Heuristics for Object-Oriented Design

by

Cleveland Augustine Gibbon

Thesis submitted to the University of Nottingham for
the degree of Doctor of Philosophy, October 1997

Abstract

Maintainability is an important property in software that enables systems to evolve in parallel with an organisation's needs. Yet software is still being implemented that is not amenable to change. Metrics are the primary means to assess the maintainability of software products but they target the experienced developer. Instead, hints, guidelines and tips are used by software engineers to communicate design experience from the expert to the novice. A means to augment the pragmatic appeal of the latter with the theoretical grounding of the former will create an alternative means of evaluation that is available to *all* those that participate in software development.

The thesis set out to bring techniques for building maintainable object-oriented (OO) software closer to the developer in the form of design heuristics. Heuristics document common design problems that *developers* encounter during software development. The heuristic catalogue provides a comprehensive reference point for both novice and expert developers to apply well-documented techniques for building maintainable software.

In light of the inter-dependencies that exists between heuristics and the complex nature of large object-oriented software systems, a prototypical system entitled TOAD was implemented to automate heuristic deployment. TOAD acts as a design aid for developers performing OO design (OOD). It is highly interactive, locating potential problems in OO software and presenting design alternatives that are applied at the user's discretion.

Design heuristics have been used for the last two years at the University of Nottingham. The success reported by the inaugural users of TOAD is the main driving force behind future research into design heuristics, their automation and the informal design evaluation process to which they both subscribe.

Acknowledgements

First and foremost I wish to thank my supervisor, Dr. Colin Higgins, for all his suggestions and encouragement throughout the course of this research, and for the editorial feedback during the preparation of this thesis.

To my fellow students and mentors at the OOPSLA'96 doctoral symposium I wish to extend my sincere gratitude. Your incisive comments and alternative perspectives on my research enabled me to focus more clearly upon the problems at hand.

To my friends and colleagues in the Department of Computer Science, I wish to extend my thanks for the assistance I received at the most unlikely of times and for the most unusual of requests. I thank you for your time and effort.

Finally, thank you to my friends in the architecture department whose quest for the ultimate building provided me with an understanding of *design* that lies beyond the realms of computer science.

Thank you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

To Juanita

Chapter 1

Introduction

How good is my design? An expert answering this question on behalf of a learner relies upon experience gained by performing numerous designs. As the ratio of learners to experts increases, innovative methods for *transferring* design experience from the latter to the former becomes increasingly more important; a problem faced by educators and trainers within academia and industry respectively.

Consider an organisation seeking an answer to this question for a large and complex software system. The expert must first *identify* the problems in the system before they can *communicate* them back to the organisation. For the lone expert, the system will be too large to comprehensively assimilate in its entirety. For groups of experts, communicating the problems amongst themselves *and* to the organisation is even more difficult without a vocabulary of common errors and anomalies arising within the software. In both cases, without tool support it is unlikely that a thorough evaluation of the software can take place.

How maintainable is my design? Supplanting *good* within *maintainable* provides a concrete example of the problems currently faced by the software engineering community. Software metrics are the primary means to address this question. However, the feedback that metrics deliver is restrictive and not comprehensible by all those participating in the design process. Metrics assume knowledge of the technology and of the low level design concepts that they are founded upon. For the majority of developers, those learning or migrating to object technology, these assumptions render metrics inappropriate for evaluating their software.

Maintainability is an important property in today's software. However, in the absence of a common vocabulary for communicating design problems, techniques for building maintainable software remains with the experts. Furthermore, without intelligent tools that can reason about the identified problems and offer insightful feedback, these techniques will be largely unused by those learning how to design or by those evaluating complex software systems. A means to express design expertise in a more *legible* fashion that is available to *all* those partaking in software development is the *principal* aim of this thesis.

1. Overview

Section 1.1 documents our objectives and aims for empowering developers with tools and techniques for implementing maintainable software systems. In section 1.2, this need to build maintainable software is put into perspective by the current mechanisms available to developers to achieve this goal. The requirements and perceived benefits of an alternative approach to design evaluation with heuristics is presented in section1.3. Section 1.4 outlines the object model that is used throughout this thesis, leaving the scope and organisation of this research to section1.5.

1.1 Objectives and aims

The aim of this research was to discover, invent, document, automate and evaluate a number of heuristics for building maintainable object-oriented designs (OOD). OOD heuristics encapsulate software problems *and* their solutions in supporting an informal approach to design evaluation. Our research deliverables are pragmatic and have been made available to those interested in constructing maintainable design architectures.

A number of searching questions set the rationale for this thesis. Are heuristics effective during software design? Can they be used to supplement more formal techniques? Are heuristics appropriate vehicles for documenting design problems? How can heuristics be objectively specified,

if at all? Can heuristics be automated within tools; effecting problem detection, providing immediate feedback and applying well-known solutions? What advantages, if any, do they have over software metrics?

To satisfy these aims a catalogue of design heuristics was created that comprehensively documents recurring problems within OO software. A prototypical system was developed entitled TOAD, or Teaching Object Analysis and Design, to automate the deployment of heuristics within this catalogue upon OOD documents. The design heuristic catalogue [Gi97b] and the TOAD system [Gi97c] are important project deliverables that supplement the research presented in this thesis.

Design heuristics were used by students taking the OO methods course at the University of Nottingham. They were also used to review a number of large OO systems. The results gained from students applying design heuristics during OOD are presented together with how heuristics scale up when evaluating large design architectures. Both types of analysis are important contributors to answering the questions set out in the thesis rationale.

1.2 Background and motivation

This section clarifies our view of maintainability and expounds its relationship with maintenance. In light of the well-documented importance placed upon maintainability [Le80][HW93][CK94] and the lack of techniques available to all developers for its improvement [GH97a], we highlight the background and motivation for delivering accessible and legible mechanisms for constructing maintainable software.

1.2.1 Maintainability and maintenance

Maintainability is mandatory and assumed to be resident within all delivered software products. The drive for maintainable systems is fuelled by the stark realisation that over 70% of software costs occur during maintenance [Ke87]. Lehman [Le80] estimated that the United States government and commercial institutions collectively spent 50-100 billion dollars on software in 1977, or

approximately 25-75 billion dollars on maintenance. The increased need, use and cost of software today has seen these figures rising, thereby demanding new and innovative mechanisms to reduce maintenance costs.

Once a system has been released, it enters a state of **maintenance**. A well-structured software architecture requires substantially less effort to maintain than a poorly structured equivalent. For (legacy) systems in a state of disrepair, the cost of maintenance can be exorbitant. However, the size of a software system's client base and/or the amount of effort previously invested, may be too great justify its expiration. Nowadays, to reduce maintenance costs in future software products, effort is expended earlier during the upstream development activities to ensure that released systems are maintainable. Clearly, the aim is to catch errors and omissions earlier rather than later. Nevertheless, maintainable *design* architectures are flexible enough to subsume these changes with minimal costs. This has resulted in a surge of software design research activities aimed at constructing maintainable systems.

Maintainability in software refers to the ability to evolve in the face of changing requirements as opposed to being *preserved*. In Figure 1—1, Lientz and Swanson [LZ79] apportion maintenance costs based on a survey of 487 sites involved in software development.

Kitchenham [Ki87] submits that the two dimensions of maintainability are *effort to debug* and *elapsed time to debug*. However, these interpretations of maintainability are for the treatment of defective software that is in a state of preservation. With only 22% of maintenance activities dedicated to corrections (see Figure 1—1), emphasis on this type of maintainability is secondary to that of enhancements requested by its users.

More preventative notions of maintainability espouse the benefits of containing software changes within flexible design architecture [Tu95]. This view of maintainability is concerned with structuring software in such a way that the addition, modification and removal of system functionality can be

made with minimal effort and with limited degrees of re-design. We adopt this view of maintainability that aims to implement software products that are *easier* to modify. The consequences of ignoring and/or being ignorant of issues pertaining to maintainability are being paid in full by a number of companies updating their (legacy) systems to be millennium compliant.

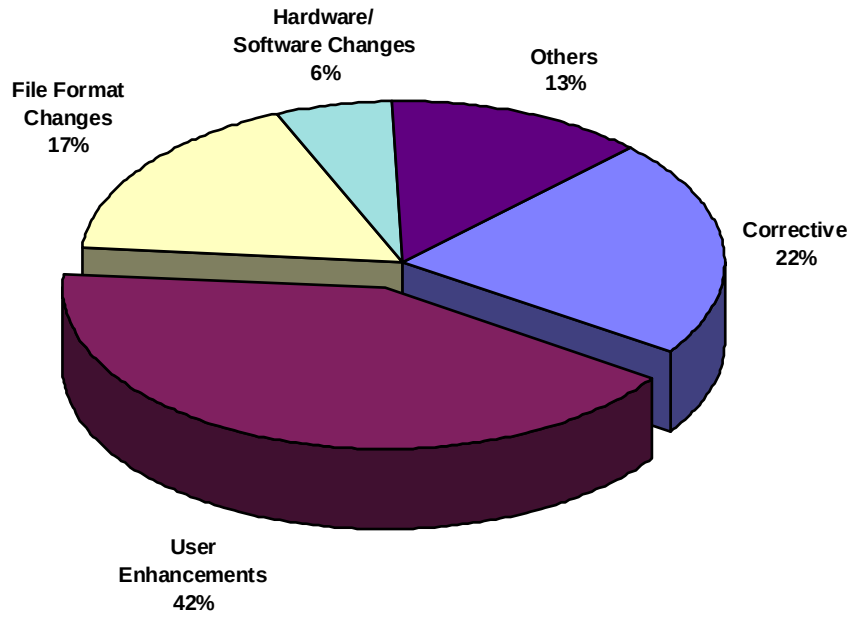


Figure 1—1 : Breakdown of maintenance costs

Summarising, the principal goal of maintainable software is to be amenable to change. However, maintenance over time, renders software unmaintainable as it increases in size and progressively loses its structure [LZ79]. This places even more importance upon building high degrees of maintainability into the original system. During the last two decades of software engineering, experience developing systems has shown software design to be the most (cost) effective place to evaluate and indoctrinate aspects of maintainability.

1.2.2 Design Maintainability

> *"We try to solve the problem by rushing through the design process so that enough time is left at the end of the project to uncover the errors that were made because we rushed through the design process" Myers[My76].*

In recent years this unfortunate cycle of events has caused emphasis to be placed upon the analysis and design phases of software development. During these early development phases, errors can be trapped before they manifest themselves into costly and often intractable problems. Figure 1—2 extrapolates data gleaned from Kendall [Ke87] to illustrate the relative costs involved in fixing an error or changing some functionality at the various phases of software development.

Clearly, the majority of incurred maintenance costs result from implementing user enhancements. The surveyed software managers also reported that maintenance sees systems growing in size with inadequate resources available to contain it. So, not only does the software need to be maintainable when it enters maintenance, but maintenance must preserve its maintainability.

Yau and Courfello [YC85] identified three approaches to controlling maintenance costs:

- improve the productivity of developers by providing essential tools needed to perform maintenance tasks (e.g. debuggers, editors, performance analysers, etc.);
- monitor the system development phase by employing metrics to control the process;
- supply tools and techniques early on in software development to guide designers towards building more maintainable software.

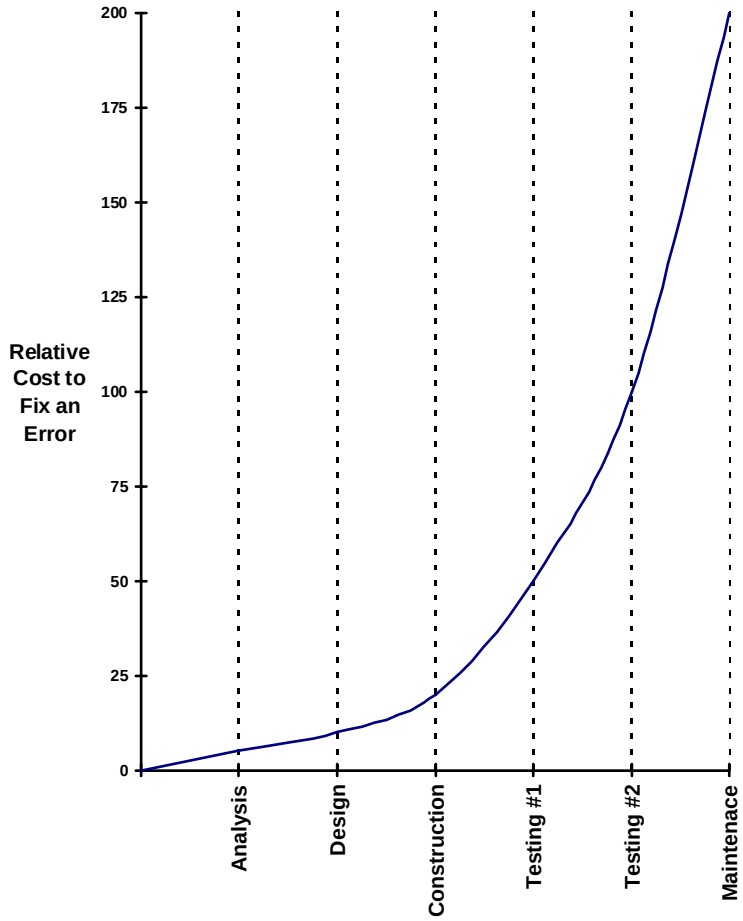


Figure 1—2 : The cost of fixing an error during software development

Regarding the first item, due to the complex nature of the development process, there will always be a need for sophisticated, post-operative tools to eradicate errors in software. The last two refer more directly to aspects of software measurement.

1.2.3 Software measurement and quality

Software engineering needs measurement to control and manage the methods used to build high quality software in a cost effective manner. However, it is this dichotomous marriage between *quality* and its *cost* that has brought measurement to the forefront of software development as a means to introduce the former, with limited amounts of the latter.

There are essentially two types of software metric: process and product. Process metrics aim to *control* software development by empowering managers with the means to plan, estimate and allocate project resources with varying degrees of confidence. Although, project planning plays an integral role in reducing maintenance costs, this research focuses upon enhancing the quality, more specifically maintainability, of a given software product regardless of how it was produced.

