BNF notation as described in Delatte et al. (1993). Meta-comments are delimited by '/*' and '*/'. Numbers in round brackets refer to the syntax phrases defined in Delatte et al..

A.1 Changes to Existing Syntax

A.1.1 Pseudo Code

pseudo_code_section ::= /* (29) */
    PSEUDO_CODE
        [code_linkage_section]
        [free_text]
| **PSEUDO_CODE**
        NONE

A.2 Pseudo Code Enhancements

code_linkage_section ::=
    OPERATION_REQUIREMENTS
        [requires_type_section]
        [reads_from_section]
        [writes_to_section]
    END_OPERATION_REQUIREMENTS
| **OPERATION_REQUIREMENTS**
        NONE

requires_type_section ::=
    REQUIRES_TYPE
        *type*_reference semi_colon
        {*type*_reference semi_colon}
| **REQUIRES_TYPE**
        NONE

reads_from_section ::=
    READS_FROM

*variable*_reference semi_colon
{*variable*_reference semi_colon}
| **READS_FROM**
NONE

writes_to_section ::=
WRITES_TO
*variable*_reference semi_colon
{*variable*_reference semi_colon}
| **WRITES_TO**
NONE

A.3 Semantics of Augmented HOOD

The previous section detailed the new syntax for Augmented HOOD. In this section, we shall explain the meanings of these new syntactic phrases.

The **code_linkage** section provides the opportunity to list additional information about the design of an operation. The **requires_type** phrase contains a list of the types used and thus required in the current operation. It is not intended that the types listed in the operation's signature should be repeated, but of course it does no harm. Nor is it necessary to list the types associated with other operations listed in the **used_operations** phrase. Its intended use is to make explicit the use of types which cannot be inferred from other parts of the ODS. Such usage will probably only be needed in 'large' operations.

The two phrases **reads_from** and **writes_to** both provide the facility to list which data items the current operation reads from and writes to respectively. It is assumed that an entry in the **writes_to** phrase implies a corresponding entry in the **reads_from** phrase. This change will only impact on operations which use data in the encapsulating object.

Additionally, one minor semantic extension is made to the **used_operations** phrase. Constants can and should be listed here, just as operations are in standard HOOD. This provides more information on the facilities required by an operation.

Appendix B

TriviCalc - An Example

B.1 *TriviCalc* Reference Manual

Note B.1. *The TriviCalc Reference Manual is taken verbatim from Aceto (1992). Aceto et al. derived it, with only minor changes, from Listov and Guttag (1986).*

TriviCalc is a program that can be used as a scratchpad for problems involving arithmetic. The user of the program enters numbers or text into the *storage area* of the computer's memory. The data are then displayed on the terminal's screen.[1] The user can combine values that are already displayed on the screen to obtain new values, which are stored in the computer and displayed elsewhere on the screen. When this is done, the program remembers the relationships between the numbers, so that the calculation can be repeated on different values.

As an example, imagine two numbers displayed at places on the screen labelled $A1$ and $B1$. The user types a command that causes *TriviCalc* to add the value stored at $A1$ to that stored at $B1$ and store the result at $C1$. If the user later changes the value at $A1$, the value at $C1$ will also change so that it remains the sum of the values at $A1$ and $B1$.

User Interface

During the operation of *TriviCalc*, the display is divided into three parts. Figure B.1 contains a diagram of the display when the program is first begun.[2]

The first line of the display, known as the *status line*, is used to display descriptions of the execution state of the program. The line is divided in two: the left portion contains error messages,[3] while the right portion displays information about the contents of the current slot.[4]

The second line of the display is used as a *working area.* When the user types textual or numerical input, it is displayed here, pending action by some user-specified command. The text displayed in this area may be edited using a simple editor, described in detail in a later section.

The remainder of the screen is occupied by what is known as the *storage area*. It is used to display textual representations of all the values stored in *TriviCalc*'s internal storage.

The storage area is divided into fields or *slots*, which are eight characters wide and one line high. Each slot is named by a number describing its row and a letter describing its column, starting

[1] This is (part of) a data requirement. Unfortunately, nowhere is it made clear what capabilities the terminal is to have. From the specified storage area indices and the given size of a slot, we can deduce that a 24×80 screen is being assumed. That should be separated out and specified clearly.

[2] Here again we are left to deduce a lot of information about the initial state of *TriviCalc*, e.g., all slots are initially blank.

[3] This is an example of incompleteness. What exactly are the error messages? How are they to be displayed?

[4] Here is an example of incompleteness: in what proportion is the line divided, and what does "information" mean?

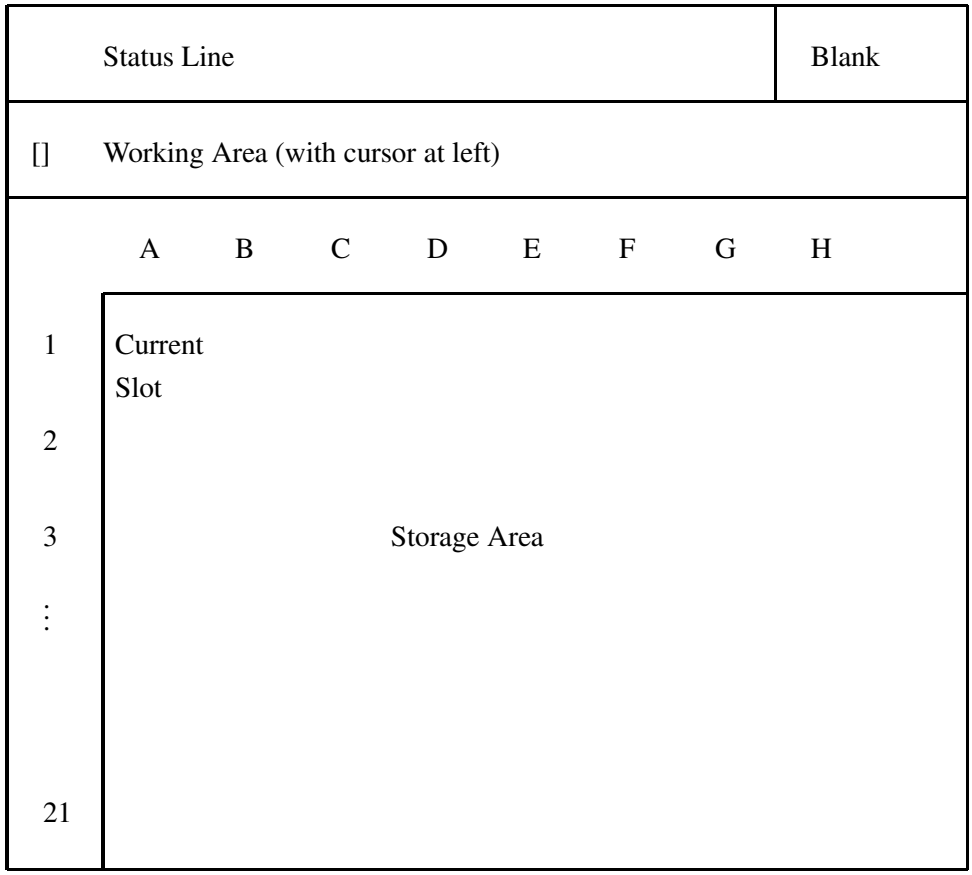


Figure B.1: Diagram of initial display

at the upper left-hand corner of the storage area. Thus the slot in this corner is named $A1$, and the slot diagonally opposite it, in the lower right-hand corner of the storage area, is named $H21$. Surrounding the storage area to the left and top is a border containing the name of each row and column.

There is always one slot in the storage area that is considered to be the *current slot*. This slot is displayed specially, perhaps in inverse video, so as to be easily recognisable by the user. The initial current slot is $A1$. To make it easier to distinguish the different areas, the current slot contents,[5] the working area and the header lines for the storage area should also be displayed specially.[6] Reverse video is not shown in Figure B.1.

Description of Operation

TriviCalc's storage consists of a two-dimensional array whose elements are referred to by a letter in the range A to H describing one dimension and a number in the range 1 to 21 describing the other.[7] A textual approximation[8] to the contents of each storage element is displayed continuously in the storage area of the terminal display.

[5]Here is an ambiguity: the current slot's contents appear at *two* locations on the screen, once in the status area and once in the storage area. Are both to be displayed specially?

[6]How? What about the error messages? We might expect that they are also to be displayed specially, at least if they are to catch the user's eye.

[7]So, there are eight slots per line. Eight slots times eight characters takes sixty-four characters per line, so there is probably intended to be space between the slots to keep them apart. Should it be at the front or at the back? This is an example of incompleteness, since there are certainly many unsatisfactory implementations of what was asked for.

[8]It is never made clear what is meant by "textual approximation".

Elements of the storage area may be of three kinds: *blank*, *value* or *comment*.

- A blank line has no value.
- A comment is a string of up to eight characters.[9]
- When a comment occupies an element of storage, it has no effect on any other element of storage.

A value is a floating point number.[10] Values in storage may be related to each other by multiplication, division, addition and subtraction. These relations are set by the user and may be changed at any time. Some values in storage will be entered by the user as constants or parameters; others will be derived as a result of one of the relations mentioned above. A derived value is said to *depend* on the other values.

There are some general rules that apply to changing the contents of the storage area of *TriviCalc*. *TriviCalc* maintains the *consistency* of the storage area. Given the set of commands discussed above, it is possible to set up a configuration of values in storage that has no consistent interpretation. The *TriviCalc* command interpreter will not allow such a situation to occur. Instead, when a command would cause the storage area to become inconsistent, the command is not carried out and an error message is displayed.[11]

The rules that the *TriviCalc* command interpreter follows to ensure the consistency of the storage area are:

1. Only elements of storage whose contents are values may be depended on by other elements.
2. When an element's contents are depended upon by other elements, that element's type may not be changed.[12]
3. When the contents of an element depend on those of other elements, changing the dependent element's contents causes the dependency to be broken.[13]
4. A given element may not depend, even indirectly, on its own value.

The *TriviCalc* Command Processor

Except for the movement commands described below, all commands and their arguments are entered through the working-area editor. When a command is executed, the entire working-area is submitted to the *TriviCalc* command processor, which either performs the command at once or rejects the command as an error. The *TriviCalc* command processor never modifies the contents of the working area of the display.

A full description of the operation of each command supported by the *TriviCalc* command processor follows. The notation 'CONTROL *character*' is used to denote holding down the CONTROL key and pressing the key labelled *character*.

`SAVE;FILE:`***name*****;** The current state of the storage area is saved in a file named *name.tc*, where *name* is the file argument.[14] The state is stored as an ASCII string containing a sequence of *TriviCalc* commands separated by line-feed characters. These commands should have the

[9] Presumably not all characters are allowed, e.g., control characters.

[10] The customer may have in mind some requirements on the range and precision of manipulation of floating point numbers. If there are none, then this should be stated.

[11] The exact text of the error message should be given in each case, and agreed with the customer.

[12] Another ambiguity: what is a type?

[13] Example of specification which is difficult to read!

[14] The allowable forms of *name* are going to be executive dependent. This should be stated as a data requirement since it details a feature of the environment in which *TriviCalc* is to run.

property that if they were executed in sequence, beginning with a blank storage area, they would generate the storage area in effect at the time the save was done.[15]

`LOAD;FILE:`***name*****;** The current state of the storage area is discarded and then reloaded based on the contents of the file named *name.tc*. The file is assumed to be in the format produced by the `SAVE` command.

`STORE-COMMENT;WITH:`***string*****;**`AT:`***slot-address*****;** The comment *string* is stored in the element of storage labelled by *slot-address*.

`STORE-VALUE;WITH:`***number*****;**`AT:`***slot-address*****;** The value *number*[16] is stored in the element labelled by *slot-address*. After this command has been executed, this element will not depend on any other elements.

`BLANK;SLOT:`***slot-address*****;** The element labelled by *slot-address* becomes blank.

`QUIT;` The execution of *TriviCalc* is terminated and control is returned to the executive.

Movement Commands

The following movement commands affect the position of the current slot of the storage area:[17]

CONTROL P The current slot is changed to be the slot vertically above the current slot, if there is one. If the row of the current slot is 1, the command has no effect.

CONTROL N The current slot is changed to be the slot vertically below the current slot, if there is one. If the row of the current slot is 21, the command has no effect.

CONTROL B The current slot is changed to be the slot to the left of the current slot, if there is one. If the column of the current slot is *A*, the command has no effect.

CONTROL F The current slot is changed to be the slot to the right of the current slot, if there is one. If the column of the current slot is *H*, the command has no effect.

Binary Operator Commands

Binary operators commands take three arguments, each of which is a label of an element of the storage area. The command relates the elements of storage labelled by the first two arguments to a result stored in the element of storage labelled by the third argument. If *slot3* labels an element of storage that is already the result of a relation, the old relation is discarded in favour of the new one.[18] All these commands are of the form

op;`VALUE1:`*slot1*;`VALUE2:`*slot2*;`GIVING:` *slot3*;

Op may have the following values:[19]

`ADD` **or** $+$ The element of storage labelled *slot3* is related to the other two elements as (*slot1*+*slot2*).

[15] Another incompleteness: it is not clear whether the cursor position is to be restored also. This relates to an earlier ambiguity (not noted): do we save *movement* commands?

[16] Nowhere is it made clear what a *number* is to look like.

[17] The author of this document has not read Section 3.7 of Mynatt (1990), and the following sentence in particular: "The use of obscure combinations of control, option, or shift keys with other key presses to move the cursor should be avoided."!

[18] This is the third statement of this consistency requirement! However, they fail to (re-)state the first requirement, that only slots containing values may be depended upon. What is to happen if (say) *slot1* or *slot2* are blank?

[19] Here is another ambiguity: who has the choice whether to use `ADD` or $+$? Can we, as implementors, make the choice to allow only one, or must both be available to the user?

`SUBTRACT` **or** $-$ The element of storage labelled *slot3* is related to the other two elements as (*slot1*$-$*slot2*).