The increased and progressive application of design metrics by organisations wishing to enhance software quality and reduce maintenance costs represent the principal benefits of measurement. However, measurement has contributed much more to the field of software development than these industry-driven objectives portray. DeMarco [De79] astutely surmised that "…before measurement, there must be understanding." In the quest for measurement, software experts have discovered, invented, refined and documented *which* design concepts and their inter-relationships contribute to building well-structured systems. The consensus is that software with good internal structure, has good internal quality which in turn has good external quality. This is Fenton's axiom of software engineering that is well-documented and widely accepted within the measurement community, otherwise "…if this assumption is wrong then almost all software engineering research and development of the last 20 years has been worthless" [Fe91].

Three decades on from the first publication on software measurement [RH68], metrics remain the primary (only?) means by which software engineers can improve its maintainability. This shortage of alternative techniques for enhancing maintainability presents a number of problems.

The use of metrics within an organisation has been somewhat artificial in that they are typically used by those seeking to enhance the quality of software products that have been produced by *others*. Lorenz and Kidd [LK94] issued warnings on the detrimental effect that quality assurance personnel and management can have if they employ metrics to *police* and *punish* software

produced by their developers. However, these caveats lie secondary to the fact that if software metrics are not being used by the majority of developers, then the design knowledge that they aim to disseminate is also not being applied. A means to make metric research available to a wider audience is needed so that developers can apply this design experience to their own software products.

Developers require *degrees* of quality to be engineered into their products, especially in an era of reusable software abstractions. Developers can ill-afford to discard software because the return on investment from its reusable parts may far outweigh the effort invested in constructing the whole. The higher the quality of the original product, the less costly it will to be*locate, extract* and *catalogue* the resident reusable abstractions. Hence, software engineers must aim to *incrementally* build degrees of quality into *all* their products. This requires vehicles for quality enhancement that are small, self-contained and legible. This will permit gradual and controlled improvements to design architecture to take place. Furthermore, they must be available to *all* developers and for a *diverse* set of software.

1.2.4 Summary

Maintenance costs are high and systems are still being constructed that are *not* amenable to change.

Software metrics are used to ascertain whether a design is maintainable and rank products along a given quality scale. Metrics are not a solution and form only a part of the software engineering equation. However, the metric-centred view of quality has brought a clearer understanding of low-level design concepts, defined their inter-relationships and illustrated how they affect maintainability. Even so, metrics are not being used by all developers. In fact, the most common type of developer, the learner, is not introduced to metrics until much further on into their software education process. Hence, issues of quality are being neither effectively assimilated nor applied by those responsible for implementing today's software and tomorrow's legacy

systems. An alternative vehicle for disseminating aspects of quality is needed to supplement that of software metrics and bring design expertise a step closer to the novice developer.

1.3 The informal evaluation of software designs

This research demonstrates that an informal approach to design evaluation is effective and beneficial to both expert and novice designers during software development. This form of evaluation is intended to be integrated into current methods of teaching OOD to provide additional support to those applying object technology. Secondly, we wish bring the experience and knowledge gained from metric research to the fore. A means to link issues of quality to the problems commonly encountered by developers applying object technology would highlight why certain concepts, such as encapsulation, are important and how they are used to enforce maintainability. Finally, our research must deliver pragmatic solutions that can be used all those participating in OOD.

To this end, design heuristics are proposed as a more accessible and informal means by which developers can evaluate OO software. Software heuristics are small, simple, legible, self-contained nuggets of design expertise. They target specific design problems within OO software and provide guidance on how to effect a solution. Unlike metrics, heuristics are outwardly defined in terms of the observable problems that occur during OOD and not low level design concepts such as coupling, cohesion and size. Instead, the lessons learned from applying these design concepts throughout metric research provide a solid theoretical foundation that heuristics build upon to document recurring and observable problems within OO systems.

Design heuristics are: available to *all* developers performing OOD; applicable within a number of software domains; permit small, incremental enhancements to maintainability; provide a common vocabulary for expressing design problems.

This thesis defines what exactly constitutes a design heuristic. Design heuristics are subsequently created, documented and structured within a catalogue. A tool entitled TOAD was implemented to automate their deployment for developers evaluating OOD documents. Finally, we present the results of applying heuristics to OOD documents and determine whether they are useful during software development.

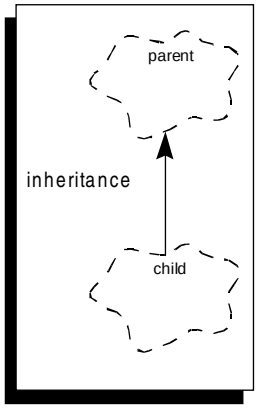
1.4 Object model definition

This section augments object models presented by Snyder [Sn87] and Booch [Bo94] in providing the thesis nomenclature for OOD elements and their relationships. The class-based components of our object model are presented and expressed using the Booch design notion.

OOD permits designers to define new classes of objects. Each **object** is an instance of one **class**. The internal state of an object is represented by a collection of **instance variables** as defined by the class. Each class defines a set of named **methods** that can be invoked on instances of that class. Methods are implemented by procedures that can access and assign to instance variables of the target object. At the analysis and design level the methods of a class are also collectively referred to as **responsibilities** or **behaviour**, and its instance variables as **attributes** or **state**.

Inheritance can be used to define a class in terms of one or more other existing classes. The class inherits the methods and the instance variables defined by existing classes, which it can augment with additional methods and instance variables.

Aggregation denotes those instance variables which have types that are user-defined classes in the design, whereas the **using** relationship stipulates those user-defined classes that appear in the method signatures of a class definition.

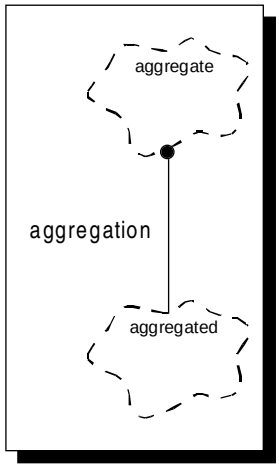


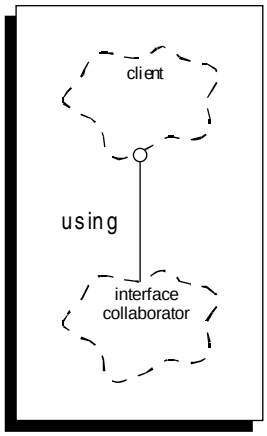
Inheritance:

If a class c directly inherits from a class p, we say that p is a *parent* of c and that c is a *child* of p. The terms *ascendent* and *descendent* are used in the obvious way.

Aggregation:

For all instance variables in a class c, that are objects of a user-defined class, we say c is the *aggregate* (or whole) and the objects are the *aggregated* (or parts).





Using:

For all user-defined classes u that appear in the method signatures of a class c, we say that c is the *client* and that u are *interface collaborators* .

1.5 Scope and organisation of thesis

This thesis both addresses the problems surrounding maintainability in software designs and the way that their delivered solutions can be applied in a pragmatic manner. Upon reviewing how software metrics assess design quality in chapter 2, a comprehensive understanding of the low-level design

concepts and how they have been employed during software development is gained. Chapter 3 expounds the design evaluation problem. Design heuristics are proposed as an informal alternative to software metrics that appeal to a wider audience and aim to*educate* developers in aspects of quality through *enhancement.* Enhancement is seen as a means to an end where the end is education.

Chapter 4 presents the objectives, structure and models employed by the design heuristic catalogue. It documents the resident design heuristics and illustrates how the majority of them can be objectively specified as part of an informal design evaluation process. Chapter 5 automates the deployment of design heuristics within TOAD; a tool to evaluate OOD documents. The results from applying TOAD during software development and preliminary conclusions on the use of design heuristics are reported in chapter 6. Finally, we place our research into perspective in chapter 7 by examining whether we have met our objectives. We look towards the future for ways to extend the work presented in this thesis.

Chapter 2

A review of software design metrics

2. Overview

A plethora of software product metrics has been published in the literature. Publications reviewing, comparing and summarising these works abound [Wh96][Zu96][Co82][Fe91][He96][Sh88][BC94][CG90][In91]. It is not the intention of this chapter to re-review these summaries but to focus upon the role design metrics play in software measurement. More specifically, this chapter identifies which design concepts are important for constructing maintainable software designs and how they have been quantified using software metrics.

This chapter commences with a brief introduction to the software complexity problem. It documents how and which low level concepts have been employed to reduce design complexity and the metrics charged with enforcing them. We describe the relative contributions of these design concepts for improving software products and report on what the published measurement community believes are the better metrics and why. Based upon these findings, chapter 3 illustrates why an alternative technique than that provided by software metrics is needed for design evaluation. Chapter 4 then builds upon the metric research presented in this chapter to implement a catalogue of design heuristics for evaluating OOD documents.

2.1 Background

2.1.1 Software complexity

Chen and Lu [CL93] contend that "...reducing software complexity will definitely improve software quality." However, "...the first problem

encountered when attempting to understand program complexity is to define what it means for a program to be complex" [HMK+82]. This now widely accepted statue that the majority of software's complexity resides in the design [CG90], means that we should address the problem of defining what exactly constitutes (design) complexity.

Complexity is an aggregate that is captured by the totality of all internal product attributes [Fe91]. These internal product attributes pertain to low level concepts such as coupling, cohesion and size. As an aggregate, complexity measures should not only gauge the presence of internal product attributes but also how they inter-relate. Wohlin [Wo96] argues that metric researchers have often misconstrued a single internal product attribute as complexity itself. This leads to incomplete complexity measures that are neither good indicators of complexity nor appropriate for representing complexity as a single, all-encompassing figure [Fe91]. Before measurement can take place, these internal product attributes need to be clearly defined and their relative contributions to the complexity problem acknowledged.

A number of important factors contribute to the production of useable software metrics. These include defining appropriate scales along which internal product attributes are measured [Zu92][Av96], the quality and volume of data needed to lend credence to the results of metric validation [We96] and tools to transparently collect and analyse project data [Br96]. Nevertheless, this chapter focuses directly upon how software product metrics have:

- defined design complexity in terms of internal product attributes;
- employed techniques to identify their presence or absence in software (design) products;
- used objective measures to quantify internal product attributes;
- provided a partial ordering on internal product attributes based upon their contribution to the complexity problem.

2.1.2 Software design complexity

Stevens et al [SMC74] highlighted five properties that high quality software designs should possess: coupling, cohesiveness, modularity, size and complexity. Yourdon and Constantine [YC79] expounded these internal product attributes which were subsequently quantified by Troy and Zweben in [TZ81]. It was not surprising to note that the majority of Troy and Zweben's complexity-based metrics could be re-defined in terms of the other four internal product attributes. This early concrete example of the confusion surrounding complexity further endorsed Fenton's preliminary observations that complexity is multifaceted. From these initial findings by Troy and Zweben, coupling, cohesion, modularity and size were seen as the principal facets of *design complexity* .

2.1.3 Software design metrics

Traditional metrics is the collective term for measures implemented within the structured design era. They encompass both **structure metrics** that evaluate the internal product attributes coupling, cohesion and modularity, and **code metrics** that measure aspects of program size. Early traditional metrics were predominantly code-based and perceived complexity as a function of size, or any one of its variants, namely length and volume. However, the inadequate nature of code metrics as indicators of complexity became more evident as programs evolved into systems and as monolithic software became modular.

In modular software, coupling is a measure of the strength of relationships between modules and cohesion the strength of dependencies inside a module [St81]. Modular software aims to maximise the independence of modules by enforcing low coupling *and* asserting that each module performs a single, identifiable task by implementing high cohesion [Em84]. Modularity is a property of software in which modules are fragmented in such a way that the resultant system is both decomposable and composable [Me88]. Structure

metrics assess modularity in software, but unlike code metrics, aim to be applied to *design* products.

Figure 2—1 illustrates modularity in a software system. The following questions represent just a few problems that structure metrics must overcome when evaluating modularity in large software architectures:

- when should the modules in cluster A forsake low coupling in order for A to be highly cohesive;
- given the dependency between clusters A and B is the result of services imported by A from a single highly cohesive module in B, on what grounds should A and B coalesce, be re-factored or the dependency between them left intact;
- what is the unit of modularity; the module, cluster or cluster of clusters?

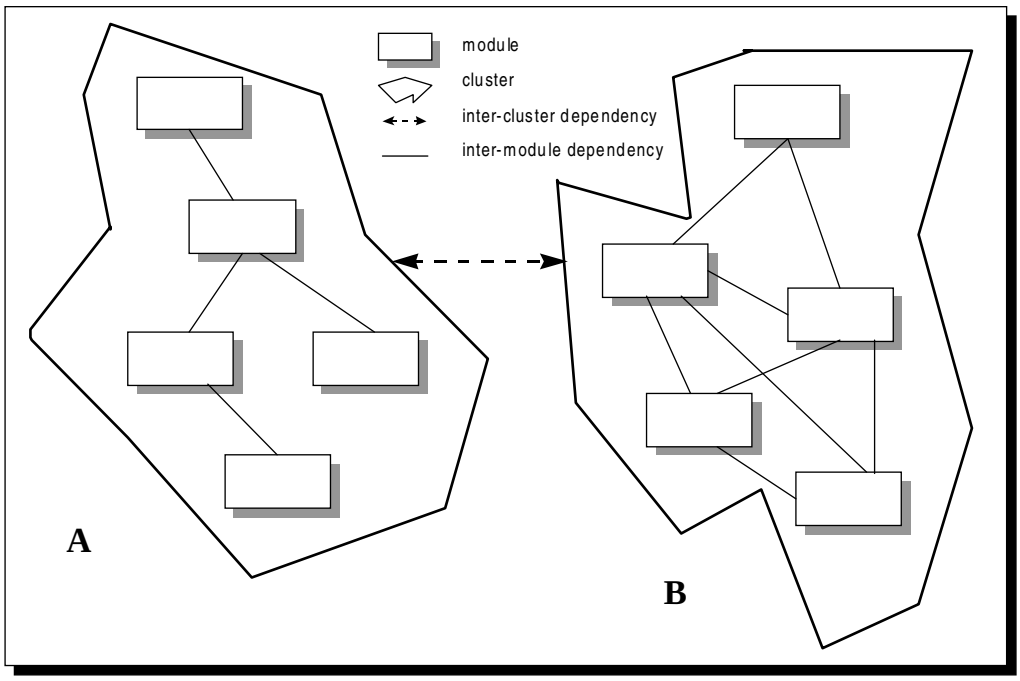


Figure 2—1 : Modularity in software

Despite the vast number of software metrics in the literature, few are original. The majority of metrics are variations on themes that can essentially be categorised as either structure, code or hybrid metrics [HS90][SCD83]. Hybrid metrics cater for the relationships that exist between structure and code metrics [HKH81]. Typically, hybrid metrics take on a two-phase process that first applies code metrics to order software on the basis of size and then deploys structure metrics to determine architectural design complexities. The reasoning behind hybrid metrics is that code and structure metrics complement one another providing a more balanced platform for measurement between disproportionate software.

2.1.4 Summary

Modularity lies at the centre of design complexity and therefore design measurement. The absence of this property in software restricts its potential for supporting a de-centralised and maintainable design architecture. The remainder of the chapter takes a more in-depth look at software design metrics by first exploring in section 2.2, what has become known amongst developers as the founding (classical) metrics for software measurement. An investigation into traditional metrics given in section2.3 describes how they have been used to quantify design complexity. The role of traditional metrics in assessing OO software is then illustrated in section 2.4.1 and the advantages and limitations of this approach discussed.

The remaining parts of section2.4 describe why the second generation of OO metrics dispense with traditional metrics when evaluating OO software. The conclusions drawn in section 2.5 provide the foundation upon which this thesis adopts an alternative approach to design evaluation than that offered by software metrics. An outline of the resultant research plan is given in section 2.6.

Although the types of metric, the products they act upon and the process by which they are evaluated are constantly changing, the fundamental concepts upon which design metrics are based remain steadfast. As this chapter

illustrates, metric research has served to define and further establish these important design concepts. Subsequent chapters demonstrate how design heuristics utilise these founding concepts to support an informal approach to software design evaluation.

2.2 Classical metrics

A *classical* metric is one that generates a large amount of publications that serve to refute or substantiate its ability to measure software. The following three traditional metrics are classical [CBO+88]:

- Lines of code (LOC)
- Halstead's software science (HSS)
- McCabe's cyclomatic complexity (MCC)

This section appraises these classical metrics. They form the nucleus of measurement research carried out in the structure era and were used by the first generation of design metrics implemented for the OO paradigm.

2.2.1 Lines of code

LOC is the most popular estimator of software size to date with over ten thousand citations in measurement literature [Zu96]. The greater the number of LOC, the larger the program size hence the more complex it must be. LOC variants include statement counts, program module/subprogram counts and the average length of a programming module. However, none of these have been able to match the simplicity and wide-acceptance of LOC.

LOC is more specifically a measure of program length than of program size. Consider Fenton's [Fe91] model of size inFigure 2—2. LOC can only be used as a size estimator for programs that solve similar problems and deliver similar functionality. When comparing two programs in which either the essential complexity of the problem or the delivered functionality differ, LOC is merely acting as an indicator of program length and not of size.

Difficulties reported by Levitin [Le86] highlight further problems surrounding LOC. For example, programming languages in which more than one statement can be placed on the same line means that LOC "...measure not the size of the program but rather the size of the program's representation." This difference becomes more pronounced in (OO) software where typographic and programming language styles promote legibility through the use of additional LOC that do not add extra functionality.

Size = *essential complexity + program length + functionality*

where

essential complexity is the complexity inherent in the in the original problem being modelled;

program length is the length of the delivered software;

functionality is the functionality in the delivered software.

Figure 2—2 : Fenton's model of size

A study conducted by Jones [Jo94] concluded that LOC cannot be used to directly measure the quality of OO software. Concurring with Jones, Tegarden, Sheetz and Monarchi [TSM92] also report on other questionable assumptions made about LOC that throw yet more doubts on its ability to measure software complexity:

- What is a LOC?
- Are two programs with the same number of LOC equally complex?
- Is every LOC of the same complexity?

The answer to the first question remains open and is typically resolved on a per programming language basis. Using Fenton's model of size, the answer

to the second question lies with whether the essential complexity and delivered functionality size attributes are similar. The remaining shortcoming for the LOC metric is one directly addressed by Halstead's software science (HSS).

2.2.2 Halstead's software science

Halstead's [Ha77] theory of software science stemmed from an analogy between programming language and natural language. HSS decomposes a computer program into lexical tokens and classifies them as either operators (verbs) or operands (nouns). Halstead's size measures, *length* and *volume*, are functions of token counts on operator and operands.

HSS provoked criticism on numerous fronts. Many of the problems resulted from the context in which Halstead's measures were created. Initially intended to be enacted upon algorithms and not programs, Halstead asserted that "…any software must consist of an ordered string of operators and operands" such that "…nothing else is required and the two categories are mutually exclusive." Such algorithmic-based measures do not scale to computer programs [SCD83][LKSL81][Le86], especially in languages whose operators and operands are used interchangeably (e.g. functional languages). In addition, flaws in the methodology and experimental techniques [HF82], software science's inconsistent counting models [Li82] and erroneous formula derivations [SCD83] have served to further discredit HSS as a general program complexity metric.

Despite the controversy surrounding the validity of HSS, many software practitioners have promulgated its practicality as a complexity measure [CC92][HC87][TSM92][AG83]. However, many have also refuted the value of these few isolated experiments [Ca81][SCD83][LKSL81] pronouncing, "…practitioners can safely ignore software science" [CG90] as "…the software measurement world has discounted the use of Halstead measures as viable in an industrial context" [In91]. A study by Woodward et al [WSD81] corroborates these rebuttals by demonstrating that HSS does not perform

noticeably better than LOC. However, as Verner and Tate [VT87] conclude, this is hardly surprising considering HSS exchanges language-dependence (LOC) for machine-dependence (token counts).

HSS was the first body of work that presented a theory of programming that attempted to differentiate between computer science and software science [CG90]. Although massively influential, HSS's role as a complexity measure is deeply flawed. In retrospect and in accordance with Fenton's model of size, LOC and HSS are size metrics that measure nothing more than program length.

2.2.3 McCabe's cyclomatic complexity

Unlike HSS and LOC, McCabe [Mc76] argued that program size is not representative of complexity and should therefore not be used as the sole measure for modularisation. Cyclomatic complexity uses graph theory to represent the decision structure of a program from which measures to quantify control flow are derived. By inspecting the number of unique paths through a program, cyclomatic complexity can be used "...to identify software modules that will be difficult to test and maintain."

The cyclomatic number is a count on the binary decisions in a module plus one. The greater the cyclomatic number the harder the module will be to understand, test and maintain. McCabe pronounced programs with a cyclomatic number greater than 10 to be complex. Although empirical evidence seemingly concurred with McCabe's complexity threshold of 10 [Wa79][KP74][My77], others believe it to have been arbitrarily chosen [Ca91], thereby questioning the correctness of its application.

At the time of writing, McCabe chose to ignore subtle problems in the way that cyclomatic complexity counted decision structures. Although McCabe removed these counting nuances in a later publication [MB89], it was not before Myer [My77] had addressed them. Myer reformulated cyclomatic complexity to correctly acknowledge and account for compound conditionals. However, like HSS, research illustrated that cyclomatic complexity performed no better than LOC [SH79] and empirical evidence provided by Kitchenham [Ki81] served to corroborate these claims.

2.2.4 Summary

Although length is an important measure, it is often used, erroneously, to denote size [Fe91]. Card notes that practitioners are not really interested in product measures such as size, coupling and cohesion but more with software indicators like development effort to guide the process [Ca91]. Software science and cyclomatic complexity metrics have been (mal)formed to gauge the amount of development effort required to build future software. However, it has been shown that these derivatives are neither convincing indicators of development effort [BSP83] nor do they perform any better at predicting development effort than the simple LOC metric [Ba81]. Given that length is often misconstrued as size, and that size is used indicate other factors in software, is size a good indicator for complexity?

McCabe argued that size is not an all encompassing complexity metric as portrayed by HSS and advocates of LOC. Boehm further debates whether complexity is really related to size at all [Bo81]. Either way, strong correlations between size (HSS, LOC) and control-flow metrics (MCC) have been reported [BSP83][HDH81].

Nonetheless, classical metrics are code metrics. Their tardy application during product development often results in software in which the identified problems are not fixed or are too costly to remove. The need for more discerning measures that can be deployed earlier and to incomplete software

products that are both large and modular caused research to focus directly upon structure metrics.

2.3 Traditional software design metrics

Table 2—1 lists the major internal product attributes for software at the different stages of the development life-cycle [Fe91]. From a design perspective, modularity is deemed the most important property which is composed of a number of contributing internal product attributes. An understanding of how these internal product attributes and their inter-dependencies are modelled by metrics serves as an important foundation upon which alternative vehicles for design evaluation can build.

Metric researchers have independently identified the *essential* attributes of modularity to be information flow, control flow, coupling and cohesion. This section reports on research efforts to understand, define and quantify the presence of modularity in structured designs. Section 2.4 applies this understanding of modularity to manage complexity in OO software. First, we describe control flow and information flow for evaluating the communication inside (intra) and between (inter) software modules respectively.

Requirements Specification	Early Design	Detailed Design/ Implementation
Size: length functionality essential complexity	*Modularity:* coupling cohesion information flow tree impurity reuse	*Modularity:* control flow data structures

Table 2—1 : Internal product attributes during software development

2.3.1 Control and information flow

McCabe's cyclomatic complexity (MCC) uses control flow within a program to measure complexity. However, Robillard and Boloix [RB89] illustrated that by ignoring information flow within programs, MCC renders itself insensitive to system modifications. This realisation underlined MCC's incompleteness as a general purpose complexity metric for maintainability.

Gilb's [Gi77] logical complexity factored program volume into the control flow complexity equation and was shown to be an improvement over MCC [LC87]. A novel approach to control flow by Woodward et al [WHH79] measured the number of control flow intersections, or *knots*, in a program whereby a high knot count indicated high complexity.

Control flow metrics are intra-modular because they focus upon the details *within* a single module. Although reported evaluations of MCC as an intra-modular metric [HC87][My77][SH79][TSM92] highlight its ability to pinpoint certain maintenance risks, intra-modular metrics defined purely in terms of control flow are incomplete. They require knowledge of how *information* is consumed within and between these modules.

Information flow is measured by examining how data is passed between modules [Fe91]. Henry and Kafura [HK81][He79] defined the total information flow between system modules in terms of *global* and *local* flows. In this model, *well-formed* information flowing between software modules in a controlled manner is representative of good internal structure. Figure 2—3 depicts Henry and Kafura's different types of information flows within a system.

Global information flow has been defined by researchers prior to Henry and Kafura as common coupling [SMC74] and subsequently as data coupling [CG90]. However, it was Henry and Kafura's complexity measure that evaluated the connections of a module to its environment based on **fan-in** and **fan-out** properties that received considerable attention from the measurement community, such that the:

- *fan-in* of a module A is the number of local flows into A and those data structures that A retrieves information from;

- *fan-out* of a module A is the number of local flows that emanate from A together with the number of data structures that A modifies.

Using Brooks law of programmer interaction [Br75] and Belady's formula for system partitioning [Be81], the complexity, or connectedness, of a module to a system is defined as:

$$\textit{complexity of module } M = \textit{length}(M) + (\textit{fan-in}(M) \times \textit{fan-out}(M))^2$$

Shepperd [Sh88] refined Henry and Kafura's metric to disambiguate between control and information flow in its definitions of local flow. Shepperd also omitted *length(M)* from his information flow metric reasoning that module length is an attribute of size and not of modularity. Shepperd's revised information flow metric is thus:

$$\textit{complexity of module } M = (\textit{fan-in}(M) \times \textit{fan-out}(M))^2$$

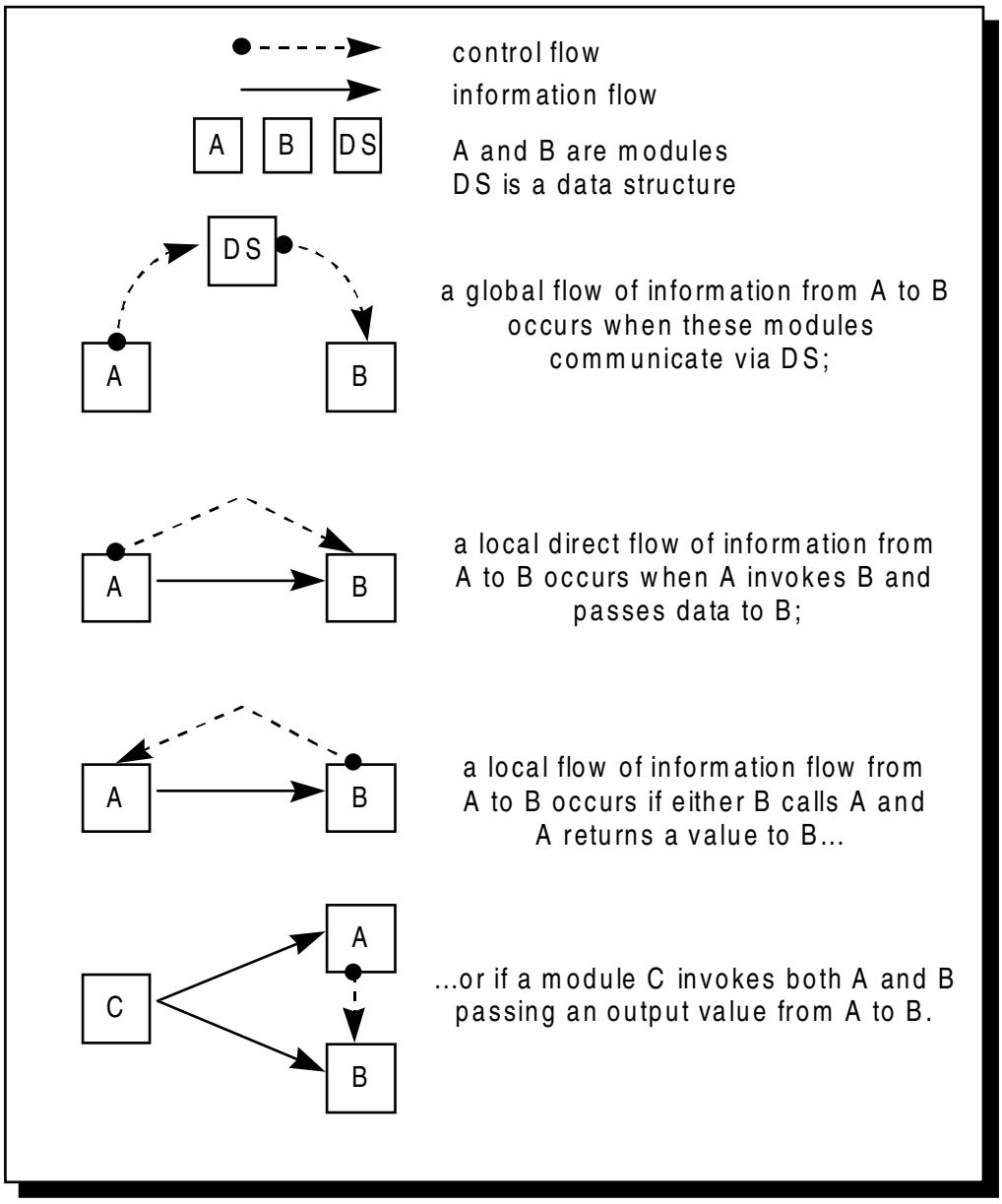


Figure 2—3 : Defining information flow between modules

Where intra-modular metrics focus upon those properties*within* a module, inter-modular metrics are more concerned with the communication*between* modules. Henry and Kafura's model of information flow inextricably combine (confuse) intra-modular and inter-modular metrics. As a result, their metric can only be applied when all the attributes needed to calculate information flow become available. Invariably, this is during the early coding phases of software development, such that information flow is not a*true* design metric [Sh88].