`MULTIPLY` **or** $\star$ The element of storage labelled *slot3* is related to the other two elements as (*slot1*$\star$*slot2*).

`DIVIDE` **or /** The element of storage labelled *slot3* is related to the other two elements as (*slot1*/*slot2*).[20]

The Working-Area Editor

The working-area editor is a simple modeless editor with special functions to simplify the input of commands to the *TriviCalc* command processor. The editor maintains a cursor in the working area. Every keystroke is considered to be a command to the editor. All commands are atomic; they are either processed to completion immediately or halt in error, after doing nothing except possibly displaying an error message. Some keystrokes denote textual values (the characters, numerals and punctuation keys). The command that is run by typing any of these keystrokes merely inserts the key's textual value at the cursor.[21] These are known as *textual input commands*.

Other keystrokes do not denote textual values. These are special keys (such the carriage-return or delete), or are typed by holding down the CONTROL key and pressing some other key. These non textual keystrokes are interpreted by the editor as commands that affect the text in the working area. A brief description of the nontextual commands follows.

CONTROL L Move the cursor to the left one position.

CONTROL R Move the cursor to the right one position.

CONTROL D Delete the character at the cursor, if there is one.

DELETE Delete the character to the left of the cursor, if there is one.

CONTROL A Operator Adjust. If the working area is of the form

slot1 op slot2

where *op* is one of the characters $\star$, /, $-$ or $+$ and *slot1*, *slot2* are strings, the contents of the working area are replaced with

op;`VALUE1:`*slot1*;`VALUE2:`*slot2*;`GIVING:%;`

Otherwise, if the contents of the working area represent a valid numerical value, the working area is interpreted as *number* and its contents replaced by

`STORE-VALUE;WITH:`*number*;`AT:%;`

Otherwise, the working area is interpreted as *string* and its contents are replaced by

`STORE-COMMENT;WITH:`*string*;`AT:%;`

Finally, the effect of a CONTROL K command with the cursor at the beginning of the working area, followed by a CONTROL E command is simulated. The effect of this is to replace the % character with the address of the current slot.

[20] What happens if the value of *slot2* is zero?

[21] Where precisely? Another example of ambiguity/incompleteness.

CONTROL K Search from the position to the right for a %, wrapping around to the beginning of the working area if the end of the working area is reached. If a % is found, delete it and leave the cursor at its position. If none is found, do nothing.[22]

CONTROL E The address of the current slot in the storage area is inserted at the cursor.

`RETURN` Pass the contents of the working area to the command interpreter for immediate execution. If the command interpreter does not flag an error, clear the contents of the working area.

`LINE-FEED` Same as `RETURN`, but leave the contents of the working area unaffected in all cases.

CONTROL V – Insert a textual representation of the contents of the current slot into the working area at the cursor.

CONTROL U – Delete the entire contents of the working area.

CONTROL W – The working area must be of the form *digit string*, where *digit* is between 1 and 9. The effect of the command is to cause the *string* to be saved at internal location *digit*. When *TriviCalc* is first started up, the contents of the first four internal locations[23] are the strings:

1: `SAVE;FILE:%;`

2: `LOAD;FILE:%;`

3: `QUIT;`

3: `BLANK;SLOT:%;`

Internal locations 5 to 9 contain the empty string initially.

CONTROL X The working area must be of the form *digit string*, where *digit* is between 1 and 9 and *string* may be empty. The effect of the command is to cause the contents of the working area to be replaced by the string stored at internal location *digit*. Then the effect of a CONTROL K followed by a CONTROL E is simulated, causing the % character, if it is present, to be replaced with the address of the current slot.[24]

Every time a nontextual editing command causes the working area's contents to change, the contents are first saved in internal location 0. Entering CONTROL X when the working area is of the form 0*string* causes the current contents of the working area and internal location 0 to be swapped.

B.2 Original *TriviCalc* design

Below is the original Augmented HOOD design for the *TriviCalc* problem.

```
OBJECT pop_11 IS ENVIRONMENT PASSIVE
  PROVIDED_INTERFACE
    TYPES
      list ;
      channel ;
      property_table ;
    OPERATIONS
      open ( file_name : IN string ; mode : IN string ) RETURN channel ;
```

[22] What happens to the cursor in this case? Where does it end?

[23] What are these?