Ideally, design complexity metrics should present a clear separation of concerns so that:

- inter-modular metrics can be deployed during early design to assess architectural and structural complexities when only the interfaces between modules are present;
- intra-modular metrics can be applied to get feedback on the internal structure of software modules when implementation details become available;
- a combination of intra-modular and inter-modular metrics can be applied both independently and synergistically to design products to gain an overall view of system modularity; a hybrid model.

Maintainable designs make good use of modularity which is reflected in the importance placed upon metrics that can accurately evaluate its presence within software early enough to apply their suggestions. Modularity metrics that consider the above criteria take a well balanced and clearly delineated approach to control and information flow. However, aspects of control and information flow take place both within and between modules. The next section examines the primary attributes of modularity: size, coupling and cohesion. Here, control and information flow metrics are used both inside modules (cohesion) and between them (coupling).

2.3.2 Modularity

2.3.2.1 Size

A number of module size limits have been proposed in the literature [WSD][We70] for which thresholding criteria include: u pages of source, v lines of code, w executable statements, x binary locations and y instructions for assembly language. Bowen [Bo84] argues that practitioners [DU79] enforcing indefensible module size limits as standards rather than as guidelines, do so to the detriment of software quality. Bowen further notes

that, "…designers and programmers are not directly concerned with module size because they have justifiable confidence that the design methodology will result in reasonably sized modules." Coupled with the assertion that arbitrary module size limits do not factor into how good programmers produce high quality modules [CPM85], software metricians have questioned the usefulness of size limits as a means to achieve modularity.

Harrison et al [HMK+82] suggest that size, on a nominal scale, can be used to allocate modules to specific complexity categories. By using established guidelines and not unvalidated standards [Gl87] researchers can then apply more discerning measures of modularity, based on coupling and cohesion, to evaluate module complexity for a particular size category. Conversely, Stevens [St81] contends that size limits should only be considered after design efforts to modularise software have been employed; that is size should not be used to artificially compartmentalise software. Either way, no industrial-wide accepted standard for module size, for any language, has ever been reached [Gl87].

2.3.2.2 Coupling

Inter-modular metrics are predominantly based upon the concept of coupling. These metrics promote loose coupling between modules indicated by a low [CPM85] and well distributed [Be81] fan-out. Henry and Kafura's information flow metric implemented fan-out but as Card and Agresti [CA88] report, without regard for the strength [HB85] or type [SMC74] of coupling between modules.

Yin and Winchester [YW78] implemented inter-modular metrics that acted upon design structure charts. In their model of design complexity, two modules were coupled if control and/or information flowed between them. Shepperd [Sh88] reported on two similar inter-modular metrics [Be79][Ch79] that also view modules as nodes and the communication between them,

namely method calls, as arcs within a graph-oriented model of software architecture.

Graph-based approaches to modularity define complexity in terms of tree impurities (the amount a graph structure diverges from a tree-based structure) that can then be re-structured to eradicate both inter-modular irregularities and cyclic dependencies. McCabe and Butler [MB89] also viewed system complexity as hierarchical tree structures. By performing reductions on the hierarchical ordering of program modules, McCabe and Butler aimed to manage system complexity by highlighting software modules *incommunicado* .

Silverman et al [SGB83] defined inter-modular metrics based upon the total number and distribution of inter-connections within a software system. Although their metrics did not differentiate between information and control module connections, they did prove useful for the comparative analysis of different designs. Using this model, highly connected modules have large metric values indicating high complexity. Silverman et al's minimalist approach to design for maintenance permitted the timely identification of potential trouble spots in software architecture and the ability to assess the impact of future system changes. Most importantly, their metrics made no assumptions about the underlying implementation.

Card et al [CCA86] showed fan-out to be the definitive attribute in the design complexity equation. As fan-out represents the degree of coupling between a module and its environment, these observations would seem to concur with Troy and Zweben's [TZ81] early conclusions that coupling is the best indicator for design complexity. The more calls module A makes to other modules, the greater it's fan-out, the harder that module will be to understand, fix, change or reuse. In addition to restricting fan-out, Belady [Be81] further observed through inter-connection matrices that the *distribution* of fan-out amongst all software modules is an important consideration when measuring system modularity.

2.3.2.3 Cohesion

Constantine and Yourdon [CY79] defined seven different types of cohesion: coincidental (weakest), logical, procedural, temporal, communicational sequential and functional (strongest). For structured designs, where the unit of decomposition is the function, a highly cohesive module implements a single function that performs a single task [St81]. Hence, cohesion metrics determine the extent to which individual modules embody a single abstraction.

Card et al [CPM85] illustrated, using FORTRAN programs, that highly cohesive modules have a lower fault rate and cost less to maintain than modules that have low cohesion. They used subjective judgement to manually assign modules a cohesion rating. Based upon their research, Card et al advised developers to write highly cohesive modules that are *large* enough to encompass the *entire* function.

Emerson [Em84] implemented a discriminant metric that objectively placed a module into one of three cohesion categories. The seven types of cohesion defined by Constantine and Yourdon were allocated to these categories on the basis of modifiability and understandability. Emerson applied his discriminant cohesion metric to FORTRAN programs where extrapolations on the number of variable references per executable statement were used as a model of cohesion. This intricate, language dependent approach to cohesion reiterated the problems associated with quantifying an inherently qualitative property. However, it highlighted a means to competently classify cohesion along a nominal scale within FORTRAN programs.

Patel et al [PCB92] provided a measure of module cohesion for modules that are composed of one or more programs. Cohesion, defined in terms of information strength [My76], was used to determine the *similarity* between programs within a composite module and other programs in the system. Programs were then logically re-allocated to the modules that they were most *friendly* with, thereby increasing cohesion and indirectly reducing coupling.

Similarity measures were calculated by computing type vectors for each program. Type vectors were in fact complex reference counts calculated by inspecting usage statistics on global, data and control variables within a module. Hence, cohesion examined the apportioning of system data to modules, subsequently assigning those programs that used this data extensively to the modules that owned it.

In developing a quantitative model of cohesion and coupling, Dhama [Dh94] addressed the frequently asked question:

> *"How many modules are optimal to implement a given functionality?"*

That is, for a large, highly cohesive module that needs to be fragmented, what criteria can be used to ensure that the resultant modules are better than the whole. Note that fragmenting highly cohesive modules is counter to the arguments of Card et al [CPM85] which explicitly state that highly cohesive modules should be *large* enough to embody the *entire* function. However, by taking this premise to the extreme, a number of large system functions implemented as single modules would result in a monolithic architecture. Dhama offered the following guideline for those developers faced with this fragmentation problem:

> *"The coupling of each of the modules should be better than or equal to that of the single module."*

Dhama's coupling and cohesion metric formulas were, like Patel et al [PCB92], counts on global and control data used within a module. Unfortunately, Dhama's metrics were only tested against two simple modules that served to highlight the possibility for future work.

Models of cohesion attempt to quantify an inherently anthropomorphic task that can lead to spurious results and misleading interpretations. Yin and Winchester [YW78] used fan-in as a measure of cohesion where high values indicated low cohesiveness. However, low cohesion is inversely related to reuse. Card and Agresti [CA88] reported that modules with large fan-in

values are typically representative of general purpose library modules. For developers that cannot distinguish between the type of modules in their systems and how they are used, fan-in proves inappropriate as a measure of cohesion and therefore modularity.

To summarise, highly cohesive modules perform a single task. Objectively, this is assessed by how well a module makes use of the data that it owns. In the absence of subjective judgement, metrics cannot guarantee that a module performs a single task, only that the majority of its data is operated on by the majority of its functions.

2.3.3 Hybrid metrics

Robillard and Boloix [RB89] defined a hybrid model as one that is composed of measures for size, control flow and information flow. They assert that hybrid models are more representative of complexity than those models that exclude any one of these attributes. Robillard and Boloix's hybrid interconnectivity metric was shown to out perform classical metrics but proved difficult to compute and could only be applied to source code.

In line with Harrison et al's [HMK+82] assertion that control and information flow metrics prove effective when evaluating programs of a similar size, Reynolds [Re84] derived a series of inter-modular metrics that could be applied to partially developed programs categorised on the basis of size. Using Halstead's software science (HSS) to calculate program size, Reynolds reasoned that design changes tend to incur size changes and therefore size metrics are the best indicators for complexity. However, because Reynolds' metrics employed HSS, the same problems that befell HSS, befell them. Furthermore, as with Robillard and Boloix's interconnectivity metric, Reynolds design metrics could only be applied to program source.

Card and Agresti [CA88] viewed a hybrid complexity model as a combination of intra-modular and inter-modular metrics that collectively defined the architectural design complexity in a system. Their model further stipulated

that design complexity is relative to the number of modules in a system such that:

```
    C = S + L
where
    n = total number of modules
    C = total design complexity  / n
    S = inter-modular complexity / n
    L = intra-modular complexity / n
```

Harrison and Cook [HC87] also separated inter-modular and intra-modular design properties by defining system complexity as:

$$\sum_{i=1}^{\#\text{ subprograms}} \left[SC(i) \quad * \quad MC(i) \right]$$

where ***SC(i)*** is the inter-modular complexity contributed by subprogram ***i*** and ***MC(i)*** is the intra-modular complexity contributed by subprogram ***i***. Harrison and Cook implemented ***MC(i)*** using McCabe's cyclomatic complexity that requires the presence of program source. Card and Agresti's intra-modular metric that counted I/O variables and information flows emanating from a module also relied upon implementation details located within the module. Nevertheless, these hybrid complexity models made a clear distinction between the contributions made by inter-modular metrics and those by intra-modular metrics. In doing so, they elevate the focus of hybrid complexity models from program source to the design level. This enables the early deployment of metrics to be achieved by applying inter-modular metrics first and then a combination of both modularity metrics when a system's implementation details become available.

2.3.4 Conclusion

Kernighan and Plauger [KP74] postulate that "…the best programs are designed in terms of loosely coupled functions that each does a simple task." Scaling up functions to modules, and programs to systems, modularity becomes the most important property in software for building maintainable designs. By focusing upon module interfaces, inter-modular metrics can be

applied in a timely manner during design [HS90]. Intra-modular metrics further annotate the results gained by applying inter-modular metrics as the level of design detail increases.

Rombach [Ro90] empirically clarified the role of inter-modular and intra-modular metrics during software development by summarising the results of measurement during the early phases of the life-cycle. He illustrated that a high correlation between architectural design documents and maintainability can be achieved using inter-modular metrics based upon modularity. Hybrid complexity models were also shown to competently discern maintainability in software designs. However, a low correlation resulted from the early deployment of intra-modular metrics to design products.

Metric research has clarified the unclear role of control flow and information flow at the architectural design level for both inter-modular and intra-modular metrics. It has also defined and applied the concepts of coupling, cohesion and size for implementing modularity metrics. In particular, research has shown coupling to be the definitive attribute of modularity that is assessed using graph/hierarchy based approaches and functions on fan-out. Although cohesion is inherently qualitative, progress towards its quantification along a nominal scale has been illustrated. Finally, the role of size as a criterion for software modularisation has been bounded and shown to be secondary in nature to that of coupling and cohesion.

Using these largely learned attributes and techniques for evaluating modularity in structured designs, the next section reports on how this knowledge has been applied to assess design quality in OO systems.

2.4 Object-oriented metrics

In OO systems, the class replaces the function as the primary unit of decomposition. Like traditional design metrics, OOD measures focus upon aspects of modularity. In doing so, they need to model the expressive relationships that now exists between classes. Methods to define, quantify and integrate these additional modularity criteria have been the main focus

of OOD metric research to date. The remainder of this chapter reports on OOD metrics in the literature, illustrating why inheritance adds additional complexity to the modularity equation. We contend that the principal benefits of inheritance comes at a significant cost to software complexity.

Initial research into OOD metrics went back to measurement first principles by examining aspects of size and functional content within classes in much the same way that program modules were. Section 2.4.1 describes how traditional metrics were used to evaluate OO software. The inherent limitations of their use within the OO paradigm are highlighted. Section 2.4.2 presents the classical metrics for OOD. These measures dispense with traditional metrics in trying to establish a firm theoretical and pragmatic foundation for OOD measurement. Finally, section 2.4.3 critiques these classical metrics for OOD illustrating how they have ultimately served to refine notions of modularity within OO software.