[24] Incompleteness: what happens if the % character is *not* present?

```
      close_file ( channel : IN channel ) ;
      matches ( pattern : IN list ; datum : IN list ) RETURN boolean ;
      parse_string ( text : IN string ) RETURN list ;
      isstring ( text : IN string ) RETURN boolean ;
      read_line ( channel : IN channel ) RETURN string ;
      get_input_char ( channel : IN channel ) RETURN character ;
      write_line ( channel : IN channel ; text : IN string ) ;
      sysexit ;
END_OBJECT pop_11



OBJECT trivicalc IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      main_program ;
  INTERNALS
    OBJECTS
      cli ;
      data_types ;
      dm ;
      em ;
      sa ;
      wae ;

    OPERATIONS
      main_program
        IMPLEMENTED_BY cli.main_program ;
END_OBJECT trivicalc



OBJECT wae IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      editor ;
      get_cs RETURN slot_id ;
      init_cl ;
      init_cs ;
      is_macro_name ( id : IN string ) RETURN boolean ;
      recall_all_macros RETURN list_strings ;
      set_cs ( slot : IN slot_id ) ;
      store_macro ( id : IN integer ; text : IN string ) ;

  REQUIRED_INTERFACE
    OBJECT cli
      OPERATIONS
        command_despatcher ( command : IN string ) RETURN validity
    OBJECT dm
      OPERATIONS
        delete_char ;
        delete_char_at_left ;
        delete_line ;
        display_cl_line ( text : IN string ) ;
        insert_char ( char : IN character ) ;
        insert_string ( text : IN string ) ;
        move_cs ( old_cs : IN slot_id ; cs : IN slot_id ) ;
        position_cl_cursor ( cursor : IN cursor_position ) ;
        ring_bell ;
```

```
        set_cs ( slot : IN slot_id ) ;
    OBJECT em
      OPERATIONS
        escape_seen ;
    OBJECT sa
      OPERATIONS
        get_contents ( slot : IN slot_id ) RETURN content ;

  INTERNALS
    OBJECTS
      cl ;  ;;; command line
      cp ;  ;;; command processor
      cs ;  ;;; current slot
      il ;  ;;; internal locations
    OPERATIONS
      editor
        IMPLEMENTED_BY cp.editor ;
      init_cl
        IMPLEMENTED_BY cl.init_cl ;
      init_cs
        IMPLEMENTED_BY cs.init_cs ;
      recall_all_macros RETURN list_strings
        IMPLEMENTED_BY il.recall_all_macros RETURN list_strings ;
      set_cs ( slot : IN slot_id )
        IMPLEMENTED_BY cs.set_cs ( slot : IN slot_id ) ;
      get_cs RETURN slot_id
        IMPLEMENTED_BY cs.get_cs RETURN slot_id ;
      is_macro_name ( id : IN string ) RETURN boolean
        IMPLEMENTED_BY il.is_macro_name
                    ( id : IN string ) RETURN boolean ;
      store_macro ( id : IN integer ; text : IN string )
        IMPLEMENTED_BY il.store_macro ( id : IN integer ;
                                        text : IN string ) ;
END_OBJECT wae


OBJECT cs IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      get_cs RETURN slot_id ;
      get_cs_content RETURN content ;
      init_cs ;
      move_cs_down ;
      move_cs_left ;
      move_cs_right ;
      move_cs_up ;
      set_cs ( slot : IN slot_id ) ;

  REQUIRED_INTERFACE
    OBJECT dm
      OPERATIONS
        move_cs ( old_cs : IN slot_id ; cs : IN slot_id ) ;
        set_cs ( slot : IN slot_id ) ;
    OBJECT sa
      OPERATIONS
        get_contents ( slot : IN slot_id ) RETURN content ;

  INTERNALS
```

```
OPERATIONS
  get_cs RETURN slot_id ;
  get_cs_content RETURN content ;
  init_cs ;
  move_cs_down ;
  move_cs_left ;
  move_cs_right ;
  move_cs_up ;
  set_cs ( slot : IN slot_id ) ;
DATA
  current_slot : slot_id ;


OPERATION_CONTROL_STRUCTURES

  OPERATION move_cs_up
    USED_OPERATIONS
      dm.move_cs ( old_cs : IN slot_id ; cs : IN slot_id ) ;
      min_row ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          current_slot ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION move_cs_up

  OPERATION move_cs_down
    USED_OPERATIONS
      dm.move_cs ( old_cs : IN slot_id ; cs : IN slot_id ) ;
      max_row ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          current_slot ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION move_cs_down

  OPERATION move_cs_left
    USED_OPERATIONS
      dm.move_cs ( old_cs : IN slot_id ; cs : IN slot_id ) ;
      min_column ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          current_slot ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION move_cs_left

  OPERATION move_cs_right
    USED_OPERATIONS
      dm.move_cs ( old_cs : IN slot_id ; cs : IN slot_id ) ;
      max_column ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          current_slot ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION move_cs_right
```

```
      OPERATION get_cs RETURN slot_id
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            READS_FROM
              current_slot ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION get_cs

      OPERATION set_cs ( slot : IN slot_id )
        USED_OPERATIONS
          dm.move_cs ( old_cs : IN slot_id ; cs : IN slot_id ) ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              current_slot ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION set_cs

      OPERATION get_cs_content RETURN content
        USED_OPERATIONS
          sa.get_contents ( slot : IN slot_id ) RETURN content ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            READS_FROM
              current_slot ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION get_cs_content

      OPERATION init_cs
        USED_OPERATIONS
          dm.set_cs ( slot : IN slot_id ) ;
          min_row ;
          min_column ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              current_slot ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION init_cs

END_OBJECT cs


OBJECT cl IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      init_cl ;
      cursor_left ;
      cursor_right ;
      delete_line ;
      delete_char ;
      delete_char_left ;
      insert_string ( text : IN string ) ;
      locate_sol ;
      get_cl RETURN string ;
      replace_percent RETURN validity ;
```

```
REQUIRED_INTERFACE
  OBJECT dm
    OPERATIONS
      delete_char ;
      delete_char_at_left ;
      delete_line ;
      display_cl_line ( text : IN string ) ;
      insert_char ( char : IN character ) ;
      insert_string ( text : IN string ) ;
      position_cl_cursor ( cursor : IN cursor_position ) ;
      ring_bell ;


INTERNALS
  CONSTANTS
    max_cursor : integer ;
    max_length : integer ;
    min_cursor : integer ;
    min_length : integer ;
  OPERATIONS
    cursor_left ;
    cursor_right ;
    delete_char ;
    delete_char_left ;
    delete_line ;
    get_cl RETURN string ;
    init_cl ;
    insert_char ( char : IN character ) ;
    insert_string ( text : IN string ) ;
    locate_sol ;
    replace_percent RETURN validity ;
  DATA
    cursor : integer ;
    eos  : integer ;
    line   : string ;

  OPERATION_CONTROL_STRUCTURES

    OPERATION cursor_left
      USED_OPERATIONS
        dm.position_cl_cursor ( cursor : IN cursor_position ) ;
        min_cursor ;
      PSEUDO_CODE
        OPERATION_REQUIREMENTS
          WRITES_TO
            cursor ;
        END_OPERATION_REQUIREMENTS
    END_OPERATION cursor_left

    OPERATION cursor_right
      USED_OPERATIONS
        dm.position_cl_cursor ( cursor : IN cursor_position ) ;
      PSEUDO_CODE
        OPERATION_REQUIREMENTS
          READS_FROM
            eos ;
          WRITES_TO
            cursor ;
```

```
    END_OPERATION_REQUIREMENTS
END_OPERATION cursor_right

OPERATION insert_char ( char : IN character )
  USED_OPERATIONS
    dm.insert_char ( char : IN character ) ;
    dm.ring_bell ;
    max_length ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        cursor ;
        eos ;
        line ;
    END_OPERATION_REQUIREMENTS
END_OPERATION insert_char

OPERATION insert_string ( text : IN string )
  USED_OPERATIONS
    dm.position_cl_cursor ( cursor : IN cursor_position ) ;
    dm.display_cl_line ( text : IN string ) ;
    dm.ring_bell ;
    max_length ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        cursor ;
        eos ;
        line ;
    END_OPERATION_REQUIREMENTS
END_OPERATION insert_string

OPERATION delete_char
  USED_OPERATIONS
    dm.delete_char ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      READS_FROM
        cursor ;
      WRITES_TO
        eos ;
        line ;
    END_OPERATION_REQUIREMENTS
END_OPERATION delete_char

OPERATION delete_char_left
  USED_OPERATIONS
    dm.delete_char_at_left ;
    min_cursor ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        cursor ;
        eos ;
        line ;
    END_OPERATION_REQUIREMENTS
END_OPERATION delete_char_left
```

```
      OPERATION delete_line
        USED_OPERATIONS
          dm.delete_line ;
          min_cursor ;
          min_length ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              cursor ;
              eos ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION delete_line

      OPERATION replace_percent RETURN validity
        USED_OPERATIONS
          dm.position_cl_cursor ( cursor : IN cursor_position ) ;
          delete_char ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            REQUIRES_TYPE
              string ;
              boolean ;
            READS_FROM
              cursor ;
              eos ;
              line ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION replace_percent

      OPERATION locate_sol
        USED_OPERATIONS
          dm.position_cl_cursor ( cursor : IN cursor_position ) ;
          min_cursor ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              cursor ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION locate_sol

      OPERATION get_cl RETURN string
        USED_OPERATIONS
          min_length ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            READS_FROM
              cursor ;
              eos ;
              line ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION get_cl

      OPERATION init_cl
        USED_OPERATIONS
          delete_line ;
      END_OPERATION init_cl

END_OBJECT cl
```

```
OBJECT il IS PASSIVE
  PROVIDED_INTERFACE
    CONSTANTS
      old_cl : integer ;
    OPERATIONS
      is_macro_name ( id : IN string ) RETURN boolean ;
      store_macro ( id : IN integer ; text : IN string ) ;
      recall_macro ( id : IN integer ) RETURN string ;

  INTERNALS
    CONSTANTS
      max_macro : integer ;
      min_macro : integer ;
      old_cl    : integer ;
    OPERATIONS
      store_macro ( id : IN integer ; text : IN string ) ;
      recall_macro ( id : IN integer ) RETURN string ;
      recall_all_macros RETURN list_strings ;
      is_macro_name ( id : IN string ) RETURN boolean ;
      init_il ;
    DATA
      macros : string ;

    OPERATION_CONTROL_STRUCTURES

      OPERATION store_macro ( id : IN integer ; text : IN string )
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              macros ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION store_macro

      OPERATION recall_macro ( id : IN integer ) RETURN string
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            READS_FROM
              macros ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION recall_macro

      OPERATION recall_all_macros RETURN list_strings
        USED_OPERATIONS
          max_macro ;
          min_macro ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            READS_FROM
              macros ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION recall_all_macros

      OPERATION is_macro_name ( id : IN string ) RETURN boolean
        USED_OPERATIONS
          max_macro ;
          min_macro ;
      END_OPERATION is_macro_name
```

```
      OPERATION init_il
        USED_OPERATIONS
          max_macro ;
          min_macro ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              macros ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION init_il

END_OBJECT il



OBJECT cp IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      editor ;

  REQUIRED_INTERFACE
    OBJECT cli
      OPERATIONS
        command_despatcher ( command : IN string ) RETURN validity
  INTERNALS
    OBJECTS
      editor ;  ;;; main editor
      ccp ;     ;;; Control Character Processing
    OPERATIONS
      editor
        IMPLEMENTED_BY editor.editor ;
END_OBJECT cp



OBJECT editor IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      editor ;

  REQUIRED_INTERFACE
    OBJECT ccp
      OPERATIONS
        process_control_char ( char : IN character ) ;
        is_control_char ( char : IN character ) RETURN boolean ;
    OBJECT dm
      OPERATIONS
        ring_bell ;
    OBJECT em
      OPERATIONS
        escape_seen ;
    OBJECT pop_11
      OPERATIONS
        get_input_char ( channel : IN channel ) RETURN character ;
        open ( file_name : IN string ; mode : IN string ) RETURN channel ;
        close_file ( channel : IN channel ) ;

  INTERNALS
```

```
OPERATIONS
  editor ;
  get_char RETURN character ;
  is_printable ( char : IN character ) RETURN boolean ;
  is_escape ( char : IN character ) RETURN boolean ;
  init_cp ;
  term_cp ;
DATA
  channel : channel ;

OPERATION_CONTROL_STRUCTURES

  OPERATION editor
    USED_OPERATIONS
      ccp.is_control_char ( char : IN character ) RETURN boolean ;
      ccp.process_control_char ( char : IN character ) ;
      dm.ring_bell ;
      em.escape_seen ;
      init_cp ;
      get_char RETURN character ;
      is_escape ( char : IN character ) RETURN boolean ;
      is_printable ( char : IN character ) RETURN boolean ;
  END_OPERATION editor

  OPERATION get_char RETURN character
    USED_OPERATIONS
      pop_11.get_input_char ( channel : IN channel )
                              RETURN character ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        READS_FROM
          channel ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION get_char

  OPERATION is_printable ( char : IN character ) RETURN boolean
  END_OPERATION is_printable

  OPERATION is_escape ( char : IN character ) RETURN boolean
  END_OPERATION is_escape

  OPERATION init_cp
    USED_OPERATIONS
      pop_11.open ( file_name : IN string ;
                    mode : IN string ) RETURN channel ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          channel ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION init_cp

  OPERATION term_cp
    USED_OPERATIONS
      pop_11.close_file ( channel : IN channel ) ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
```

```
            channel ;
        END_OPERATION_REQUIREMENTS
    END_OPERATION term_cp

END_OBJECT editor


OBJECT ccp IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      process_control_char ( char : IN character ) ;
      is_control_char ( char : IN character ) RETURN boolean ;
  REQUIRED_INTERFACE
    OBJECT cl
      OPERATIONS
        cursor_left ;
        cursor_right ;
        get_cl RETURN string ;
        delete_line ;
        delete_char ;
        delete_char_left ;
        insert_string ( text : IN string ) ;
        locate_sol ;
        replace_percent RETURN validity ;
    OBJECT cs
      OPERATIONS
        move_cs_down ;
        move_cs_left ;
        move_cs_right ;
        move_cs_up ;
        get_cs RETURN slot_id ;
        get_cs_content RETURN content ;
    OBJECT il
      CONSTANTS
        old_cl ;
      OPERATIONS
        store_macro ( id : IN integer ; text : IN string ) ;
        recall_macro ( id : IN integer ) RETURN string ;
        is_macro_name ( id : IN string ) RETURN boolean ;
    OBJECT cli
      OPERATIONS
        command_despatcher ( command : IN string ) RETURN validity
    OBJECT em
      OPERATIONS
        report_error ( text : IN string ) ;
    OBJECT sa
      OPERATIONS
        is_comment ( text : IN string ) RETURN boolean ;
        is_float ( text : IN string ) RETURN boolean ;
        is_operation ( text : IN string ) RETURN boolean ;
        is_slot ( text : IN string ) RETURN boolean ;
    OBJECT pop_11
      TYPES
        property_table ;
      OPERATIONS
        matches ( pattern : IN list ; datum : IN list ) RETURN boolean ;

  INTERNALS
```

```
CONSTANTS
  despatch_table : pop_11.property_table ;
OPERATIONS
  ccp_adjust ;
  ccp_cli ;
  ccp_cli_keep ;
  ccp_delete_char ;
  ccp_delete_char_left ;
  ccp_delete_line ;
  ccp_get_cs ;
  ccp_get_cs_content ;
  ccp_recall_macro ;
  ccp_replace_percent ;
  ccp_store_macro ;
  is_control_char ( char : IN character ) RETURN boolean ;
  obey_cl RETURN validity ;
  process_control_char ( char : IN character ) ;
  replace_cl ( text : IN string ) ;
  save_cl ;

OPERATION_CONTROL_STRUCTURES

  OPERATION is_control_char ( char : IN character ) RETURN boolean
    USED_OPERATIONS
      despatch_table ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        REQUIRES_TYPE
          pop_11.property_table ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION is_control_char

  OPERATION process_control_char ( char : IN character )
    USED_OPERATIONS
      despatch_table ;
      ccp_adjust ;
      ccp_cli ;
      ccp_cli_keep ;
      ccp_delete_char ;
      ccp_delete_char_left ;
      ccp_delete_line ;
      ccp_get_cs ;
      ccp_get_cs_content ;
      ccp_recall_macro ;
      ccp_replace_percent ;
      ccp_store_macro ;
      cl.cursor_left ;
      cl.cursor_right ;
      cs.move_cs_down ;
      cs.move_cs_left ;
      cs.move_cs_right ;
      cs.move_cs_up ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        REQUIRES_TYPE
          pop_11.property_table ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION process_control_char
```

```
OPERATION save_cl
  USED_OPERATIONS
    il.store_macro ( id : IN integer ; text : IN string ) ;
    il.old_cl ;
    cl.get_cl RETURN string ;
END_OPERATION save_cl

OPERATION obey_cl RETURN validity
  USED_OPERATIONS
    cl.get_cl RETURN string ;
    cli.command_despatcher ( command : IN string ) RETURN validity
END_OPERATION obey_cl

OPERATION replace_cl ( text : IN string )
  USED_OPERATIONS
    cl.delete_line ;
    cl.insert_string ( text : IN string ) ;
    cl.locate_sol ;
END_OPERATION replace_cl

OPERATION ccp_delete_char
  USED_OPERATIONS
    cl.delete_char ;
    save_cl ;
END_OPERATION ccp_delete_char

OPERATION ccp_delete_char_left
  USED_OPERATIONS
    cl.delete_char_left ;
    save_cl ;
END_OPERATION ccp_delete_char_left

OPERATION ccp_delete_line
  USED_OPERATIONS
    cl.delete_line ;
    save_cl ;
END_OPERATION ccp_delete_line

OPERATION ccp_get_cs
  USED_OPERATIONS
    cl.insert_string ( text : IN STRING ) ;
    cs.get_cs RETURN slot_id ;
    save_cl ;
END_OPERATION ccp_get_cs

OPERATION ccp_get_cs_content
  USED_OPERATIONS
    cl.insert_string ( text : IN STRING ) ;
    cs.get_cs_content RETURN content ;
    save_cl ;
END_OPERATION ccp_get_cs_content

OPERATION ccp_adjust
  USED_OPERATIONS
    cl.get_cl RETURN string ;
    cl.insert_string ( text : IN string ) ;
    cl.replace_percent RETURN validity ;
```

```
    cs.get_cs RETURN slot_id ;
    em.report_error ( text : IN string ) ;
    sa.is_comment ( text : IN string ) RETURN boolean ;
    sa.is_float ( text : IN string ) RETURN boolean ;
    sa.is_operation ( text : IN string ) RETURN boolean ;
    sa.is_slot ( text : IN string ) RETURN boolean ;
    pop_11.matches ( pattern : IN list ;
                     datum : IN list ) RETURN boolean ;
    replace_cl ( text : IN string ) ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        float ;
        operation_math ;
        s_string ;
    END_OPERATION_REQUIREMENTS
END_OPERATION ccp_adjust

OPERATION ccp_replace_percent
  USED_OPERATIONS
    cl.replace_percent RETURN validity ;
    save_cl ;
END_OPERATION ccp_replace_percent

OPERATION ccp_cli
  USED_OPERATIONS
    cl.delete_line ;
    save_cl ;
    obey_cl RETURN validity ;
END_OPERATION ccp_cli

OPERATION ccp_cli_keep
  USED_OPERATIONS
    save_cl ;
    obey_cl RETURN validity ;
END_OPERATION ccp_cli_keep

OPERATION ccp_store_macro
  USED_OPERATIONS
    cl.get_cl RETURN string ;
    em.report_error ( text : IN string ) ;
    il.is_macro_name ( id : IN string ) RETURN boolean ;
    il.old_cl ;
    il.store_macro ( id : IN integer ; text : IN string ) ;
    pop_11.matches ( pattern : IN list ;
                     datum : IN list ) RETURN boolean ;
END_OPERATION ccp_store_macro

OPERATION ccp_recall_macro
  USED_OPERATIONS
    cl.get_cl RETURN string ;
    em.report_error ( text : IN string ) ;
    il.is_macro_name ( id : IN string ) RETURN boolean ;
    il.old_cl ;
    il.recall_macro ( id : IN integer ) RETURN string ;
    il.store_macro ( id : IN integer ; text : IN string ) ;
    replace_cl ( text : IN string ) ;
END_OPERATION ccp_recall_macro
```

```
END_OBJECT ccp


OBJECT cli IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      command_despatcher ( command : IN string ) RETURN validity
      main_program ;

  REQUIRED_INTERFACE
    OBJECT dm
      OPERATIONS
        init_dm ;
    OBJECT em
      OPERATIONS
        init_em ;
        report_error ( text : IN string ) ;
    OBJECT sa
      OPERATIONS
        init_sa ;
        is_comment ( text : IN string ) RETURN boolean ;
        is_float ( text : IN string ) RETURN boolean ;
        is_operation ( text : IN string ) RETURN boolean ;
        is_slot ( text : IN string ) RETURN boolean ;
        save_sa RETURN list_strings ;
        set_slot ( slot : IN slot_id ;
                   value : IN content ) RETURN validity ;
    OBJECT wae
      OPERATIONS
        editor ;
        get_cs RETURN slot_id ;
        init_cl ;
        init_cs ;
        is_macro_name ( id : IN string ) RETURN boolean ;
        recall_all_macros RETURN list_strings ;
        set_cs ( slot : IN slot_id ) ;
        store_macro ( id : IN integer ; text : IN string ) ;
    OBJECT pop_11
      OPERATIONS
        close_file ( channel : IN channel ) ;
        matches ( pattern : IN list ; datum : IN list ) RETURN boolean ;
        open ( file_name : IN string ; mode : IN string ) RETURN channel ;
        parse_string ( text : IN string ) RETURN list ;
        read_line ( channel : IN channel ) RETURN string ;
        sysexit ;
        write_line ( channel : IN channel ; text : IN string ) ;
  INTERNALS
    OPERATIONS
      command_despatcher ( command : IN string ) RETURN validity
      full_file_name ( name : IN string ) RETURN string
      is_file_name ( name : IN string ) RETURN boolean
      load_file ( command : IN list_strings ) RETURN validity
      main_program ;
      quit ( command : IN list_strings ) RETURN validity
      reinitialise_system ;
      save_file ( command : IN list_strings ) RETURN validity
      set_blank ( command : IN list_strings ) RETURN validity
```

```
  set_current_slot ( command : IN list_strings ) RETURN validity
  store_comment ( command : IN list_strings ) RETURN validity
  store_expression ( command : IN list_strings ) RETURN validity
  store_macro ( command : IN list_strings ) RETURN validity
  store_value ( command : IN list_strings ) RETURN validity
DATA
  load_in_progress : boolean ;

OPERATION_CONTROL_STRUCTURES

  OPERATION command_despatcher ( command : IN string ) RETURN validity
    USED_OPERATIONS
      pop_11.parse_string ( text : IN string ) RETURN list ;
      store_comment( command : IN list_strings ) RETURN validity
      store_value ( command : IN list_strings ) RETURN validity
      set_blank ( command : IN list_strings ) RETURN validity
      store_expression ( command : IN list_strings ) RETURN validity
      quit ( command : IN list_strings ) RETURN validity
      save_file ( command : IN list_strings ) RETURN validity
      load_file ( command : IN list_strings ) RETURN validity
      set_current_slot ( command : IN list_strings ) RETURN validity
      store_macro ( command : IN list_strings ) RETURN validity
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        READS_FROM
          load_in_progress ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION command_despatcher

  OPERATION store_comment
               ( command : IN list_strings ) RETURN validity
    USED_OPERATIONS
      pop_11.matches ( pattern : IN list ;
                       datum : IN list ) RETURN boolean ;
      sa.set_slot ( slot : IN slot_id ;
                    value : IN content ) RETURN validity ;
      sa.is_comment ( text : IN string ) RETURN boolean ;
      sa.is_slot ( text : IN string ) RETURN boolean ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        REQUIRES_TYPE
          s_string ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION store_comment

  OPERATION store_value ( command : IN list_strings ) RETURN validity
    USED_OPERATIONS
      pop_11.matches ( pattern : IN list ;
                       datum : IN list ) RETURN boolean ;
      sa.set_slot ( slot : IN slot_id ;
                    value : IN content ) RETURN validity ;
      sa.is_float ( text : IN string ) RETURN boolean ;
      sa.is_slot ( text : IN string ) RETURN boolean ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        REQUIRES_TYPE
          float ;
          list ;
```

```
    END_OPERATION_REQUIREMENTS
END_OPERATION store_value

OPERATION set_blank ( command : IN list_strings ) RETURN validity
  USED_OPERATIONS
    pop_11.matches ( pattern : IN list ;
                     datum : IN list ) RETURN boolean ;
    sa.set_slot ( slot : IN slot_id ;
                  value : IN content ) RETURN validity ;
    sa.is_slot ( text : IN string ) RETURN boolean ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        s_string ;
    END_OPERATION_REQUIREMENTS
END_OPERATION set_blank

OPERATION store_expression
              ( command : IN list_strings ) RETURN validity
  USED_OPERATIONS
    pop_11.matches ( pattern : IN list ;
                     datum : IN list ) RETURN boolean ;
    sa.set_slot ( slot : IN slot_id ;
                  value : IN content ) RETURN validity ;
    sa.is_slot ( text : IN string ) RETURN boolean ;
    sa.is_operation ( text : IN string ) RETURN boolean ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        operation_full ;
        operation_math ;
        data_types.expression ;
    END_OPERATION_REQUIREMENTS
END_OPERATION store_expression

OPERATION quit ( command : IN list_strings ) RETURN validity
  USED_OPERATIONS
    pop_11.matches ( pattern : IN list ;
                     datum : IN list ) RETURN boolean ;
    pop_11.sysexit ;
END_OPERATION quit

OPERATION save_file ( command : IN list_strings ) RETURN validity
  USED_OPERATIONS
    pop_11.matches ( pattern : IN list ;
                     datum : IN list ) RETURN boolean ;
    pop_11.open ( file_name : IN string ;
                  mode : IN string ) RETURN channel ;
    pop_11.write_line ( channel : IN channel ; text : IN string ) ;
    pop_11.close_file ( channel : IN channel ) ;
    is_file_name ( name : IN string ) RETURN boolean ;
    full_file_name ( name : IN string ) RETURN string ;
    em.report_error ( text : IN string ) ;
    sa.save_sa RETURN list_strings ;
    wae.get_cs RETURN slot_id ;
    wae.recall_all_macros RETURN list_strings ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
```

```
      REQUIRES_TYPE
        integer ;
    END_OPERATION_REQUIREMENTS
END_OPERATION save_file

OPERATION load_file ( command : IN list_strings ) RETURN validity
  USED_OPERATIONS
    pop_11.matches ( pattern : IN list ;
                     datum : IN list ) RETURN boolean ;
    pop_11.open ( file_name : IN string ;
                  mode : IN string ) RETURN channel ;
    pop_11.read_line ( channel : IN channel ) RETURN string ;
    pop_11.close_file ( channel : IN channel ) ;
    is_file_name ( name : IN string ) RETURN boolean ;
    full_file_name ( name : IN string ) RETURN string ;
    reinitialise_system ;
    command_despatcher ( command : IN string ) RETURN validity ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        load_in_progress ;
    END_OPERATION_REQUIREMENTS
END_OPERATION load_file

OPERATION set_current_slot
              ( command : IN list_strings ) RETURN validity
  USED_OPERATIONS
    pop_11.matches ( pattern : IN list ;
                     datum : IN list ) RETURN boolean ;
    sa.is_slot ( text : IN string ) RETURN boolean ;
    wae.set_cs ( slot : IN slot_id ) ;
END_OPERATION set_current_slot

OPERATION store_macro ( command : IN list_strings ) RETURN validity
  USED_OPERATIONS
    pop_11.matches ( pattern : IN list ;
                     datum : IN list ) RETURN boolean ;
    wae.is_macro_name ( id : IN string ) RETURN boolean ;
    wae.store_macro ( id : IN integer ; text : IN string ) ;
END_OPERATION store_macro

OPERATION is_file_name ( name : IN string ) RETURN boolean
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        integer ;
    END_OPERATION_REQUIREMENTS
END_OPERATION is_file_name

OPERATION full_file_name ( name : IN string ) RETURN string
END_OPERATION full_file_name

OPERATION reinitialise_system
  USED_OPERATIONS
    dm.init_dm ;
    wae.init_cl ;
    wae.init_cs ;
    sa.init_sa ;
```

```
      END_OPERATION reinitialise_system

      OPERATION main_program
        USED_OPERATIONS
          em.init_em ;
          reinitialise_system ;
          wae.editor ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              load_in_progress ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION main_program

END_OBJECT cli


OBJECT sa IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      get_contents ( slot : IN slot_id ) RETURN content ;
      init_sa ;
      is_comment ( text : IN string ) RETURN boolean ;
      is_float ( text : IN string ) RETURN boolean ;
      is_operation ( text : IN string ) RETURN boolean ;
      is_slot ( text : IN string ) RETURN boolean ;
      save_sa RETURN list_strings ;
      set_slot ( slot : IN slot_id ; value : IN content )
                RETURN validity ;

  REQUIRED_INTERFACE
    OBJECT dm
      OPERATIONS
        display_value ( slot : IN slot_id ) ;
    OBJECT em
      OPERATIONS
        report_error ( text : IN string ) ;

  INTERNALS
    CONSTANTS
      blank      : slot_type ;
      comment    : slot_type ;
      expression : slot_type ;
      float      : slot_type ;
    OPERATIONS
      get_contents ( slot : IN slot_id ) RETURN content ;
      init_sa ;
      is_comment ( text : IN string ) RETURN boolean ;
      is_float ( text : IN string ) RETURN boolean ;
      is_operation ( text : IN string ) RETURN boolean ;
      is_slot ( text : IN string ) RETURN boolean ;
      save_sa RETURN list_strings ;
      set_slot ( slot : IN slot_id ;
                 value : IN content ) RETURN validity ;
      blank ( slot : IN slot_id ) ;
      address ( slot : IN slot_id ) RETURN slots_index ;
      create_sa ;
      create_slot ( column : IN column_position ;
```

```
              row : IN row_position ) RETURN a_slot ;
  add_successor ( slot : IN slot_id ; to_slot : IN slot_id ) ;
  remove_successor ( slot : IN slot_id ; from_slot : IN slot_id ) ;
  list_successors ( slot : IN slot_id ) RETURN list_slot_ids ;
  is_successor ( slot : IN slot_id ;
                 of_slot : IN slot_id ) RETURN boolean ;
  complete_update ( slot : IN slot_id ; success : IN boolean ) ;
  display_value ( slot : IN slot_id ) ;
  depth_first_search ( slot : IN slot_id ) RETURN slot_id ;
  update_order ( slot : IN slot_id ) RETURN list_slot_ids ;
  update_slots ( slots : IN list_slot_ids ) RETURN validity ;
  evaluate ( slot : IN slot_id ) RETURN full_value ;
  is_slot_arithmetic ( slot : IN slot_id ) RETURN boolean ;
  is_slot_float ( slot : IN slot_id ) RETURN boolean ;
  is_slot_blank ( slot : IN slot_id ) RETURN boolean ;
  is_slot_comment ( slot : IN slot_id ) RETURN boolean ;
  is_slot_expression ( slot : IN slot_id ) RETURN boolean ;
  get_value ( slot : IN slot_id ;
              new_value : IN boolean ) RETURN value ;
DATA
  slots : slot_array ;
  stack : list_slot_ids ;

OPERATION_CONTROL_STRUCTURES

  OPERATION set_slot ( slot : IN slot_id ;
                       value : IN content ) RETURN validity
    USED_OPERATIONS
      em.report_error ( text : IN string ) ;
      address ( slot : IN slot_id ) RETURN slots_index ;
      complete_update ( slot : IN slot_id ; success : IN boolean ) ;
      evaluate ( slot : IN slot_id ) RETURN full_value ;
      is_successor ( slot : IN slot_id ;
                     of_slot : IN slot_id ) RETURN boolean ;
      remove_successor ( slot : IN slot_id ;
                         from_slot : IN slot_id ) ;
      add_successor ( slot : IN slot_id ; to_slot : IN slot_id ) ;
      update_order ( slot : IN slot_id ) RETURN list_slot_ids ;
      update_slots ( slots : IN list_slot_ids ) RETURN validity ;
      blank ;
      comment ;
      expression ;
      float ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        REQUIRES_TYPE
          data_types.expression ;
          full_slot_type ;
          slot_id ;
          slot_type ;
        WRITES_TO
          slots ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION set_slot

  OPERATION init_sa
    USED_OPERATIONS
      address ( slot : IN slot_id ) RETURN slots_index ;
```

```
    blank ( slot : IN slot_id ) ;
    create_sa ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        row_position ;
        column_position ;
    END_OPERATION_REQUIREMENTS
END_OPERATION init_sa

OPERATION create_sa
  USED_OPERATIONS
    create_slot ( column : IN column_position ;
                  row : IN row_position ) RETURN a_slot ;