2.4.1 Traditional metrics for object-oriented software

[CK89][Bo89][Lo90][AM93][HTM91] have argued that traditional metrics cannot be used to measure (OO) software. They warned that traditional metrics cannot be applied to software fundamentally different from the structured model of programming that they were implemented to measure. However, in defending the role of structured metrics in OO software measurement, Tegarden et al [TSM92] rebuffed Moreau and Dominick's [MD89] remark that "…the only place that existing, traditional, intra-object software metrics might be effective is within a particular method within an object." Tegarden et al demonstrated how traditional metrics can be used to reduce procedural complexity in OO software by focusing upon notions of inheritance and polymorphism. They adapted Halstead's metrics so that operators related to class methods and operands to variables. It was shown that classes that employed inheritance and polymorphism had fewer operators and operands in the final system and therefore lower procedural complexity.

Coppick and Cheatham [CC92] also applied traditional metrics to OO software. In treating the class as a program module, they suggested a cyclomatic complexity for an object to be 100. This figure was ten times that of McCabe's original threshold and was meant to cater for the entire complexity of an object contributed by all its methods. Although Tegarden et al also applied cyclomatic complexity to *individual* class methods, they warned that the arbitrary summing of class method complexity was undefined. Furthermore, it implied that descendent objects do not rely upon methods implemented within their ascendent objects.

Coppick and Cheatham [CC92] and Tegarden et al [TSM92] treat the class as a better program module. In fact, they redefine the semantics of a class to suit that of a program module. This enabled them to apply both Halstead's and McCabe's metrics to classes. However, in both cases they were ignoring the expressive relationships that exists between classes. More specifically, their metrics ignored complexity that was*inherited*.

So, if the OO model of software is fundamentally different from that of the structured model [Bo89][Lo90], such that software engineers cannot enforce function-oriented measurement techniques upon OO software products [CK91], what should OOD metrics measure and what properties should they possess?

2.4.2 Classical OOD metrics

In answer to these questions, Chidamber and Kemerer[CK91] presented a candidate metric suite that modelled the static properties within an OOD. Chidamber and Kemerer formulated a suite of OOD metrics to quantify the observable properties of classes defined by the Booch object model [Bo86].

Chidamber and Kemerer's OOD metrics outlined in Table 2—2 can be categorised according to the attributes of modularity they purport to measure. The noted introduction of inheritance in Table 2—3 highlights its unique role within OOD measurement.

Weighted Methods per Class (WMC) sums the complexity of individual methods. If a weight of one is assigned to every method, WMC regards class complexity as the number of methods in a class.
Depth of Inheritance (DIT) asserts that deeper class trees constitute greater design complexity since more methods and classes are involved.
Number Of Children (NOC) counts the number of immediate subclasses to a class in the inheritance hierarchy. NOC gives an idea of the potential influence a class has on the design.
Coupling Between Objects (CBO) counts the number of non-inheritance related couples between classes. Object A is coupled to object B if A uses the methods and/or instance variables of B, or vice-versa.
Response For a Class (RFC) is the set of all methods available to an object. The greater the RFC the greater the level of understanding required on the part of the tester.
Lack of Cohesion in Methods (LCOM) defines class cohesion by amount of use methods make of their instance variables. LCOM identifies disparity between a class's methods and its data, thereby indicating flaws in the design of these classes.

Table 2—2 : Chidamber and Kemerer's OOD Metric Suite

Modularity Attribute	Classical OOD Metric
coupling	coupling between objects (CBO) response for a class (RFC)
cohesion	lack of cohesion in a class (LCOM)
size	weighted methods per class (WMC)
inheritance	depth of inheritance (DIT) number of children (NOC)

Table 2—3 : Modularity attributes for the classical OOD metrics

Chidamber and Kemerer's theoretically stated OOD metrics were analytically evaluated against Weyuker's [We88] list of software metric evaluation criteria that defines several properties that syntactic complexity metrics should exhibit. Although Weyuker's properties have received criticism on many fronts pertaining to multiple views of complexity [Fe91], inconsistent scaling

characteristics [Zu92] and insufficient conditions for determining good complexity metrics [CS91], Chidamber and Kemerer argued that Weyuker's properties represented a rare, well-documented, formal and analytical approach to metric validation. Chidamber and Kemerer [CK94] also empirically validated their OOD metric suite against large C++ and Smalltalk class libraries.

Chidamber and Kemerer's OOD metrics were the first to be both theoretically and empirically validated. Their metrics provided an important foundation upon which many future OOD metrics were based.

2.4.3 OOD metrics

2.4.3.1 Objects coupling and cohesion

Chen and Lu [CL93] proposed an OOD metric suite that both extended and revised Chidamber and Kemerer's classical OOD metrics. They revamped both the `RFC` and `CBO` coupling-based metrics to discern aspects of fan-in, fan-out and cyclic coupling at both the method and the class level.

Chen and Lu also redefined class cohesion solely in terms of its methods. They argued that `LCOM` was inappropriate as a cohesion metric because it relies upon the presence of instance variables that might not be available during early OOD. However, if class cohesion describes the binding of all elements defined within the same class [EKS94], then as`LCOM` portrays, this *includes* instance variables. The presence of instance variable based metrics within Chen and Lu's measurement suite served to contradict their reasons for defining class cohesion strictly in terms of their methods.

Mingins et al [MDS93] also defined cohesion solely in terms of class methods. They viewed the degree of cohesion in a class, or coherence, as the similarity of methods within a single class such that "…a class is coherent if the methods work together to carry out a single identifiable purpose." Using their method-biased model of cohesion they prioritised classes in an Eiffel hierarchy according to their coherence values. They demonstrated that

coherence aids reuse by reducing complexity and that it can be determined without knowledge of how a method is implemented.

Hopkins [Ho94] also concentrated upon interface-oriented class metrics that hold no pre-conceived notions of a class's underlying implementation. Although his metrics did not specifically measure cohesion, they defined method complexity in terms of the cardinality and polymorphic capabilities of its parameters. Class complexity was then calculated as the sum of its method complexity values. In both cases Mingins et al and Hopkins demonstrated a novel approach to measurement that focused purely upon the inter-modular properties of a class.

Focusing more directly on the issue of maintainability in OO systems, Wei and Henry's [HW93][WH93] metric suite included five of the six classical OOD metrics. They purposefully omitted`CBO` to define coupling along three vectors: inheritance, message passing and data abstraction. Their coupling metrics modelled the relationships that exists between classes as per their definition supplied by Booch [Bo86] such that:

- using (message passing metrics)
- inheritance (inheritance metrics)
- aggregation (data abstraction metrics)

Coupling through inheritance was defined in terms of`DIT` and `NOC`, where data abstraction coupling represented black box reuse via aggregation. For their message passing metrics, Wei and Henry expressed coupling in terms of fan-out. These early research efforts into classical OOD metrics re-enforced the primary role coupling plays in modularity measurement and that fan-out still appears to be the most effective means to measure it.

2.4.3.2 System vs. class level metrics

The maturation of the OO paradigm saw OOD metrics differentiating between class and system level metrics. Class level metrics measure the

complexity of individual classes composing the design, whereas system level metrics evaluate the architectural and structural integrity of software by inspecting groups of classes, or clusters.

Kowele [Ko93] categorised all the classical OOD metrics bar `NOC` as class level metrics. Stiglic et al [SHR95], in line with Chidamber and Kemerer's thinking, placed all the classical OOD metrics at the class level. Both [Ko93] and [SHR95] implemented system level metrics as averages, means and medians over values gleaned from class level metrics. This lack of true system level measures has only recently been observed by the OO and metric community. Classes are no longer considered the unit of design. Instead classes collaborate in groups that are highly inter-operable [GHJ+94]. System measures for clusters, or patterns, of classes have not been forthcoming.

2.4.3.3 Summary

Attendees at an OOPSLA metric workshop [BL93] concluded that Chidamber and Kemerer's metrics provide a solid foundation for future OOD measures but not a solution. Numerous authors have applied, refined and re-defined the classical OOD metrics to gain a deeper understanding of the concepts they purport to measure. [HW93][CL93] provided more informed opinions on coupling within OO software. [MSD93][CL93] gave an alternative method-based view of cohesion, [SHR95][Ko93] categorised current OOD metrics as either class or system level measures and [CM92] noted that Chidamber and Kemerer's metrics were insufficient to cope with class hierarchies that possessed repeated and/or multiple inheritance.

2.5 Conclusions

The orthogonal nature of software produced by the object-oriented and structured paradigms ensures that OO metrics remain fundamentally different from traditional metrics. The function-oriented view held by traditional metrics saw early attempts to apply them to OO software impeded by the unique relationship that exists between classes: inheritance.

Traditional metrics have been restricted to measuring the intra-object properties within an OOD [MD89][AC94].

Although classical OOD metrics are simple and incomplete [BL93], they identified early on that inheritance plays an integral part in defining system modularity. These metrics have been critiqued over time and extended to address specific quality criteria such as maintainability and reusability. A metric taxonomy framework defined by Abreu and Carapuca [AC94] illustrates that the majority of OOD measurement has taken place at the method and class level with minimal activity at the system level.

As a cautionary note, Byard [By94] warned that although much progress has been made with regard to class metrics, a concerted effort by both metric implementers and their users is required to prevent the class from becoming a convenient counting vehicle. To demonstrate his point Byard described how scientists in the 19th century filled human skulls with lead shot to measure the volume of the brain. Byard illustrates that "…from this objective measure, these same scientists wrote that they had, in fact, measured mental capacity."

Byard contends that software developers are counting classes (lead shot) in software (the skull) and incorrectly interpreting what is actually being measured. Formalising (class) metrics is a two phase process that involves moving from concepts to formulae and back again to ensure that the formulae are a true representation of the concepts. Unfortunately, it is on this return journey that developers sometimes fall short, resulting in metrics that are detached from the concepts they original set out to measure. A means to comprehensively annotate and provide extensive feedback on the original concepts during design evaluation form a major objective for this research.

2.6 Summary

Modularity plays an important role in creating high quality software designs. Both inter-modular and intra-modular metrics are used to evaluate the

architectural and structural integrity of design products. In general, inter-modular metrics can be deployed early in the design process offering timely and instructive quality feedback. As the design becomes progressively more detailed, intra-modular metrics provide additional evaluation information in support of the feedback gained from inter-modular metrics.

By examining the design metrics from both the structured and object-oriented paradigms, clearly the only similarities between them are the concepts upon which they are based. These concepts are seen as the attributes of modularity: coupling, cohesion and size. Coupling is considered the most important modularity attribute, followed by cohesion and then by size. Inheritance in OOD metrics is a special case that affects all of the aforementioned modularity attributes.

The next chapter describes the rationale for moving away from this metric-centred view of software quality. It specifically demonstrates why alternative mechanisms for OO software evaluation are required and by whom. Chapter 4 reports on the implementation of a design heuristic catalogue for evaluating software designs in an informal manner. A prototypical tool responsible for deploying these design heuristics is presented in chapter 5. How the tool was integrated into OO curricula and its role as a design reviewer is discussed in chapter 6.

Chapter 3

The design evaluation problem

3. Overview

This chapter elaborates the design evaluation problem. Section 3.1 presents our definition of design evaluation and its objectives. It highlights a number of key areas to address so that we may equip software practitioners with pragmatic tools and techniques for effectively evaluating their software. The process by which designs are typically assessed is outlined in section 3.2. Collectively, these preliminary sections demonstrate a need for an alternative mechanism for performing design evaluation than that provided by software metrics*and* its requirements.

Section 3.3 reports on two surveys that were carried out to ascertain who stands to benefit from solving these design evaluation problems. The survey respondents described how they were currently tackling issues regarding design evaluation within their software processes. From these surveys, a requirements short-list for improving design evaluation within these institutions was drafted. Section 3.4 proposes heuristics as an alternative approach to software design evaluation based upon the identified problems in sections 3.1 and 3.2 and their solution requirements documented in section 3.3.

Design heuristics are shown to be concentrated pieces of design expertise that deliver knowledge and experience from the expert to the novice. They build directly upon the metric research presented in chapter 2 to provide a practical alternative for enhancing the maintainability of OOD documents.

3.1 The design evaluation problem

Our definition design evaluation:

> *"Design evaluation is the means by which a user articulates a design product to gain a deeper understanding of both its good and bad features. The user not only learns from the discovered problems but equally from the observed solutions presented within the context of their own design."*

Currently, the evaluation of software design products is a task reserved for software metrics. This chapter raises a number of important issues surrounding design evaluation that software metrics do not address. Most importantly, according to the above definition, design evaluation relies upon elements of *learning* as well as product improvement. With a large number software practitioners learning object technology, there is a need for design evaluation techniques that place more emphasis on education than on enhancement [GH96b]. A means to combine the two, so that through enhancement comes education, is one of the major objectives of our research.

Throughout this chapter software metrics are constantly used as a means of comparison. This is not because software metrics themselves are plagued with problems, but merely because they are considered as the primary means by which to evaluate software designs. However, the review of software measurement in the previous chapter brought to the fore several important features that are lacking and/or implied when performing metric-based design evaluation. Reflecting upon the following list of omissions, we contend that software metrics form only part of the design evaluation problem:

- there is no common vocabulary for *problems* arising during design;
- no explicit and/or informative links exist between a design problem and its solution(s);

- design problems are defined in terms of low level concepts and are particular to a given domain and/or development phase;

- informal processes that employ formal methods of assessment forego many of the benefits gained by informally evaluating a design.

Neither the definition of design evaluation nor its associated problems are indigenous to computer science. These problems can be seen in such diverse disciplines as architecture, medicine, civil engineering, physics and so on.

For example, consider a doctor evaluating the health of her patient. Medicine, like software design, is a vast subject populated with conflicting and contradictory approaches to problem solving. However in medicine, complex problems have been categorised and fragmented into smaller, well-defined problems that form a common *vocabulary*. An established vocabulary permits groups of doctors to readily communicate diagnostic information. Furthermore, because every *problem* has recognised *treatments*, doctors can make informed and timely decisions on what corrective actions to take.

Patient diagnosis is typically a *two-phase process* in which a precursory informal evaluation precedes a more formal and rigorous medical procedure. The informal evaluation looks to trends and symptomatic conditions as early warning indicators before moving to more formal techniques. A common vocabulary for problems and their respective solutions together with a two-phase evaluation process provides an effective and established means by which doctors diagnose and treat their patients. In software, metrics support a formal approach to design evaluation. We present an alternative approach to design evaluation for use within development processes where the degree of formality offered by metrics is neither warranted nor desirable.