    min_letter ;
    max_letter ;
    min_row ;
    max_row ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        slot_id ;
      WRITES_TO
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION create_sa

OPERATION create_slot ( column : IN column_position ;
                        row : IN row_position ) RETURN a_slot
  USED_OPERATIONS
    blank ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        boolean ;
        list ;
        slot_type ;
    END_OPERATION_REQUIREMENTS
END_OPERATION create_slot

OPERATION add_successor ( slot : IN slot_id ; to_slot : IN slot_id )
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        list ;
        list_slot_ids ;
      WRITES_TO
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION add_successor

OPERATION remove_successor ( slot : IN slot_id ;
                             from_slot : IN slot_id )
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        list ;
        list_slot_ids ;
```

```
      WRITES_TO
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION remove_successor

OPERATION list_successors ( slot : IN slot_id ) RETURN list_slot_ids
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        list ;
      READS_FROM
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION list_successors

OPERATION is_successor ( slot : IN slot_id ;
                         of_slot : IN slot_id ) RETURN boolean
  USED_OPERATIONS
    list_successors ( slot : IN slot_id ) RETURN list_slot_ids
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        list ;
    END_OPERATION_REQUIREMENTS
END_OPERATION is_successor

OPERATION complete_update ( slot : IN slot_id ;
                            success : IN boolean )
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
    display_value ( slot : IN slot_id ) ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        a_slot ;
        value ;
        boolean ;
      WRITES_TO
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION complete_update

OPERATION display_value ( slot : IN slot_id )
  USED_OPERATIONS
    dm.display_value ( slot : IN slot_id ) ;
END_OPERATION display_value

OPERATION depth_first_search ( slot : IN slot_id ) RETURN slot_id
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        boolean ;
        list ;
        list_slot_ids ;
      READS_FROM
        slots ;
```

```
      WRITES_TO
        stack ;
    END_OPERATION_REQUIREMENTS
END_OPERATION depth_first_search

OPERATION update_order ( slot : IN slot_id ) RETURN list_slot_ids
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
    depth_first_search ( slot : IN slot_id ) RETURN slot_id ;
    list_successors ( slot : IN slot_id ) RETURN list_slot_ids ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        a_slot ;
        boolean ;
        list ;
      WRITES_TO
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION update_order

OPERATION update_slots ( slots : IN list_slot_ids ) RETURN validity
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
    evaluate ( slot : IN slot_id ) RETURN full_value ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        a_slot ;
        list ;
        full_value ;
      WRITES_TO
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION update_slots

OPERATION evaluate ( slot : IN slot_id ) RETURN full_value
  USED_OPERATIONS
    em.report_error ( text : IN string ) ;
    min_float ;
    max_float ;
    false ;
    get_contents ( slot : IN slot_id ) RETURN content ;
    is_slot_arithmetic ( slot : IN slot_id ) RETURN boolean ;
    is_slot_float ( slot : IN slot_id ) RETURN boolean ;
    get_value ( slot : IN slot_id ;
                new_value : IN boolean ) RETURN value ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        float ;
        list ;
        operation_math ;
    END_OPERATION_REQUIREMENTS
END_OPERATION evaluate

OPERATION is_slot_blank ( slot : IN slot_id ) RETURN boolean
  USED_OPERATIONS
```

```
    address ( slot : IN slot_id ) RETURN slots_index ;
    blank ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        a_slot ;
        slot_type ;
      READS_FROM
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION is_slot_blank

OPERATION is_slot_comment ( slot : IN slot_id ) RETURN boolean
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
    comment ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        a_slot ;
        slot_type ;
      READS_FROM
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION is_slot_comment

OPERATION is_slot_float ( slot : IN slot_id ) RETURN boolean
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
    float ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        a_slot ;
        slot_type ;
      READS_FROM
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION is_slot_float

OPERATION is_slot_expression ( slot : IN slot_id ) RETURN boolean
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
    expression ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        a_slot ;
        slot_type ;
      READS_FROM
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION is_slot_expression

OPERATION is_slot_arithmetic ( slot : IN slot_id ) RETURN boolean
  USED_OPERATIONS
    is_slot_expression ( slot : IN slot_id ) RETURN boolean
    is_slot_float ( slot : IN slot_id ) RETURN boolean
```

```
END_OPERATION is_slot_arithmetic

OPERATION blank ( slot : IN slot_id )
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
    blank ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        a_slot ;
        boolean ;
        content ;
        list ;
        list_slot_ids ;
        slot_type ;
        string ;
        value ;
      WRITES_TO
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION blank

OPERATION address ( slot : IN slot_id ) RETURN slots_index
  USED_OPERATIONS
    min_letter ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        integer ;
        string ;
    END_OPERATION_REQUIREMENTS
END_OPERATION address

OPERATION is_slot ( text : IN string ) RETURN boolean
  USED_OPERATIONS
    isstring ( text : IN string ) RETURN boolean ;
    max_column ;
    min_column ;
    max_row ;
    min_row ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        slots_index ;
        slot_id ;
        slot_letter ;
        slot_number ;
    END_OPERATION_REQUIREMENTS
END_OPERATION is_slot

OPERATION is_operation ( text : IN string ) RETURN boolean
  USED_OPERATIONS
    isstring ( text : IN string ) RETURN boolean ;
    "=" ( left : IN string ; right : IN string ) RETURN boolean ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        operation_text ;
```

```
    END_OPERATION_REQUIREMENTS
END_OPERATION is_operation

OPERATION is_float ( text : IN string ) RETURN boolean
  USED_OPERATIONS
    isstring ( text : IN string ) RETURN boolean ;
    max_float ;
    min_float ;
END_OPERATION is_float

OPERATION is_comment ( text : IN string ) RETURN boolean
  USED_OPERATIONS
    isstring ( text : IN string ) RETURN boolean ;
    max_s_string ;
END_OPERATION is_comment

OPERATION get_value ( slot : IN slot_id ;
                      new_value : IN boolean ) RETURN value
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        slots_index ;
      READS_FROM
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION get_value

OPERATION get_contents ( slot : IN slot_id ) RETURN content
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        slots_index ;
      READS_FROM
        slots ;
    END_OPERATION_REQUIREMENTS
END_OPERATION get_contents

OPERATION save_sa RETURN list_strings
  USED_OPERATIONS
    address ( slot : IN slot_id ) RETURN slots_index ;
    depth_first_search ( slot : IN slot_id ) RETURN slot_id ;
    get_contents ( slot : IN slot_id ) RETURN content ;
    is_slot_blank ( slot : IN slot_id ) RETURN boolean ;
    set_slot ( slot : IN slot_id ;
               value : IN content ) RETURN validity ;
    max_column ;
    min_column ;
    max_row ;
    min_row ;
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      REQUIRES_TYPE
        boolean ;
        row_position ;
```

```
              column_position ;
              list ;
              list_slot_ids ;
              list_strings ;
              slots_index ;
            READS_FROM
              slots ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION save_sa

END_OBJECT sa


OBJECT dm IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      clear_error_display ;
      display_error_message ( text : IN string ) ;
      display_value ( slot : IN slot_id ) ;
      move_cs ( old_cs : IN slot_id ; cs : IN slot_id ) ;
      set_cs ( cs : IN slot_id ) ;
      delete_char ;
      delete_char_at_left ;
      delete_line ;
      display_cl_line ( text : IN string ) ;
      insert_char ( char : IN character ) ;
      insert_string ( text : IN string ) ;
      position_cl_cursor ( cursor : IN cursor_position ) ;
      ring_bell ;
      init_dm ;

  INTERNALS
    TYPES
      vdu ;
    OPERATIONS
      clear_error_display ;
      clear_inverse_video ( slot : IN slot_id ) ;
      delete_char ;
      delete_char_at_left ;
      delete_line ;
      display_cl_line ( text : IN string ) ;
      display_content ( content : IN content ) ;
      display_error_message ( text : IN string ) ;
      display_value ( slot : IN slot_id ) ;
      init_dm ;
      insert_char ( char : IN character ) ;
      insert_string ( text : IN string ) ;
      locate_slot ( slot : IN slot_id ) ;
      move_cs ( old_cs : IN slot_id ; cs : IN slot_id ) ;
      position_cl_cursor ( cursor : IN cursor_position ) ;
      reset_dm ;
      ring_bell ;
      set_cs ( cs : IN slot_id ) ;
      set_inverse_video ( slot : IN slot_id ) ;
    DATA
      vdu : vdu ;
```

```
OPERATION_CONTROL_STRUCTURES

  OPERATION clear_error_display
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          vdu ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION ced

  OPERATION clear_inverse_video ( slot : IN slot_id )
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          vdu ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION civ

  OPERATION delete_char
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          vdu ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION dc

  OPERATION delete_char_at_left
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          vdu ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION dcal

  OPERATION delete_line
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          vdu ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION dl

  OPERATION display_cl_line ( text : IN string )
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          vdu ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION dcll

  OPERATION display_content ( content : IN content )
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          vdu ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION dc
```

```
OPERATION display_error_message ( text : IN string )
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        vdu ;
    END_OPERATION_REQUIREMENTS
END_OPERATION dem

OPERATION display_value ( slot : IN slot_id )
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        vdu ;
    END_OPERATION_REQUIREMENTS
END_OPERATION dv

OPERATION init_dm
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        vdu ;
    END_OPERATION_REQUIREMENTS
END_OPERATION idm

OPERATION insert_char ( char : IN character )
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        vdu ;
    END_OPERATION_REQUIREMENTS
END_OPERATION ic

OPERATION insert_string ( text : IN string )
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        vdu ;
    END_OPERATION_REQUIREMENTS
END_OPERATION insert_string

OPERATION locate_slot ( slot : IN slot_id )
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        vdu ;
    END_OPERATION_REQUIREMENTS
END_OPERATION ls

OPERATION move_cs ( old_cs : IN slot_id ; cs : IN slot_id )
  PSEUDO_CODE
    OPERATION_REQUIREMENTS
      WRITES_TO
        vdu ;
    END_OPERATION_REQUIREMENTS
END_OPERATION mcs

OPERATION position_cl_cursor ( cursor : IN cursor_position )
  PSEUDO_CODE
```

```
          OPERATION_REQUIREMENTS
            WRITES_TO
              vdu ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION pclc

      OPERATION reset_dm
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              vdu ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION rdm

      OPERATION ring_bell
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              vdu ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION rb

      OPERATION set_cs ( cs : IN slot_id )
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              vdu ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION scs

      OPERATION set_inverse_video ( slot : IN slot_id )
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            WRITES_TO
              vdu ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION siv

END_OBJECT dm


OBJECT em IS PASSIVE
  PROVIDED_INTERFACE
    OPERATIONS
      escape_seen ;
      init_em ;
      report_error ( text : IN string ) ;
  REQUIRED_INTERFACE
    OBJECT dm
      OPERATIONS
        clear_error_display ;
        display_error_message ( text : IN string ) ;
  INTERNALS
    OPERATIONS
      add_to_queue ( text : IN string ) ;
      display_error_message ( text : IN string ) ;
      escape_seen ;
      init_em ;
```

```
  remove_from_queue RETURN string ;
  report_error ( text : IN string ) ;
DATA
  display_in_use : boolean ;
  error_queue  : list_strings ;

OPERATION_CONTROL_STRUCTURES

  OPERATION report_error ( text : IN string )
    USED_OPERATIONS
      add_to_queue ( text : IN string ) ;
      display_error_message ( text : IN string ) ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        READS_FROM
          display_in_use ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION report_error

  OPERATION escape_seen
    USED_OPERATIONS
      remove_from_queue RETURN string ;
      display_error_message ( text : IN string ) ;
      dm.clear_error_display ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          display_in_use ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION escape_seen

  OPERATION add_to_queue ( text : IN string )
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        REQUIRES_TYPE
          list ;
        WRITES_TO
          error_queue ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION add_to_queue

  OPERATION remove_from_queue RETURN string
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        REQUIRES_TYPE
          list ;
        WRITES_TO
          error_queue ;
      END_OPERATION_REQUIREMENTS
  END_OPERATION remove_from_queue

  OPERATION display_error_message ( text : IN string )
    USED_OPERATIONS
      dm.display_error_message ( text : IN string ) ;
    PSEUDO_CODE
      OPERATION_REQUIREMENTS
        WRITES_TO
          display_in_use ;
```

```
        END_OPERATION_REQUIREMENTS
      END_OPERATION display_error_message

      OPERATION init_em
        USED_OPERATIONS
          dm.clear_error_display ;
        PSEUDO_CODE
          OPERATION_REQUIREMENTS
            REQUIRES_TYPE
              list ;
            WRITES_TO
              display_in_use ;
              error_queue ;
          END_OPERATION_REQUIREMENTS
      END_OPERATION init_em

END_OBJECT em



OBJECT data_types IS PASSIVE
  PROVIDED_INTERFACE
    TYPES
      a_slot ;
      row_position ;
      column_position ;
      content ;
      cursor_position ;
      expression ;
      full_slot_type ;   --{ type+  extended type of a slot
                  false | blank | comment | fp_value | expression }--
      full_value ; --{ value+   extended value of a slot expression
               false | <fp_value> | <s_string> }--
      list_slot_ids ;
      list_strings ;
      operation_math ; --{ + | - | *  | / }--
      operation_full ; --{ <operation_math> | <operation_text> }--
      operation_text ; --{ 'add' | 'subtract' | 'multiply' | 'divide' }--
      s_string ;
      slot_array ;
      slot_id ;
      slot_letter ;
      slot_number ;
      slot_type ; --{ blank | comment | fp_value | expression }--
      slots_index ;
      validity ;
      value ; --{ value of slot expression -- <fp_value> | <s_string> }--

    CONSTANTS
      max_column   : integer ;
      min_column   : integer ;
      max_float  : float ;
      min_float  : float ;
      max_letter   : character ;
      min_letter   : character ;
      max_row    : integer ;
      min_row    : integer ;
      max_s_string : integer ;
      nil      : list ;
```

```
      zero_float   : float ;
END_OBJECT data_types
```

B.3 *TriviCalc* Module Structure

This section presents the modular structure of *TriviCalc*. The nested lists shown delimited by '[%' and '%]' the contents of each object. The kind of each basic entity is indicated, and the full signatures of operations and constants are shown.

The leading 'C' can be ignored, it is just used by *Morpheus* and Pop-11.

B.3.1 Original *TriviCalc* design Module Structure

Below is the original module structure for the *TriviCalc* problem, as derived from the *Morpheus*'s Data Analyser.

```
[%
  [%
    C('<operation>$STANDARD.<<=>>:$STANDARD.STRING*$STANDARD.STRING
                                          ->$STANDARD.BOOLEAN'),
    C('<type>$STANDARD.STRING'),
    C('<type>$STANDARD.CHARACTER'),
    C('<type>$STANDARD.FLOAT'),
    C('<type>$STANDARD.INTEGER'),
    C('<constant>$STANDARD.FALSE:->$STANDARD.BOOLEAN'),
    C('<type>$STANDARD.BOOLEAN'),
  %],
  [%
    C('<operation>POP_11.SYSEXIT'),
    C('<operation>POP_11.WRITE_LINE:POP_11.CHANNEL*$STANDARD.STRING'),
    C('<operation>POP_11.GET_INPUT_CHAR:POP_11.CHANNEL->$STANDARD.CHARACTER'),
    C('<operation>POP_11.READ_LINE:POP_11.CHANNEL->$STANDARD.STRING'),
    C('<operation>POP_11.ISSTRING:$STANDARD.STRING->$STANDARD.BOOLEAN'),
    C('<operation>POP_11.PARSE_STRING:$STANDARD.STRING->POP_11.LIST'),
    C('<operation>POP_11.MATCHES:POP_11.LIST*POP_11.LIST->$STANDARD.BOOLEAN'),
    C('<operation>POP_11.CLOSE_FILE:POP_11.CHANNEL'),
    C('<operation>POP_11.OPEN:$STANDARD.STRING*$STANDARD.STRING
                                          ->POP_11.CHANNEL'),
    C('<type>POP_11.PROPERTY_TABLE'),
    C('<type>POP_11.CHANNEL'),
    C('<type>POP_11.LIST'),
  %],
  [%
    C('<operation>TRIVICALC.MAIN_PROGRAM'),
    [%
      C('<operation>CLI.MAIN_PROGRAM'),
      C('<operation>CLI.REINITIALISE_SYSTEM'),
      C('<operation>CLI.FULL_FILE_NAME:$STANDARD.STRING->$STANDARD.STRING'),
      C('<operation>CLI.IS_FILE_NAME:$STANDARD.STRING->$STANDARD.BOOLEAN'),
      C('<operation>CLI.STORE_MACRO:DATA_TYPES.LIST_STRINGS
                                          ->DATA_TYPES.VALIDITY'),
      C('<operation>CLI.SET_CURRENT_SLOT:DATA_TYPES.LIST_STRINGS
                                          ->DATA_TYPES.VALIDITY'),
      C('<operation>CLI.LOAD_FILE:DATA_TYPES.LIST_STRINGS->DATA_TYPES.VALIDITY'),
      C('<operation>CLI.SAVE_FILE:DATA_TYPES.LIST_STRINGS->DATA_TYPES.VALIDITY'),
      C('<operation>CLI.QUIT:DATA_TYPES.LIST_STRINGS->DATA_TYPES.VALIDITY'),
      C('<operation>CLI.STORE_EXPRESSION:DATA_TYPES.LIST_STRINGS
                                          ->DATA_TYPES.VALIDITY'),
      C('<operation>CLI.SET_BLANK:DATA_TYPES.LIST_STRINGS->DATA_TYPES.VALIDITY'),
      C('<operation>CLI.STORE_VALUE:DATA_TYPES.LIST_STRINGS
                                          ->DATA_TYPES.VALIDITY'),
      C('<operation>CLI.STORE_COMMENT:DATA_TYPES.LIST_STRINGS
```

```
                                        ->DATA_TYPES.VALIDITY'),
    C('<operation>CLI.COMMAND_DESPATCHER:$STANDARD.STRING
                                        ->DATA_TYPES.VALIDITY'),
    C('<data>CLI.LOAD_IN_PROGRESS'),
  %],
  [%
    C('<constant>DATA_TYPES.ZERO_FLOAT:->$STANDARD.FLOAT'),
    C('<constant>DATA_TYPES.NIL:->POP_11.LIST'),
    C('<constant>DATA_TYPES.MAX_S_STRING:->$STANDARD.INTEGER'),
    C('<constant>DATA_TYPES.MIN_ROW:->$STANDARD.INTEGER'),
    C('<constant>DATA_TYPES.MAX_ROW:->$STANDARD.INTEGER'),
    C('<constant>DATA_TYPES.MIN_LETTER:->$STANDARD.CHARACTER'),
    C('<constant>DATA_TYPES.MAX_LETTER:->$STANDARD.CHARACTER'),
    C('<constant>DATA_TYPES.MIN_FLOAT:->$STANDARD.FLOAT'),
    C('<constant>DATA_TYPES.MAX_FLOAT:->$STANDARD.FLOAT'),
    C('<constant>DATA_TYPES.MIN_COLUMN:->$STANDARD.INTEGER'),
    C('<constant>DATA_TYPES.MAX_COLUMN:->$STANDARD.INTEGER'),
    C('<type>DATA_TYPES.VALUE'),
    C('<type>DATA_TYPES.VALIDITY'),
    C('<type>DATA_TYPES.SLOTS_INDEX'),
    C('<type>DATA_TYPES.SLOT_TYPE'),
    C('<type>DATA_TYPES.SLOT_NUMBER'),
    C('<type>DATA_TYPES.SLOT_LETTER'),
    C('<type>DATA_TYPES.SLOT_ID'),
    C('<type>DATA_TYPES.SLOT_ARRAY'),
    C('<type>DATA_TYPES.S_STRING'),
    C('<type>DATA_TYPES.OPERATION_TEXT'),
    C('<type>DATA_TYPES.OPERATION_FULL'),
    C('<type>DATA_TYPES.OPERATION_MATH'),
    C('<type>DATA_TYPES.LIST_STRINGS'),
    C('<type>DATA_TYPES.LIST_SLOT_IDS'),
    C('<type>DATA_TYPES.FULL_VALUE'),
    C('<type>DATA_TYPES.FULL_SLOT_TYPE'),
    C('<type>DATA_TYPES.EXPRESSION'),
    C('<type>DATA_TYPES.CURSOR_POSITION'),
    C('<type>DATA_TYPES.CONTENT'),
    C('<type>DATA_TYPES.COLUMN_POSITION'),
    C('<type>DATA_TYPES.ROW_POSITION'),
    C('<type>DATA_TYPES.A_SLOT'),
  %],
  [%
    C('<operation>DM.SET_INVERSE_VIDEO:DATA_TYPES.SLOT_ID'),
    C('<operation>DM.SET_CS:DATA_TYPES.SLOT_ID'),
    C('<operation>DM.RING_BELL'),
    C('<operation>DM.RESET_DM'),
    C('<operation>DM.POSITION_CL_CURSOR:DATA_TYPES.CURSOR_POSITION'),
    C('<operation>DM.MOVE_CS:DATA_TYPES.SLOT_ID*DATA_TYPES.SLOT_ID'),
    C('<operation>DM.LOCATE_SLOT:DATA_TYPES.SLOT_ID'),
    C('<operation>DM.INSERT_STRING:$STANDARD.STRING'),
    C('<operation>DM.INSERT_CHAR:$STANDARD.CHARACTER'),
    C('<operation>DM.INIT_DM'),
    C('<operation>DM.DISPLAY_VALUE:DATA_TYPES.SLOT_ID'),
    C('<operation>DM.DISPLAY_ERROR_MESSAGE:$STANDARD.STRING'),
    C('<operation>DM.DISPLAY_CONTENT:DATA_TYPES.CONTENT'),
    C('<operation>DM.DISPLAY_CL_LINE:$STANDARD.STRING'),
    C('<operation>DM.DELETE_LINE'),
    C('<operation>DM.DELETE_CHAR_AT_LEFT'),
    C('<operation>DM.DELETE_CHAR'),
```

```
  C('<operation>DM.CLEAR_INVERSE_VIDEO:DATA_TYPES.SLOT_ID'),
  C('<operation>DM.CLEAR_ERROR_DISPLAY'),
  C('<data>DM.VDU'),
  C('<type>DM.VDU'),
%],
[%
  C('<operation>EM.INIT_EM'),
  C('<operation>EM.DISPLAY_ERROR_MESSAGE:$STANDARD.STRING'),
  C('<operation>EM.REMOVE_FROM_QUEUE:->$STANDARD.STRING'),
  C('<operation>EM.ADD_TO_QUEUE:$STANDARD.STRING'),
  C('<operation>EM.ESCAPE_SEEN'),
  C('<operation>EM.REPORT_ERROR:$STANDARD.STRING'),
  C('<data>EM.ERROR_QUEUE'),
  C('<data>EM.DISPLAY_IN_USE'),
%],
[%
  C('<operation>SA.SAVE_SA:->DATA_TYPES.LIST_STRINGS'),
  C('<operation>SA.GET_CONTENTS:DATA_TYPES.SLOT_ID->DATA_TYPES.CONTENT'),
  C('<operation>SA.GET_VALUE:DATA_TYPES.SLOT_ID*$STANDARD.BOOLEAN
                                    ->DATA_TYPES.VALUE'),
  C('<operation>SA.IS_COMMENT:$STANDARD.STRING->$STANDARD.BOOLEAN'),
  C('<operation>SA.IS_FLOAT:$STANDARD.STRING->$STANDARD.BOOLEAN'),
  C('<operation>SA.IS_OPERATION:$STANDARD.STRING->$STANDARD.BOOLEAN'),
  C('<operation>SA.IS_SLOT:$STANDARD.STRING->$STANDARD.BOOLEAN'),
  C('<operation>SA.ADDRESS:DATA_TYPES.SLOT_ID->DATA_TYPES.SLOTS_INDEX'),
  C('<operation>SA.BLANK:DATA_TYPES.SLOT_ID'),
  C('<operation>SA.IS_SLOT_ARITHMETIC:DATA_TYPES.SLOT_ID->$STANDARD.BOOLEAN'),
  C('<operation>SA.IS_SLOT_EXPRESSION:DATA_TYPES.SLOT_ID->$STANDARD.BOOLEAN'),
  C('<operation>SA.IS_SLOT_FLOAT:DATA_TYPES.SLOT_ID->$STANDARD.BOOLEAN'),
  C('<operation>SA.IS_SLOT_COMMENT:DATA_TYPES.SLOT_ID->$STANDARD.BOOLEAN'),
  C('<operation>SA.IS_SLOT_BLANK:DATA_TYPES.SLOT_ID->$STANDARD.BOOLEAN'),
  C('<operation>SA.EVALUATE:DATA_TYPES.SLOT_ID->DATA_TYPES.FULL_VALUE'),
  C('<operation>SA.UPDATE_SLOTS:DATA_TYPES.LIST_SLOT_IDS
                                    ->DATA_TYPES.VALIDITY'),
  C('<operation>SA.UPDATE_ORDER:DATA_TYPES.SLOT_ID
                                    ->DATA_TYPES.LIST_SLOT_IDS'),
  C('<operation>SA.DEPTH_FIRST_SEARCH:DATA_TYPES.SLOT_ID
                                    ->DATA_TYPES.SLOT_ID'),
  C('<operation>SA.DISPLAY_VALUE:DATA_TYPES.SLOT_ID'),
  C('<operation>SA.COMPLETE_UPDATE:DATA_TYPES.SLOT_ID*$STANDARD.BOOLEAN'),
  C('<operation>SA.IS_SUCCESSOR:DATA_TYPES.SLOT_ID*DATA_TYPES.SLOT_ID
                                    ->$STANDARD.BOOLEAN'),
  C('<operation>SA.LIST_SUCCESSORS:DATA_TYPES.SLOT_ID
                                    ->DATA_TYPES.LIST_SLOT_IDS'),
  C('<operation>SA.REMOVE_SUCCESSOR:DATA_TYPES.SLOT_ID*DATA_TYPES.SLOT_ID'),
  C('<operation>SA.ADD_SUCCESSOR:DATA_TYPES.SLOT_ID*DATA_TYPES.SLOT_ID'),
  C('<operation>SA.CREATE_SLOT:DATA_TYPES.COLUMN_POSITION*
                                      DATA_TYPES.ROW_POSITION
                                    ->DATA_TYPES.A_SLOT'),
  C('<operation>SA.CREATE_SA'),
  C('<operation>SA.INIT_SA'),
  C('<operation>SA.SET_SLOT:DATA_TYPES.SLOT_ID*DATA_TYPES.CONTENT
                                    ->DATA_TYPES.VALIDITY'),
  C('<data>SA.STACK'),
  C('<data>SA.SLOTS'),
  C('<constant>SA.FLOAT:->DATA_TYPES.SLOT_TYPE'),
  C('<constant>SA.EXPRESSION:->DATA_TYPES.SLOT_TYPE'),
  C('<constant>SA.COMMENT:->DATA_TYPES.SLOT_TYPE'),
```

```
    C('<constant>SA.BLANK:->DATA_TYPES.SLOT_TYPE'),
  %],
  [%
    C('<operation>WAE.STORE_MACRO:$STANDARD.INTEGER*$STANDARD.STRING'),
    C('<operation>WAE.SET_CS:DATA_TYPES.SLOT_ID'),
    C('<operation>WAE.RECALL_ALL_MACROS:->DATA_TYPES.LIST_STRINGS'),
    C('<operation>WAE.IS_MACRO_NAME:$STANDARD.STRING->$STANDARD.BOOLEAN'),
    C('<operation>WAE.INIT_CS'),
    C('<operation>WAE.INIT_CL'),
    C('<operation>WAE.GET_CS:->DATA_TYPES.SLOT_ID'),
    C('<operation>WAE.EDITOR'),
    [%
      C('<operation>CL.INIT_CL'),
      C('<operation>CL.GET_CL:->$STANDARD.STRING'),
      C('<operation>CL.LOCATE_SOL'),
      C('<operation>CL.REPLACE_PERCENT:->DATA_TYPES.VALIDITY'),
      C('<operation>CL.DELETE_LINE'),
      C('<operation>CL.DELETE_CHAR_LEFT'),
      C('<operation>CL.DELETE_CHAR'),
      C('<operation>CL.INSERT_STRING:$STANDARD.STRING'),
      C('<operation>CL.INSERT_CHAR:$STANDARD.CHARACTER'),
      C('<operation>CL.CURSOR_RIGHT'),
      C('<operation>CL.CURSOR_LEFT'),
      C('<data>CL.LINE'),
      C('<data>CL.EOS'),
      C('<data>CL.CURSOR'),
      C('<constant>CL.MIN_LENGTH:->$STANDARD.INTEGER'),
      C('<constant>CL.MIN_CURSOR:->$STANDARD.INTEGER'),
      C('<constant>CL.MAX_LENGTH:->$STANDARD.INTEGER'),
      C('<constant>CL.MAX_CURSOR:->$STANDARD.INTEGER'),
    %],
    [%
      C('<operation>CP.EDITOR'),
      [%
        C('<operation>EDITOR.TERM_CP'),
        C('<operation>EDITOR.INIT_CP'),
        C('<operation>EDITOR.IS_ESCAPE:$STANDARD.CHARACTER
                                    ->$STANDARD.BOOLEAN'),
        C('<operation>EDITOR.IS_PRINTABLE:$STANDARD.CHARACTER
                                    ->$STANDARD.BOOLEAN'),
        C('<operation>EDITOR.GET_CHAR:->$STANDARD.CHARACTER'),
        C('<operation>EDITOR.EDITOR'),
        C('<data>EDITOR.CHANNEL'),
      %],
      [%
        C('<operation>CCP.CCP_RECALL_MACRO'),
        C('<operation>CCP.CCP_STORE_MACRO'),
        C('<operation>CCP.CCP_CLI_KEEP'),
        C('<operation>CCP.CCP_CLI'),
        C('<operation>CCP.CCP_REPLACE_PERCENT'),
        C('<operation>CCP.CCP_ADJUST'),
        C('<operation>CCP.CCP_GET_CS_CONTENT'),
        C('<operation>CCP.CCP_GET_CS'),
        C('<operation>CCP.CCP_DELETE_LINE'),
        C('<operation>CCP.CCP_DELETE_CHAR_LEFT'),
        C('<operation>CCP.CCP_DELETE_CHAR'),
        C('<operation>CCP.REPLACE_CL:$STANDARD.STRING'),
        C('<operation>CCP.OBEY_CL:->DATA_TYPES.VALIDITY'),
```

```
        C('<operation>CCP.SAVE_CL'),
        C('<operation>CCP.PROCESS_CONTROL_CHAR:$STANDARD.CHARACTER'),
        C('<operation>CCP.IS_CONTROL_CHAR:$STANDARD.CHARACTER
                                    ->$STANDARD.BOOLEAN'),
        C('<constant>CCP.DESPATCH_TABLE:->POP_11.PROPERTY_TABLE'),
      %],
    %],
    [%
      C('<operation>CS.INIT_CS'),
      C('<operation>CS.GET_CS_CONTENT:->DATA_TYPES.CONTENT'),
      C('<operation>CS.SET_CS:DATA_TYPES.SLOT_ID'),
      C('<operation>CS.GET_CS:->DATA_TYPES.SLOT_ID'),
      C('<operation>CS.MOVE_CS_RIGHT'),
      C('<operation>CS.MOVE_CS_LEFT'),
      C('<operation>CS.MOVE_CS_DOWN'),
      C('<operation>CS.MOVE_CS_UP'),
      C('<data>CS.CURRENT_SLOT'),
    %],
    [%
      C('<operation>IL.INIT_IL'),
      C('<operation>IL.IS_MACRO_NAME:$STANDARD.STRING->$STANDARD.BOOLEAN'),
      C('<operation>IL.RECALL_ALL_MACROS:->DATA_TYPES.LIST_STRINGS'),
      C('<operation>IL.RECALL_MACRO:$STANDARD.INTEGER->$STANDARD.STRING'),
      C('<operation>IL.STORE_MACRO:$STANDARD.INTEGER*$STANDARD.STRING'),
      C('<constant>IL.OLD_CL:->$STANDARD.INTEGER'),
      C('<data>IL.MACROS'),
      C('<constant>IL.MIN_MACRO:->$STANDARD.INTEGER'),
      C('<constant>IL.MAX_MACRO:->$STANDARD.INTEGER'),
    %],
  %],
%],
%]
```