This research places design evaluation and its associated problems within a software context and specifically for design documents produced within the object-oriented paradigm. The remainder of this section discusses each of these problems in more detail. By addressing these problems in section 3.4,

design heuristics are presented as effective mechanisms for the informal evaluation of OOD documents.

3.1.1 Documenting problems

Chapter 1 alluded to the fact that design heuristics need to be *small*, *simple* and *legible* in order for them to be useful to the majority of developers performing design evaluation. Chapter 2 then reported on the metric-centred view of design evaluation. In this section we examine more closely how design problems are defined by metrics and reformulate these important observations as requirements for design heuristics.

Metrics are used to document *large* design problems. Consider, for example a modularity metric. Before software practitioners can confidently apply a modularity metric and interpret its feedback, they must have an intimate knowledge of low level concepts such as coupling and cohesion *and* how they affect a software design within a given paradigm. In a learning environment, this is too much to presuppose. During design evaluation only a limited knowledge of the paradigm is assumed and the focus lies resolutely with gaining an increased understanding of the resident design problems. Large problems expressed in terms of low level concepts do not engender learning on behalf of the (in)experienced software engineer. We contend that the design problems must be small enough to be assimilated in their entirety.

In addition to fragmenting problems into smaller and more manageable parts, the resultant parts must also be legible. Legible design problems refer to observable problems that a *developer* encounters [GH97b]. Observable problems must therefore be expressed in terms of the target paradigm. For example, observable problems that occur during OOD refer to applications of inheritance, the construction of abstract interfaces, the manipulation of polymorphic objects and the creation of robust, yet flexible classes. Metrics, on the other hand, tend to be more direct and specify design problems in terms of low level design concepts. In doing so they target experienced

developers or those specifically charged with instilling high degrees of quality in an organisation's software products.

In practice, metrics are neither available nor legible to the wider audience to the extent that it is unlikely that newcomers to object technology will relate to the information that metrics attempt to convey [GH97a]. Developers communicate in terms of the problems that *they* encounter during design and these problems are typically pragmatic uses of the technology.

The documentation of recurring design problems provides a common vocabulary that developers can use to describe design errors and anomalies. A similar sort of documentation for software solutions was one of the main driving forces behind the growing design patterns movement [GHJ+94]. The development and application of software patterns now pervade OO literature but their focus lies with selecting solutions from *recognised* design problems. However, design patterns are not intuitive to apply [Ri96], thereby requiring developers to first appreciate the problem before exacting a solution.

This research focuses upon identifying and fully documenting common design problems voiced by both novice and experienced developers performing OOD. In doing so, a central repository of *legible* design expertise becomes readily available during design evaluation to *all* those that partake in learning, applying and developing object technology.

3.1.2 Keeping solutions with problems

Design evaluation encompasses more than just an understanding of the problem; design evaluation must embody aspects of a problem's solution. From an educational perspective, a problem without a (partial) solution has limited benefits and a solution without a problem has no context. Design evaluation within a learning environment must augment the two to ensure that when problems are detected the appropriate solutions are presented. Furthermore, the suggested solutions should be tailored to a set of pre-determined design evaluation criterion. For example, when aiming to construct maintainable software systems, the solutions to problems located

within a design should be tailored with respect to notions of maintainability. If the main objective is the production of reusable software abstractions, solutions that enhance reusability should take precedence over all other product qualities.

3.1.3 Multi-domain, multi-phase design documents

OOD metrics are typically validated within a specific application domain *and* target software products delivered at a particular phase of the life cycle. This creates a problem with today's software that usually spans multiple application domains and whose software products need to be continually evaluated from as early as late analysis through to implementation and thereafter. A means by which to describe a set of generalised design problems that transcend both the domain and development phase boundaries would increase the range of software contexts in which they could be deployed and the timeliness of their application.

3.1.4 Informal vs. formal

In certain circumstances, the precision afforded to metrics is neither warranted nor desirable. The metric-centred view of evaluation aims to provide software scales along which design products can be compared, contrasted and ranked. This type of evaluation requires the presence of *accurate* formal definitions to measure software products along superior scales than that offered by the nominal and ordinal types. According to our definition of design evaluation, such precision is secondary in nature to understanding. As reported in section 3.4, evaluation mechanisms that employ such formal methods of assessment to drive informal processes, forego some of the benefits attributed to informal techniques.

3.2 The design evaluation process

Design evaluation usually takes place during design review. At this time a design document's conformance to specification is checked and decisions on whether it has attained the required level of quality are made. Typically, it is

the role of metrics to ascertain the presence or absence of software quality. However, the majority of learners do not undergo design review and metrics have only limited benefits during design evaluation [GH96b]. Hence, design evaluation for this, the most common type of developer, is not forthcoming.

Teaching methods are used to educate learners in the (mis)practices of software design. They are highly pragmatic and informal in their approach to OOD education [HE95], such that a means for design evaluation within these methods should also be informal [GH96b]. Teaching methods serve to supplement well-documented OOD methods [Bo94][Ru91][Wi90][CY91] that are seen to be incomplete [Ma95], process-biased and provide only limited assistance at the design product level [GLH96]. It is the responsibility of the teaching method to complement incomplete OOD methods, providing insight into what is considered good and bad regarding their design products. Gibbon and Higgins [GH96b] observe that in order to achieve this goal, teaching methods must consider the following characteristics when deciding which design evaluation technique to employ:

- the software development environment is predominantly informal;
- emphasis is on quality education rather than enhancement;
- quality issues and the technology under scrutiny may be largely unknown to its learners.

Metrics focus upon locating software defects. However, they rarely divulge an informative *why*. As section 3.4 reports, metric nomenclature revolves around low level concepts and not the common design problems faced by learners. In an educational environment, where the *whys* take precedence over the *whats*, *wheres* and *hows*, metrics as they currently stand, are an undesirable option for design evaluation within teaching methods.

To date, design evaluation has been subsumed by metric research. This is as expected considering who (industrialists) have been evaluating the majority of today's software and the importance placed upon doing it accurately.

Nevertheless, the industry cannot ignore the fact that the amount of OOD learners far outnumber the amount of OOD experts. To redress the balance, techniques by which newcomers to the technology can readily and effectively learn (in)directly from experienced developers are required. Early exposure to design evaluation is an essential part of a developer's training in issues of software quality and how to achieve it using object technology [GH97a].

Figure 3—1 illustrates how design evaluation can be incorporated within teaching methods [Gi96]. Typically, educating learners in aspects of OOD takes on a three phase process in which concepts are presented in lectures, applied during design and then assessed by formal examination. Design evaluation is usually a precursor to the assessment phase once the concepts have been learnt and applied.

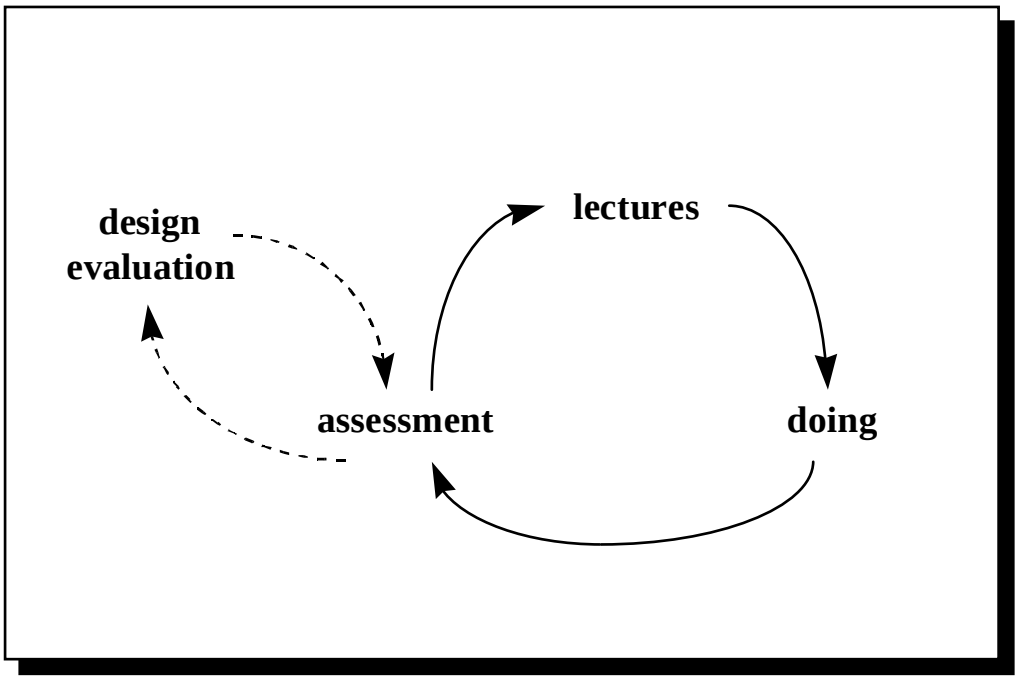


Figure 3—1 : Design evaluation within teaching methods

The problem with this model is that at the most important phase, the doing phase [SP96][SK96], learners are not getting the required feedback, and the feedback that they do receive is presented too late. Chonoles et al [CSM95]

report that over 75% of design errors are identified before, rather than during the assessment phase. Therefore, a means to evaluate designs before assessment, and more importantly during the doing phase, will ensure that design feedback is timely, and therefore instructive to the learner.

3.3 Surveying OOD curricula

Two surveys were conducted in conjunction with Staffordshire University to establish how object technology is being presented to learners. The first survey ascertained the state of OOD curricula within UK academia. The second survey was conducted at the OOPSLA'96 Educators' Symposium that had international representatives from both academic and industrial institutions. The objectives of the two surveys were clear:

- to determine whether there is a common set of OOD methods, programming languages and CASE tools employed by educators;
- to establish whether tools and techniques to enhance the quality of software designs would enrich the teaching/OOD process;
- to evaluate the effort required to integrate such tools and techniques into current teaching/OOD processes.

The UK survey had 27 respondents and the OOPSLA'96 survey had 25. The latter survey went into slightly more detail than the former soliciting from its respondents additional issues pertaining to CASE tool usage and course organisation. A more in-depth analysis of the second survey was presented at the OOPSLA'96 Educators' Symposium panel. Both surveys have been published in [GH96a][Do96], of which [GH96a] have been explicated further by Chandler and Hand [CH96]. A concise summary of the poignant survey points is reported in this section.

3.3.1 OOD methods

Booch [Bo94] and OMT [Ru91] are the most popular OOD methods amongst educators. Figure 3—2 collectively apportions the OOD methods employed by both the UK and OOPSLA survey respondents into a single graph.

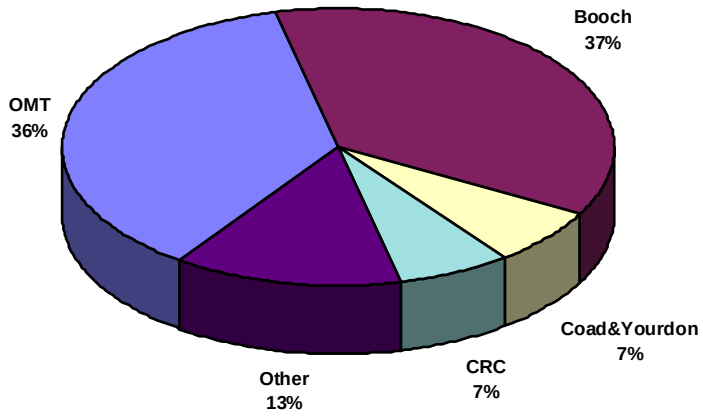


Figure 3—2 : OOD method usage

An important trend within academia and industry is a move by educators to adopt a multi-method approach to OOD. It is highly unlikely that any one OOD method cannot fulfil all the analysis and design needs for a teaching method or software project [GH96b][Ma95]. Consider, for example the Booch OOD method. It is well documented that the Booch method possesses an expressive design notation used to convey its rich object model. However, the Booch method does not provide adequate mechanisms to perform OOA. What it lacks at the analysis level, educators replace with other methods that are better suited to OOA such as Wirfs-Brock [Wi90] or CRC [Na95].

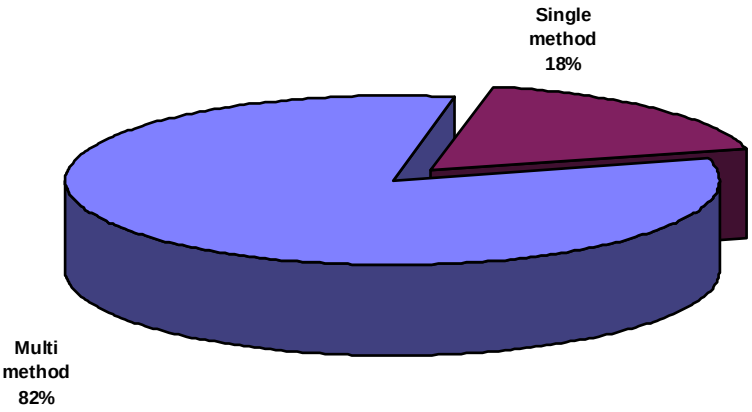


Figure 3—3 : Method mixing at OOPSLA

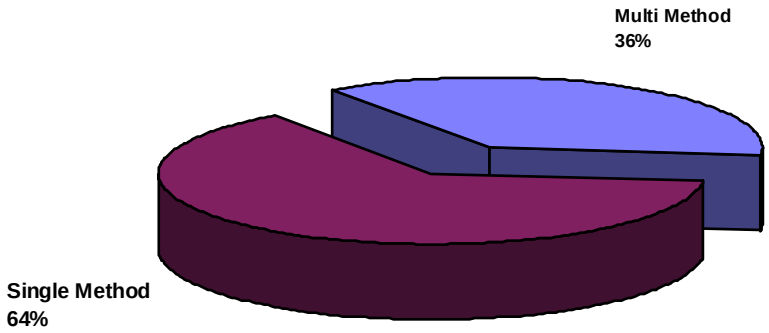


Figure 3—4 : Method mixing in the UK

Method mixing involves assembling the strongest conceptual method parts from the available design processes to create a hybrid method that is synergistically superior: a teaching method. Our surveys suggest that international OO courses (see Figure 3—3) exhibit a greater tendency to employ a multi-method approach to OOD than those in the UK (seeFigure 3—4).