B.3.2 Final *TriviCalc* design Module Structure

Below is the final module structure for the *TriviCalc* problem, proposed by *Morpheus*.

```
[%
  [%
    [%
      [%
        [%
          [%
            [%
              [%
                C('<constant>DATA_TYPES.MAX_FLOAT:->$STANDARD.FLOAT'),
                C('<constant>DATA_TYPES.MIN_FLOAT:->$STANDARD.FLOAT'),
                C('<operation>SA.IS_FLOAT:$STANDARD.STRING
                                          ->$STANDARD.BOOLEAN'),
              %],
              [%
                C('<constant>DATA_TYPES.MAX_S_STRING:->$STANDARD.INTEGER'),
                C('<operation>SA.IS_COMMENT:$STANDARD.STRING
                                          ->$STANDARD.BOOLEAN'),
              %],
              C('<constant>DATA_TYPES.MAX_LETTER:->$STANDARD.CHARACTER'),
              C('<constant>DATA_TYPES.MAX_ROW:->$STANDARD.INTEGER'),
              C('<constant>DATA_TYPES.MIN_LETTER:->$STANDARD.CHARACTER'),
              C('<constant>DATA_TYPES.MIN_ROW:->$STANDARD.INTEGER'),
              C('<constant>DATA_TYPES.NIL:->POP_11.LIST'),
              C('<constant>DATA_TYPES.ZERO_FLOAT:->$STANDARD.FLOAT'),
              C('<constant>SA.BLANK:->DATA_TYPES.SLOT_TYPE'),
              C('<constant>SA.COMMENT:->DATA_TYPES.SLOT_TYPE'),
              C('<constant>SA.EXPRESSION:->DATA_TYPES.SLOT_TYPE'),
              C('<constant>SA.FLOAT:->DATA_TYPES.SLOT_TYPE'),
              C('<operation>SA.CREATE_SA'),
              C('<operation>SA.CREATE_SLOT:DATA_TYPES.COLUMN_POSITION*
                                             DATA_TYPES.ROW_POSITION
                                          ->DATA_TYPES.A_SLOT'),
              C('<operation>SA.DISPLAY_VALUE:DATA_TYPES.SLOT_ID'),
              C('<operation>SA.EVALUATE:DATA_TYPES.SLOT_ID
                                          ->DATA_TYPES.FULL_VALUE'),
              C('<operation>SA.INIT_SA'),
              C('<operation>SA.IS_SLOT_ARITHMETIC:DATA_TYPES.SLOT_ID
                                          ->$STANDARD.BOOLEAN'),
              C('<type>DATA_TYPES.COLUMN_POSITION'),
              C('<type>DATA_TYPES.CONTENT'),
              C('<type>DATA_TYPES.FULL_VALUE'),
              C('<type>DATA_TYPES.OPERATION_MATH'),
              C('<type>DATA_TYPES.ROW_POSITION'),
              C('<type>DATA_TYPES.SLOT_ID'),
              C('<type>DATA_TYPES.SLOT_TYPE'),
            %],
            [%
              C('<data>SA.SLOTS'),
              C('<data>SA.STACK'),
              C('<operation>SA.ADDRESS:DATA_TYPES.SLOT_ID
                                          ->DATA_TYPES.SLOTS_INDEX'),
              C('<operation>SA.ADD_SUCCESSOR:DATA_TYPES.SLOT_ID*
                                            DATA_TYPES.SLOT_ID'),
              C('<operation>SA.BLANK:DATA_TYPES.SLOT_ID'),
              C('<operation>SA.COMPLETE_UPDATE:DATA_TYPES.SLOT_ID*
```

```
                                  $STANDARD.BOOLEAN'),
      C('<operation>SA.DEPTH_FIRST_SEARCH:DATA_TYPES.SLOT_ID
                                  ->DATA_TYPES.SLOT_ID'),
      C('<operation>SA.GET_CONTENTS:DATA_TYPES.SLOT_ID
                                  ->DATA_TYPES.CONTENT'),
      C('<operation>SA.GET_VALUE:DATA_TYPES.SLOT_ID*$STANDARD.BOOLEAN
                                  ->DATA_TYPES.VALUE'),
      C('<operation>SA.IS_SLOT_BLANK:DATA_TYPES.SLOT_ID
                                  ->$STANDARD.BOOLEAN'),
      C('<operation>SA.IS_SLOT_COMMENT:DATA_TYPES.SLOT_ID
                                  ->$STANDARD.BOOLEAN'),
      C('<operation>SA.IS_SLOT_EXPRESSION:DATA_TYPES.SLOT_ID
                                  ->$STANDARD.BOOLEAN'),
      C('<operation>SA.IS_SLOT_FLOAT:DATA_TYPES.SLOT_ID
                                  ->$STANDARD.BOOLEAN'),
      C('<operation>SA.LIST_SUCCESSORS:DATA_TYPES.SLOT_ID
                                  ->DATA_TYPES.LIST_SLOT_IDS'),
      C('<operation>SA.REMOVE_SUCCESSOR:DATA_TYPES.SLOT_ID*
                                    DATA_TYPES.SLOT_ID'),
      C('<operation>SA.UPDATE_ORDER:DATA_TYPES.SLOT_ID
                                  ->DATA_TYPES.LIST_SLOT_IDS'),
      C('<type>DATA_TYPES.A_SLOT'),
      C('<type>DATA_TYPES.LIST_SLOT_IDS'),
      C('<type>DATA_TYPES.SLOTS_INDEX'),
      C('<type>DATA_TYPES.SLOT_ARRAY'),
      C('<type>DATA_TYPES.VALUE'),
    %],
    C('<operation>SA.IS_SUCCESSOR:DATA_TYPES.SLOT_ID*
                                    DATA_TYPES.SLOT_ID
                                  ->$STANDARD.BOOLEAN'),
  %],
  C('<operation>SA.SAVE_SA:->DATA_TYPES.LIST_STRINGS'),
  C('<operation>SA.SET_SLOT:DATA_TYPES.SLOT_ID*DATA_TYPES.CONTENT
                                  ->DATA_TYPES.VALIDITY'),
  C('<operation>SA.UPDATE_SLOTS:DATA_TYPES.LIST_SLOT_IDS
                                  ->DATA_TYPES.VALIDITY'),
%],
[%
  [%
    [%
      C('<operation>SA.IS_OPERATION:$STANDARD.STRING
                                  ->$STANDARD.BOOLEAN'),
      C('<type>DATA_TYPES.OPERATION_TEXT'),
    %],
    C('<operation>CLI.QUIT:DATA_TYPES.LIST_STRINGS->DATA_TYPES.VALIDITY'),
    C('<operation>CLI.SET_CURRENT_SLOT:DATA_TYPES.LIST_STRINGS
                                  ->DATA_TYPES.VALIDITY'),
    C('<operation>CLI.STORE_EXPRESSION:DATA_TYPES.LIST_STRINGS
                                  ->DATA_TYPES.VALIDITY'),
    C('<type>DATA_TYPES.LIST_STRINGS'),
    C('<type>DATA_TYPES.OPERATION_FULL'),
    C('<type>DATA_TYPES.VALIDITY'),
  %],
  [%
    C('<constant>DATA_TYPES.MAX_COLUMN:->$STANDARD.INTEGER'),
    C('<constant>DATA_TYPES.MIN_COLUMN:->$STANDARD.INTEGER'),
    C('<operation>CLI.SET_BLANK:DATA_TYPES.LIST_STRINGS
                                  ->DATA_TYPES.VALIDITY'),
```

```
      C('<operation>CLI.STORE_COMMENT:DATA_TYPES.LIST_STRINGS
                                    ->DATA_TYPES.VALIDITY'),
      C('<operation>SA.IS_SLOT:$STANDARD.STRING->$STANDARD.BOOLEAN'),
      C('<type>DATA_TYPES.EXPRESSION'),
      C('<type>DATA_TYPES.FULL_SLOT_TYPE'),
      C('<type>DATA_TYPES.SLOT_LETTER'),
      C('<type>DATA_TYPES.SLOT_NUMBER'),
      C('<type>DATA_TYPES.S_STRING'),
    %],
    C('<data>CLI.LOAD_IN_PROGRESS'),
    C('<operation>CLI.COMMAND_DESPATCHER:$STANDARD.STRING
                                    ->DATA_TYPES.VALIDITY'),
    C('<operation>CLI.FULL_FILE_NAME:$STANDARD.STRING->$STANDARD.STRING'),
    C('<operation>CLI.IS_FILE_NAME:$STANDARD.STRING->$STANDARD.BOOLEAN'),
    C('<operation>CLI.LOAD_FILE:DATA_TYPES.LIST_STRINGS
                                    ->DATA_TYPES.VALIDITY'),
    C('<operation>CLI.STORE_VALUE:DATA_TYPES.LIST_STRINGS
                                    ->DATA_TYPES.VALIDITY'),
  %],
%],
[%
  C('<constant>CCP.DESPATCH_TABLE:->POP_11.PROPERTY_TABLE'),
  C('<operation>CCP.CCP_CLI'),
  C('<operation>CCP.CCP_CLI_KEEP'),
  C('<operation>CCP.CCP_DELETE_CHAR'),
  C('<operation>CCP.CCP_DELETE_CHAR_LEFT'),
  C('<operation>CCP.CCP_DELETE_LINE'),
  C('<operation>CCP.CCP_GET_CS'),
  C('<operation>CCP.CCP_GET_CS_CONTENT'),
  C('<operation>CCP.CCP_REPLACE_PERCENT'),
  C('<operation>CCP.OBEY_CL:->DATA_TYPES.VALIDITY'),
  C('<operation>CCP.PROCESS_CONTROL_CHAR:$STANDARD.CHARACTER'),
  C('<operation>CCP.SAVE_CL'),
%],
[%
  C('<constant>CL.MAX_CURSOR:->$STANDARD.INTEGER'),
  C('<constant>CL.MAX_LENGTH:->$STANDARD.INTEGER'),
  C('<constant>CL.MIN_CURSOR:->$STANDARD.INTEGER'),
  C('<constant>CL.MIN_LENGTH:->$STANDARD.INTEGER'),
  C('<data>CL.CURSOR'),
  C('<data>CL.EOS'),
  C('<data>CL.LINE'),
  C('<operation>CCP.REPLACE_CL:$STANDARD.STRING'),
  C('<operation>CL.CURSOR_LEFT'),
  C('<operation>CL.CURSOR_RIGHT'),
  C('<operation>CL.DELETE_CHAR'),
  C('<operation>CL.DELETE_CHAR_LEFT'),
  C('<operation>CL.DELETE_LINE'),
  C('<operation>CL.GET_CL:->$STANDARD.STRING'),
  C('<operation>CL.INIT_CL'),
  C('<operation>CL.INSERT_CHAR:$STANDARD.CHARACTER'),
  C('<operation>CL.INSERT_STRING:$STANDARD.STRING'),
  C('<operation>CL.LOCATE_SOL'),
  C('<operation>CL.REPLACE_PERCENT:->DATA_TYPES.VALIDITY'),
  C('<operation>DM.POSITION_CL_CURSOR:DATA_TYPES.CURSOR_POSITION'),
  C('<operation>WAE.INIT_CL'),
  C('<type>DATA_TYPES.CURSOR_POSITION'),
%],
```

```
[%
  C('<constant>IL.MAX_MACRO:->$STANDARD.INTEGER'),
  C('<constant>IL.MIN_MACRO:->$STANDARD.INTEGER'),
  C('<constant>IL.OLD_CL:->$STANDARD.INTEGER'),
  C('<data>IL.MACROS'),
  C('<operation>CCP.CCP_RECALL_MACRO'),
  C('<operation>CCP.CCP_STORE_MACRO'),
  C('<operation>CLI.STORE_MACRO:DATA_TYPES.LIST_STRINGS
                                  ->DATA_TYPES.VALIDITY'),
  C('<operation>IL.INIT_IL'),
  C('<operation>IL.IS_MACRO_NAME:$STANDARD.STRING->$STANDARD.BOOLEAN'),
  C('<operation>IL.RECALL_ALL_MACROS:->DATA_TYPES.LIST_STRINGS'),
  C('<operation>IL.RECALL_MACRO:$STANDARD.INTEGER->$STANDARD.STRING'),
  C('<operation>IL.STORE_MACRO:$STANDARD.INTEGER*$STANDARD.STRING'),
  C('<operation>WAE.IS_MACRO_NAME:$STANDARD.STRING->$STANDARD.BOOLEAN'),
  C('<operation>WAE.STORE_MACRO:$STANDARD.INTEGER*$STANDARD.STRING'),
%],
[%
  C('<data>CS.CURRENT_SLOT'),
  C('<operation>CCP.CCP_ADJUST'),
  C('<operation>CS.GET_CS:->DATA_TYPES.SLOT_ID'),
  C('<operation>CS.GET_CS_CONTENT:->DATA_TYPES.CONTENT'),
  C('<operation>CS.INIT_CS'),
  C('<operation>CS.MOVE_CS_DOWN'),
  C('<operation>CS.MOVE_CS_LEFT'),
  C('<operation>CS.MOVE_CS_RIGHT'),
  C('<operation>CS.MOVE_CS_UP'),
  C('<operation>CS.SET_CS:DATA_TYPES.SLOT_ID'),
  C('<operation>DM.MOVE_CS:DATA_TYPES.SLOT_ID*DATA_TYPES.SLOT_ID'),
  C('<operation>WAE.GET_CS:->DATA_TYPES.SLOT_ID'),
  C('<operation>WAE.INIT_CS'),
  C('<operation>WAE.SET_CS:DATA_TYPES.SLOT_ID'),
%],
[%
  C('<data>DM.VDU'),
  C('<operation>DM.CLEAR_ERROR_DISPLAY'),
  C('<operation>DM.CLEAR_INVERSE_VIDEO:DATA_TYPES.SLOT_ID'),
  C('<operation>DM.DELETE_CHAR'),
  C('<operation>DM.DELETE_CHAR_AT_LEFT'),
  C('<operation>DM.DELETE_LINE'),
  C('<operation>DM.DISPLAY_CL_LINE:$STANDARD.STRING'),
  C('<operation>DM.DISPLAY_CONTENT:DATA_TYPES.CONTENT'),
  C('<operation>DM.DISPLAY_ERROR_MESSAGE:$STANDARD.STRING'),
  C('<operation>DM.DISPLAY_VALUE:DATA_TYPES.SLOT_ID'),
  C('<operation>DM.INIT_DM'),
  C('<operation>DM.INSERT_CHAR:$STANDARD.CHARACTER'),
  C('<operation>DM.INSERT_STRING:$STANDARD.STRING'),
  C('<operation>DM.LOCATE_SLOT:DATA_TYPES.SLOT_ID'),
  C('<operation>DM.RESET_DM'),
  C('<operation>DM.RING_BELL'),
  C('<operation>DM.SET_CS:DATA_TYPES.SLOT_ID'),
  C('<operation>DM.SET_INVERSE_VIDEO:DATA_TYPES.SLOT_ID'),
  C('<type>DM.VDU'),
%],
[%
  C('<data>EM.DISPLAY_IN_USE'),
  C('<data>EM.ERROR_QUEUE'),
  C('<operation>EM.ADD_TO_QUEUE:$STANDARD.STRING'),
```

```
        C('<operation>EM.DISPLAY_ERROR_MESSAGE:$STANDARD.STRING'),
        C('<operation>EM.ESCAPE_SEEN'),
        C('<operation>EM.INIT_EM'),
        C('<operation>EM.REMOVE_FROM_QUEUE:->$STANDARD.STRING'),
        C('<operation>EM.REPORT_ERROR:$STANDARD.STRING'),
      %],
    %],
    [%
      [%
        C('<data>EDITOR.CHANNEL'),
        C('<operation>EDITOR.GET_CHAR:->$STANDARD.CHARACTER'),
        C('<operation>EDITOR.INIT_CP'),
        C('<operation>EDITOR.TERM_CP'),
      %],
      C('<operation>CCP.IS_CONTROL_CHAR:$STANDARD.CHARACTER->$STANDARD.BOOLEAN'),
      C('<operation>CP.EDITOR'),
      C('<operation>EDITOR.EDITOR'),
      C('<operation>EDITOR.IS_ESCAPE:$STANDARD.CHARACTER->$STANDARD.BOOLEAN'),
      C('<operation>EDITOR.IS_PRINTABLE:$STANDARD.CHARACTER->$STANDARD.BOOLEAN'),
      C('<operation>WAE.EDITOR'),
    %],
    C('<operation>CLI.MAIN_PROGRAM'),
    C('<operation>CLI.REINITIALISE_SYSTEM'),
    C('<operation>CLI.SAVE_FILE:DATA_TYPES.LIST_STRINGS->DATA_TYPES.VALIDITY'),
    C('<operation>TRIVICALC.MAIN_PROGRAM'),
    C('<operation>WAE.RECALL_ALL_MACROS:->DATA_TYPES.LIST_STRINGS'),
  %],
  [%
    C('<constant>$STANDARD.FALSE:->$STANDARD.BOOLEAN'),
    C('<operation>$STANDARD.<<=>>:$STANDARD.STRING*$STANDARD.STRING
                                    ->$STANDARD.BOOLEAN'),
    C('<type>$STANDARD.BOOLEAN'),
    C('<type>$STANDARD.CHARACTER'),
    C('<type>$STANDARD.FLOAT'),
    C('<type>$STANDARD.INTEGER'),
    C('<type>$STANDARD.STRING'),
  %],
  [%
    C('<operation>POP_11.CLOSE_FILE:POP_11.CHANNEL'),
    C('<operation>POP_11.GET_INPUT_CHAR:POP_11.CHANNEL->$STANDARD.CHARACTER'),
    C('<operation>POP_11.ISSTRING:$STANDARD.STRING->$STANDARD.BOOLEAN'),
    C('<operation>POP_11.MATCHES:POP_11.LIST*POP_11.LIST->$STANDARD.BOOLEAN'),
    C('<operation>POP_11.OPEN:$STANDARD.STRING*$STANDARD.STRING
                                    ->POP_11.CHANNEL'),
    C('<operation>POP_11.PARSE_STRING:$STANDARD.STRING->POP_11.LIST'),
    C('<operation>POP_11.READ_LINE:POP_11.CHANNEL->$STANDARD.STRING'),
    C('<operation>POP_11.SYSEXIT'),
    C('<operation>POP_11.WRITE_LINE:POP_11.CHANNEL*$STANDARD.STRING'),
    C('<type>POP_11.CHANNEL'),
    C('<type>POP_11.LIST'),
    C('<type>POP_11.PROPERTY_TABLE'),
  %],
%]
```

Appendix C

Glossary and Abbreviations

Abstract Data Types	An object which encapsulates a type and its operations and operators, but without declaring a data item for the type. (Robinson, 1992a, p.227)
Abstract State Machine	Is a module that encapsulates the states of the object and provides operations to act on this state, or to direct the object to perform some actions. (Rosen, 1997, p.23)
ADT	Abstract Data Types, see also Abstract Data Types.
Architectural Design	The creation of a software model. The identification of components, their interactions with other components, and the hierarchically structure of the system.
Basic Entity	A design component, specifically excluding objects. In HOOD these consist of types, operations, constants, variables, operation_sets, and exception.
BNF	Backus Naur Form, a notation used to define the formal syntax of a language.
CASE	Computer Aided Software Engineering.
Closed System	A system is said to be closed, if there are no references in the system to entities not defined in the system.
Cohesion	The 'relatedness' of a set of basic entities in an object. Also, 'singleness' of purpose of an object. High cohesion is desirable.
Complexity	Intuitively, complexity measures the difficulty of understanding something. Unfortunately, this idea is not suitable for this thesis, since it involves a study of psychological phenomena which are outside the scope of this work. We therefore define complexity as the 'difficulty' of describing the architecture of a software design.
Component Design	takes each component identified during architectural design and determines how it will work in detail.

Coupling	The dependency between two objects. High coupling indicates that a change to one object is likely to impact on another. Low cohesion is desirable.
Degree	The number of edges incident on a graph's node.
Digraphs	A graph, where the direction of the edges is significant. Also called directed graphs. All graphs in this thesis are digraphs.
DFD	Data Flow Diagram.
Encapsulation	The method of combining data and operations on those data in an object. (Robinson, 1992a, p.228)
Entity (design)	A design component, including objects. In HOOD these consist of objects, types, operations, constants, variables, operation_sets, and exception.
Environmental Object	An object which represents the provided interface of another object used by the system to be designed, but which is not part of the [current] HOOD design tree. (Robinson, 1992a, p.228)
ESA	European Space Agency.
Forest	A set of trees, usually implying at least two disjoint trees.
Generic	An object template to represent a reusable object with type, constant and operation parameters.
Graph	A graph $G(\mathcal{N}, \mathcal{E})$ consists of a set of finite nodes $\mathcal{N}$ and a set of edges $\mathcal{E}$ over $\mathcal{N} \times \mathcal{N}$. The existence of a particular edge (n_i, n_j) implies that there is a relationship between the two nodes n_i and n_j. In this thesis, we are only concerted with *directed graphs* (or *digraphs*), in which case the order of the pair (n_i, n_j) is significant. For brevity, we often use the term *graph* to refer to a directed graph.
HOOD	Hierarchical Object Oriented Design.
HRM	HOOD Reference Manual.
HUG	HOOD User Group.
HUM	HOOD User Manual.
Information Hiding	Information hiding supplements encapsulation by preventing an entity, usually data, from being visible to other software, by declaring it in the internals of an object rather than in the interface (e.g., declaration only in the body of an Ada package and not in the specification, or as a private or protected declaration in C++), or by hiding implementation details of an entity, e.g., as a private type. (Robinson, 1992a, p.228)
Inheritance	The ability to define a class which is an extension of an existing class (called a base class), so that the new class *inherits* all the attributes (data and types) and all the actions (operations) of the base class. (Robinson, 1992a, p.228)

Inter-	Prefix meaning among, between, together, one with another, etc. (Bancroft, 1969, p.181)
Interface	Specification of the usable (visible) part of an object.
Internals (HOOD)	The *hidden* details of an object's implementation.
Intra-	Prefix meaning within, inside. (Bancroft, 1969, p.182)
Isomorphic	Having the same-shape. Technically, there exists a one-one relation between the two isomorphic items.
Morpheus	was a dream-god who caused human shapes to appear to dreamers. In spite of popular associations with sleep itself ('safe in the arms of Morpheus' and so on) his name actually means 'form' (*morphe*), the reference being to the 'forms' or shapes seen in dreams. In a sense, therefore, he is really a 'transformer'. (Room, 1990, p.208)
MDL	Minimum Description Length.
Object	An encapsulation of data or a hardware interface with the operations to access and change it, with relevant type definitions, constants and exceptions. (Robinson, 1992a, p.228)
Object-based	A weaker (older) form of OOD, lacking facilities such as inheritance.
Object Oriented	The essence of an object-oriented method is the identification and development of a design using an object as the basic building block of the design. The term 'object-oriented' is often taken to imply development of classes as well as objects, so that the executable design consists of objects as instances of these classes. (Robinson, 1992a, p.229)
ODS	Object Description Skeleton. The formal textual notation of the design on object.
OO	Object Oriented, see also Object Oriented.
OOD	Object Oriented Design, see also Object Oriented.
Open System	A system is said to be open, if it is not closed.
Operation	An action that is performed on an object, to be represented by a procedure or function in Ada. (Robinson, 1992a, p.229)
Operation_Sets	A shorthand form for a set of operations which precludes the necessarity to write long lists of operations in a HOOD diagram. (Robinson, 1992a, p.229)
Ordinal Scale	A scale with at least a partial order.
OS	Operating System.

Overloading	The ability for an operation name to be repeated with the definition of a single object, providing that there is some way of differentiating between them, i.e., by having different parameter and result type profiles in Ada, or different argument signature in C++. (Robinson, 1992a, p.229)
Polymorphism	The ability for the selection of an operation body to be determined at run time, according to the class of the object to which the operation is currently referring. (Robinson, 1992a, p.229)
Provided Interface	Defines the services that an object provides to its clients.
Ratio Scales	A scale with a total ordering permitting statements such as "*A* is twice as big as *B*" to be meaningful.
Requires	An edge in a design graph represents a requires relationship, i.e., an entity requires the services of another entity in oder for the first to provide its services to others.
Required Interface	Identifies the services required from other objects.
Root	The 'top' of a tree. It is the unique item in a tree having no parent.
Root Object	The top-level object which represents the system to be designed. (Robinson, 1992a, p.229)
SIF	Standard Interchange Format. The precise textual format used for exchanging ODSes between different HOOD toolsets.
Tree	A tree is a special kind of graph, which has no cycles.
TriviCalc	A small spreadsheet used as a design study, and for empirical validation of *Morpheus*.

Appendix D

Notation Summary

$\varnothing$	The empty set.
$a \in A$	a is a member of the set A.
$A = B$	Set equality.
$\|A\|$	The cardinality of the set A.
$A \cup B$	Set union.
$A \cap B$	Set intersection.
$A \subseteq B$	Subset.
$A \subset B$	Proper subset.
$\mathcal{N}$	The finite set of nodes of a graph $G(\mathcal{N}, \mathcal{E})$.
$\mathcal{E}$	The finite set of edges of a graph $G(\mathcal{N}, \mathcal{E})$.
$\mathcal{E}^*$	The finite bag of edges of a multi-graph $G^*(\mathcal{N}, \mathcal{E}^*)$.
d_i	The degree of node n_i in a graph $G(\mathcal{N}, \mathcal{E})$.
D	The sum of the degree of all nodes in a graph $G(\mathcal{N}, \mathcal{E})$, $D = \sum_{i \in \mathcal{N}} d_i$.
$G(\mathcal{N}, \mathcal{E})$	A graph G with a set of nodes $\mathcal{N}$, and a set of edges $\mathcal{E}$.
$G^*(\mathcal{N}, \mathcal{E}^*)$	A multi-graph G^* with a set of nodes $\mathcal{N}$, and a finite bag of edges $\mathcal{E}^*$.
$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubseteq G_2(\mathcal{N}_2, \mathcal{E}_2)$	$G_1(\mathcal{N}_1, \mathcal{E}_1)$ is a subgraph of $G_2(\mathcal{N}_2, \mathcal{E}_2)$.
$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqsubset G_2(\mathcal{N}_2, \mathcal{E}_2)$	$G_1(\mathcal{N}_1, \mathcal{E}_1)$ is a proper subgraph of $G_2(\mathcal{N}_2, \mathcal{E}_2)$.
$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcup G_2(\mathcal{N}_2, \mathcal{E}_2)$	Graph union.
$G_1(\mathcal{N}_1, \mathcal{E}_1) \sqcap G_2(\mathcal{N}_2, \mathcal{E}_2)$	Graph intersection.
$A \circ B$	The concatenation of two designs A and B.
$\Pr(x)$	The probability of x occurring.

$\mathrm{KC}(x)$	The Kolmogorov Complexity of x.
$A \cong B$	Indicates an encoding, such that, B is an encoding of A.
$\mathcal{L}^*(n)$	An optimal universal prefix code for all positive integers. Each integer has an encoding of the form, $\log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + \cdots$, terminating when $\log(\ldots \log n) \leq 0$. See Section 5.2.3.
$\log_2^*(n)$	Length (in bits) of the $\mathcal{L}^*(n)$ function, given by $\mathcal{L}^*(n) + \log_2 2.865064$. See Section 5.2.3.
$\mathcal{S}$	Set of all possible design graphs.
$\Psi(G)$	The complexity of the design graph G. See Section 6.2.