3.3.2 Implementation Language

Figure 3—5 and Figure 3—6 illustrate that the majority of educators are using C++ as their primary OO programming language (OOPL). Of those courses that do not, the majority of them apply C++ as their secondary OOPL.

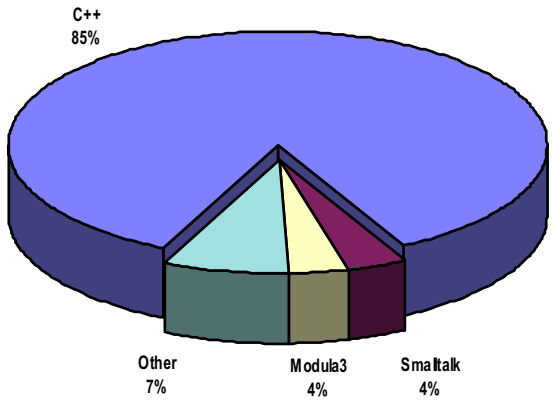


Figure 3—5 : OOPL used in UK OO curricula

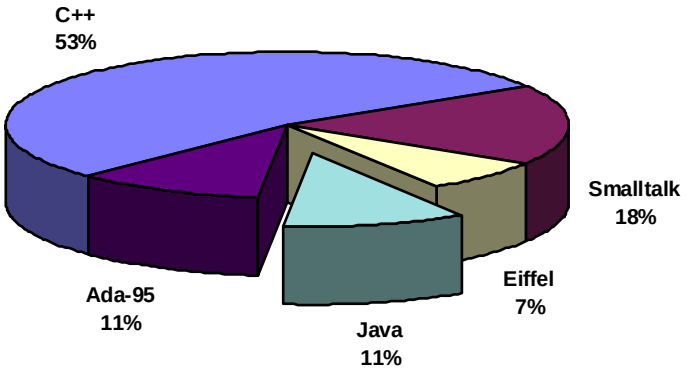


Figure 3—6 : OOPL used in OO curricula by OOPSLA respondents

Reasons for C++'s wide-spread adoption within teaching institutions include: student demands for better job prospects; availability of cheap (free) C++ resources such as compilers, environments and public domain libraries; voluminous amounts of C++ literature; no shortage of staff with C++ knowledge.

However, C++ appears not to be the language of choice, but of use. Survey respondents often quoted its unsuitability as a teaching language for object technology preferring purer OOPLs such as Eiffel and Smalltalk. Educators

believe that the multi-paradigm nature of C++ prevents learners from embracing objects and readily permits them to revert to a procedural style of thinking. Despite the rabid criticism C++ received in the surveys, when asked what their preferred OOPL was, Figure 3—7 depicts C++ as the clear language of *choice* amongst educators. Nevertheless, the appearance of Java as an alternative to C++ has been welcomed by educators, who voiced a high likelihood of Java replacing C++ in future OO curricula.

3.3.3 Prior OO experience

The survey respondents noted that all learners brought to their OO courses varying levels of object thinking. They also noted that the majority of learners still approach problem solving with an algorithmic mindset.

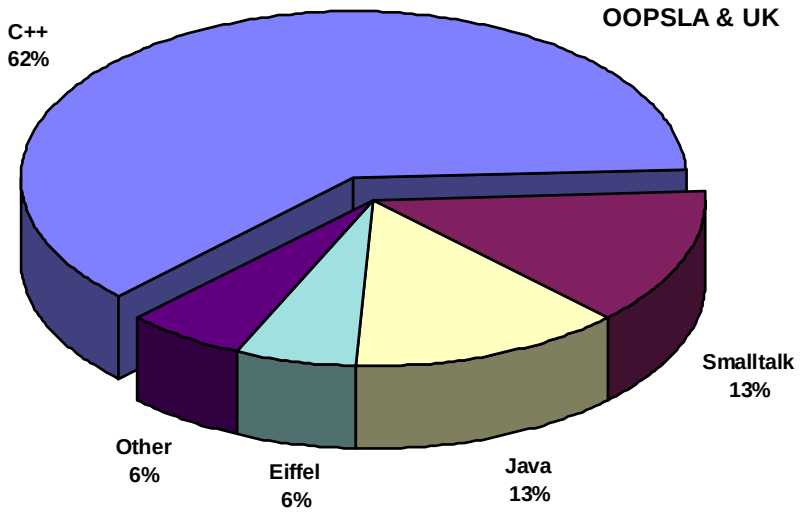


Figure 3—7 : Apportioning of OOPL given the choice

Many of the undergraduate students are taught object-based techniques before embarking upon OO courses. Adherents to this approach feel that a solid grounding in programming with abstract data types and common data structures is invaluable and presents students with a sound footing in things object-oriented or otherwise. Detractors argue that students only have to

unlearn these (futile) structured teachings before grasping the fundamental aspects of object thinking. In the our experience; learners do (should) not unlearn structured techniques, however, it does take noticeably more effort to convert them to object thinking [GH97b]. Gehringer and Manns [GM96] allude to this point in stipulating that structurally minded learners should be presented with *analogies* that highlight differences and/or similarities between the two paradigms. The educator should then draw upon these analogies to guide the learner towards an OO mindset.

3.3.4 Classroom sizes

A typical classroom in the US has between 30-40 students compared with 60-70 students in European classrooms. Although the proportion of time spent *lecturing* during OO courses was similar amongst all educators, the amount of time spent assisting students *doing* design diminished for those educators with larger classroom sizes. It was apparent from the survey results that large classroom sizes first cut into an educator's time to provide hands-on design support and then into the time allotted to assess, evaluate and review student designs. Clearly a means to redress the balance within larger classrooms is required.

3.3.5 Tool Support

The consensus amongst the surveyed respondents was for more affordable tool support. More specifically, tool support was preferred for teaching methods that made use of C++ or Java, rather than OOPLs such as Smalltalk and Eiffel that are inextricably linked to their development environments. Only a few educators believed tool support to be detrimental to the teaching process.

3.3.6 Summary

The surveys demonstrated the pervasive nature of C++, and the Booch and OMT OOD methods within today's teaching institutions. A common complaint amongst educators is that design documents implemented using

the multi-paradigm OOPL C++ does not engender object thinking. Tools and techniques for evaluating design documents implemented in C++, or otherwise, must therefore accentuate their inherent object-oriented properties. Furthermore, these tools should embody a generic class model that can be used by a number of different OOD methods. This would facilitate tool integration into teaching processes that use these well-documented OOD methods and those that employ the increasingly popular multi-method approach to OOD.

Increasing classroom sizes and unmitigated teaching resources highlighted the need for automated design assistance. Educators supported the use of tools for second-checking designs and providing quality feedback to learners. In fact, the most interesting and commonly requested tool feature was that its accompanying suite of evaluation techniques be implemented with a view to becoming an environment and not just another CASE tool. This requirement necessitates a minimal level of language and method independence to be exhibited by both the evaluation techniques*and* the tools that support them.

Reflecting upon the results of the two surveys, design evaluation, as portrayed in Figure 3—8, should take place *whilst applying* object technology. By presenting learners with an immediate and interactive form of design feedback, the concepts introduced in lectures can be re-enforced and examples of their application illustrated directly and effectively within the context of the learner's own designs. This way design assessment remains completely independent from the way in which design evaluation is performed. This enables alternative and possibly more formal assessment techniques to be utilised, if needed, without disrupting the design process.

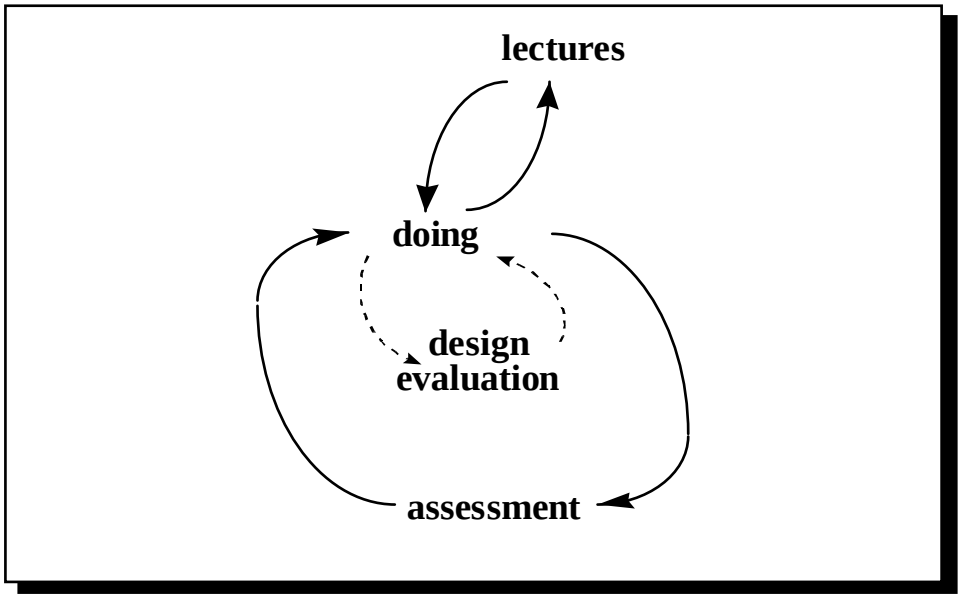


Figure 3—8 : Integrating design evaluation within a teaching method

Chapter 4 documents object-oriented design heuristics and chapter 5 presents a tool that automates their deployment. The next section describes design heuristics in more detail and illustrates how they satisfy the evaluation goals outlined in sections 3.1 and 3.2 and given these process requirements.

3.4 An alternative approach to OOD evaluation

3.4.1 Quality enhancing techniques

A quality enhancing technique (QET) is used to evaluate software products with the aim to improving their quality. Gibbon and Higgins [GH96b], in describing why metrics prove undesirable when *informally* assessing software designs, highlight a number of properties that QETs (pronounced *kets*) should possess. Metrics do not exhibit all the properties listed in Table 3—1 deemed necessary to be an educational QET. Design heuristics have the *potential* to possess all the properties required by a QET. Nevertheless, upon defining what a design heuristic is in section 3.4.2, it is shown in section 3.4.3 that the majority of design heuristics in the literature do not fulfil their potential.

3.4.2 What is a design heuristic?

Heuristics offer insightful information based upon experience that is known to work in practice. They provide a means by which knowledge and experience can be delivered from the expert to the novice. To achieve this goal, heuristics must identify and encapsulate experience; their subsequent application ultimately results in the reuse of this experience.

A heuristic is not a metric, nor vice-versa. A heuristic is a rule of thumb that places its users in the region of what is correct [Bo95]. It is not meant to be exact; in fact, heuristics derive their benefits from this imprecision by providing an informal guide to good and bad practices. In short, metrics define what complexity is and what needs to be done to reduce it, whereas heuristics provide the knowledge and judgement. For example, when determining what to do with a complex class in an unstable inheritance hierarchy, the metric provides a means to determine when a class is complex or when an inheritance hierarchy is unstable. It is the heuristic that makes informed decisions in light of these metric observations, taking into consideration the subtleties of the paradigm and context in which the design elements are set.

Property	Description
Informal	Design expertise is more readily expressed using informal mechanisms. Formalising the interpretation of intelligence, as done by software metrics, is a restrictive process that can result in locking up expertise with the expert. Informal QETs are imprecise and require less effort to integrate design expertise gleaned from experienced developers into the design evaluation process.
Traceable	A well-documented and clear mapping from quality criteria to concepts pertaining to the technology constitutes a traceable QET. For example, traceability in object technology reifies the mapping from maintainability to the OO concepts such as inheritance, polymorphism and encapsulation. Without traceability, a QET knows *how* to enhance design quality without understanding *why* certain OO concepts influence it.

Informative	Learners learn best by example. A QET not only provides insight into quality, but also educates learners in the context of their own designs. Context-driven education sensitises learners to errors and anomalies in their own work, making them more appreciative of the suggested corrective measures.
Simple	This is an obvious but often neglected property of evaluation mechanisms. An unclear QET will be unused, or worse still, misused. Keeping QETs simple, reduces the likelihood of misunderstanding the design problems that they document.
Reusable	For a QET to be reusable it must be applicable within a diverse set of problem domains and at various stages throughout the development life cycle. This requires QETs to be based around design problems that are general enough to be described during the early stages of software design.

Table 3—1 : The properties of a Quality Enhancing Technique

In the context of OOD, a design heuristic focuses upon a single, self-contained design problem. The design heuristic not only imparts knowledge of the documented problem but also the ways in which to solve it. By representing small yet focused design problems, heuristics are applicable within a number of diverse software domains*and* at different phases of the life-cycle. Typically, the problems documented by design heuristics are visible as early as the latter stages of analysis right through to implementation.

3.4.3 Design heuristics

In the literature, heuristics have been widely applied at the programming, design and analysis level, but without any real focus [GH97a]. Both metric developers [YW78][CG90][AC94][CK91] and design methodologists [Bo94] [Ru91][CY91] have actively recognised the role of design heuristics during software development. Some of the more specialist publications on design heuristics [JF88][St92][Fi95] have further underlined their importance during system construction. However, design heuristics still remain largely ill-defined and inextricably linked to the means by which they were first published.

Those heuristics that have been published tend to omit the informative and traceable properties required by an educational QET [GH96b]. By presenting them as guidelines, hints and tips, these publications do not sufficiently document a heuristic's intent, convey its rationale, outline the consequences of its application, describe common contexts in which it is used nor offer common design alternatives to the recurring problems that they identify. Clearly the majority of the design guidelines, hints and tips in the literature are incomplete, documenting only part of the required information necessary to define a design heuristic.

A recent publication by Riel [Ri96] was possibly the first in-depth research effort that focused directly upon the need*and* use of OOD heuristics within software development. The heuristics were presented in an example-driven manner and were comprehensively described. However, there were no common mechanisms for documenting the design heuristics as self-contained, transferable pieces of design expertise, no means by which to describe how they inter-relate and no plan of how to deploy heuristics in support of an informal approach to design evaluation. Furthermore, although a number of Riel's design heuristics were qualitative, those that could be automated were not.

In practice, the most common means by which developers assimilate and transfer design expertise is through guidelines, hints and tips. However these ad-hoc pieces of design knowledge are scattered throughout the literature and lack the level of documentation needed to make them effective within an informal design evaluation process. This chapter has defined the properties necessary for heuristics to be effective during design evaluation. Subsequent chapters comprehensively document designs problems as heuristics, automate their application via tools and evaluate the effectiveness of their employment. Ultimately, our research provides a repository for common design problems encountered during OOD, their possible solution(s) and a tool to automate their deployment.

3.5 Summary

A number of important issues have been raised that serve as the rationale for this thesis. First, design evaluation for informal processes have been restricted to formal techniques pertaining to software metrics. Within certain contexts, notably educational environments, this is neither warranted nor desirable. Second, the results of a nation-wide and international survey substantiate the need for informal OOD evaluation and identify those people who stand to benefit directly from the deliverables of this research.

Finally, design heuristics have the potential to exhibit all the properties needed to support an informal approach to design evaluation. However, the design heuristics in the literature have shown themselves to be incomplete. A means to comprehensively document design heuristics and their inter-relationships remains outstanding.

Chapter 4 presents the OOD heuristic catalogue; a working document [Gi97c]. It provides *all* practitioners that participate in OOD with a *handbook* of common design problems and their well-documented solutions. A prototypical tool described in chapter 5 automates the deployment of the design heuristics described within the catalogue. Chapter 6 reports upon the advantages and limitations of applying these design heuristics. Ongoing research and future directions for design heuristics are discussed in chapter 7 together with our thesis conclusions.

Chapter 4

The design heuristic catalogue

4. Overview

This chapter describes the properties and models that collectively define the object-oriented design heuristic catalogue. Using heuristics, the catalogue documents a number of design problems that developers encounter when building object-oriented software. The process by which design heuristics are discovered, invented, documented, annotated, structured and finally automated within the catalogue, illustrates some of the difficulties involved transferring design experience from the expert to the novice.

It is not the intention of this chapter to summarise the contents of the design heuristic catalogue. An on-line working document [Gi97b] comprehensively describes all design heuristics currently within the catalogue. Instead, this chapter focuses upon the research efforts that led to the production of the catalogue and the implementation of its heuristic models [Gi97c].

4.1 Objectives

In light of the design evaluation problems outlined in the previous chapter, the following objectives and motivation behind the production of the heuristic catalogue were identified:

- the use of design heuristics as a vocabulary for frequently occurring OOD problems;
- the provision of a heuristic form for documenting current and future design problems in a standardised manner;

- to highlight explicit links between maintainability and paradigmatic concepts;
- the timely deployment of heuristics from early design through to implementation;
- a means to structure heuristics by modelling and maintaining their inter-relationships;
- to automate the deployment of design heuristics for the informal evaluation of software designs.

This chapter addresses these objectives. The catalogue requirements in section 4.2 describe how heuristics have been implemented with a view to enhancing design maintainability. We illustrate that a true approach to *design* evaluation requires design heuristics to be applied to interfaces*and* for them to make limited assumptions about a system's implementation details. To this end, design heuristics are applied to an interface-specific, class-based model of an OOD.

Section 4.3 describes how design problems are documented with heuristics. The various sources of design heuristic are reported before illustrating how heuristic forms are used to document design problems in a formal manner.

Section 4.4 outlines the structure of the catalogue. It describes the different types of design heuristic in the catalogue and the categories by which they are grouped. Section 4.5 describes the heuristic models that embody the knowledge and judgement for deciding what constitutes good class structure *and* how it can be achieved using object technology. Finally, section 4.6 reports on the advantages and limitations that arise when automating the design heuristics within the catalogue.

This chapter demonstrates the importance of heuristics as design aids for the informal evaluation of OO software systems. By meeting the catalogue's objectives, we aim to make this approach to design evaluation and the

software expertise that it embodies, available to *all* those that participate in OOD.

4.2 Catalogue requirements

4.2.1 Design for maintainability

Design heuristics strive to enhance design quality. Two of the most desirable qualities in OO software are maintainability and reusability. This research focuses primarily upon constructing maintainable software. Nevertheless, because design maintainability requires the presence of well-encapsulated objects, our research efforts indirectly address issues pertaining to reusability.

Design maintainability has been interpreted in a multitude of ways [CG90][HW93]. We view design maintainability as that: the ability to maintain a design. The harder it is to understand, change or fix a design, the less maintainable it is. Implementing maintainable software involves designing classes that are resilient to modification. In seeking to build robust software architectures Tudball [Tu95] refers to maintainability as the "…containment of change."

The heuristics within the catalogue strive to increase design maintainability by encapsulating both potentially costly and frequently occurring design changes. Design heuristics aim to *contain software changes* by applying the paradigmatic concepts that enforce modularity.

4.2.2 Interface vs. implementation

Chapter 2 reported that the majority of design metrics in the literature cannot be used during design because they rely upon details *inside the class*. The heuristic catalogue adopts a minimalist approach to evaluation by analysing only those software elements available during early design. Only *interface-specific* information contained within OO software is used by heuristics during design evaluation.

In this class-based model of an OOD, a class is defined solely in terms of its externally observable behaviour, instance variables and inheritance relations. Figure 4—1 depicts this interface-specific view of classes that heuristics use to evaluate design documents presented during early design right though to late implementation.

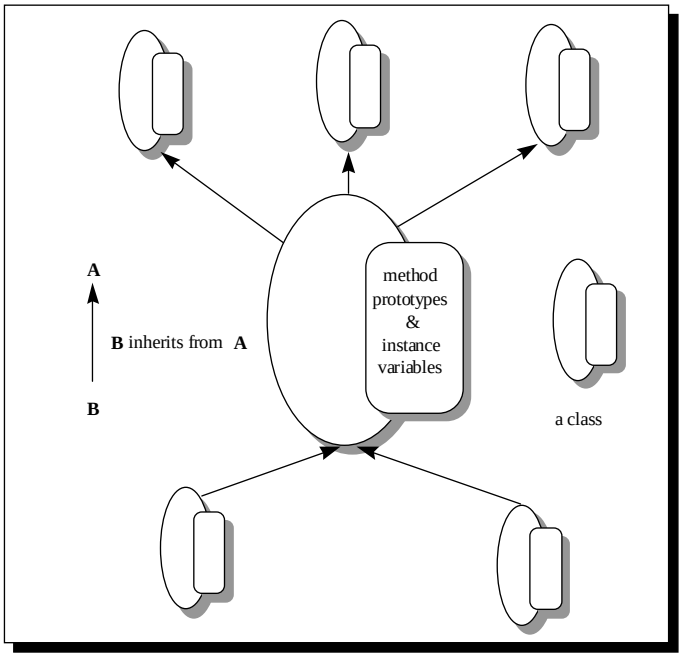


Figure 4—1 : Interface-specific view of an OOD

Without access to class implementation details, the interface-specific view a design cannot provide a complete picture of the object collaborations and messaging within an OO system. But as Rombach [Ro90] demonstrated empirically, comprehensive and informative evaluations of design maintainability *can* be achieved by focusing *purely* upon the interface-specific aspects of software. The catalogue implements interface-specific *design* heuristics that do not make any assumptions about the underlying implementation. In the presence of more detailed design models, more formal techniques of evaluation, such as software metrics, could supplement the feedback gleaned from heuristic evaluation.

4.3 Documenting design heuristics

4.3.1 Source of design heuristics

There are two primary sources of design heuristic [GLH96]: those discovered in the field by paper writing OO consultants and those invented by applying the concepts and principles within a given paradigm. Paper writing OO consultants are experienced system developers who disseminate design expertise gleaned from completed OO projects. Unfortunately only limited degrees of design expertise learnt in an industrial capacity are published in the literature. This is because exceptional design solutions become invaluable company assets that are not public domain. The more promising design heuristics are salvaged from failed projects. However, companies typically prevent these design lessons on doing things*wrong* from being promulgated back to the OO community because of the way in which they were discovered.

The second reported source of design heuristics pertains to extrapolating the rules of the methodology and elementary object technology. In this case, a number of assertions are made as to what exactly constitutes quality classes and their objects. The validity of these assertions is tested against class libraries to ascertain whether they are practised within a working development environment. This is the main source of design heuristics presented in this chapter. Occasionally, other published design heuristics substantiate the assertions made by those in the catalogue. However, it is more likely that through their *use*, a better indication of their ability to evaluate an OOD document can be discerned. Chapter 6 details how effective design heuristics were during software development.

Both sources of design heuristics are important. Those discovered by experienced developers have been informally validated by their use in the field. Those that rely upon the rules of the methodology seek validation through their application and corroboration with independently published design heuristics.

4.3.2 Heuristic forms

A heuristic form is a template that provides a standardised way of documenting recurring design problems and their respective solutions. A heuristic form captures the essence of a design heuristic. Table 4—1 depicts the various fields in a heuristic form and gives an example of an inheritance-based heuristic from the catalogue.

A similar type of form is used to describe a software pattern [Bu97]. What is noticeably different between a pattern and a heuristic form is the presence of the suggestions field in the latter. This field highlights which design solutions can be applied to fix the described problem. These solutions can be as well documented as design patterns, refer to known design patterns, or be less formally stated as ad-hoc guidelines on how to approach the problem.

The terminology used to describe design problems on a heuristic form should be tailored to their target audience. For example, the heuristic form for a learner will be less technical than that of an experienced developer. Automation permits the user to switch between learner and expert modes to receive the appropriate level of design feedback.

However, before a design heuristic can be automated a more precise description of the details recorded on its heuristic form must be defined. Automatable design heuristics are heuristics that encapsulate formulae used to detect their presence within a software design. By encapsulating notions of formality, automatable heuristics are divorcing the design problems they document from the means by which they are detected. This enables automatable design heuristics to enhance their problem detection formulae without recourse, provided that they continue to support the design problems documented on their respective forms. For these reasons, heuristic forms do not have a field describing how they are objectively specified. Furthermore, aspects of automation are secondary, and therefore superfluous to those applying design heuristics during software evaluation [GH97a].

Instead, these problem detection formulae are documented within the TOAD system [Gi97c].

Field	Field Description	Example
Name:	description of the heuristic	Beware of over-generalising the parent class.
Intent:	aim of the heuristic	To ensure that inheritance hierarchies and their levels of abstraction are well formed.
Rationale:	outline of the problem	For families of semantically similar classes, over-generalisation on the parent class begs the question: what commonality does every child class have with its parent *and* with its siblings? A wide class is one that has numerous child classes. A wide class that has a relatively small interface may present an interface relevant to all its descendent classes. On the other hand, a wide class may embody overlapping abstractions that also increases the number of (potential) child classes. This heuristic signals the possibly of these two cases.
Consequences	things to be aware of when applying the heuristics	Mixin classes and applications constructed within a single inheritance hierarchy tend to breach this heuristic. In some cases, more so in the former, parent classes are being used to hold together families of semantically disparate classes. For wide classes that propagate small interfaces down the inheritance hierarchy, the chance of being labelled as an over-generalised class increases. The factory method design pattern [GHJ+94] is an example of this.
Contextual Information:	important design specific details that affect a heuristic's behaviour	The size of the object interface; large ones signal overlapping abstractions where small interfaces indicate small, possibly incomplete, abstractions. Position in the hierarchy; acceptable over-generalisation takes place at the top of an inheritance hierarchy within generalised abstract classes and at the bottom where concrete classes specialise at the leaf nodes.
Inter-heuristic dependencies:	heuristics that have a direct bearing on this heuristic	Limit the number of methods per class. Limit the messages that an object can receive. Strive to make abstract classes type definitions.

		The greater the number of child classes, the higher the likelihood that the parent class should be abstract. In particular, this targets classes assuming an intermediary position in the hierarchy.
Suggestions:	things to be considered in light of the heuristic being breached	If inheriting data, ensure inheritance is not being used to model aggregation. If inheriting methods and data, consider using delegation. If the parent interface is small, consider the possibility that it is too general and that another level of abstraction is required to categorise its child classes. If the parent interface is large, consider the possibility that the abstraction is too large. Is every child class conforming to the interface of the parent class *and* supporting the behaviour it distributes? If not, the parent class may need to be fragmented.

Table 4—1 : A heuristic form

4.4 Catalogue structure

The catalogue is composed of a number of heuristics that document design problems in a standardised manner using forms. These heuristics aim to improve design maintainability by evaluating the interface-specific properties in OO software. This section describes how the catalogue structures the contained heuristics according to the relationships that exists between classes.

The catalogue is not an ad-hoc repository for design heuristics. It is a highly cohesive, inter-related group of problem and solution sets. The catalogue is responsible for classifying the various *types* of design heuristics and managing their inter-dependencies by grouping them into *conceptual* and *quality* categories. This section describes each of these catalogue functions.

4.4.1 Types of heuristic

There are two main types of design heuristic:*class* and *relationship*.

4.4.1.1 Class heuristics

Class heuristics are the simplest type of heuristic that act upon individual classes. They evaluate class properties pertaining to method complexity, size, behavioural content, encapsulation, mechanisms that breach information

hiding, centralised control, and so on. They are purposefully simple and typically used in conjunction with relationship heuristics.

4.4.1.2 Relationship heuristics

Relationship heuristics are defined for the *inheritance*, *aggregation* and *using* class relationships. A relationship heuristic acts upon a group of classes, or *class group*, that participates in a specific class relationship. A *head class* is representative of a class group such that all relationship heuristics are directed at the head classes. For example, the head class in an inheritance relationship is the root of a class hierarchy and the members of the class group are its descendent classes. Inheritance-based heuristics applied to a root class, `R`, will be evaluating `R`'s class group.

The catalogue implements a relationship model for every supported class relationship. These models define how classes inter-relate within a given class relationship by describing what the (non) essential paradigmatic concepts are and how they contribute to building maintainable class groups. Hence, the relationship model embodies explicit links between quality and the paradigmatic concepts that strive to achieve it. Relationship heuristics use this model as a reference point for defining well-structured class groups. Section 4.5 describes the class and relationship heuristic models that the catalogue's heuristics are based upon.

Table 4—2 lists the design heuristics that currently make up the catalogue. Each heuristic has a unique *identification code*. The first letter documents the *type* of heuristic as either class (`C`) or relationship (`R`). The second letter in the identification code indicates which *model* the heuristic is based upon: `I` for inheritance, `A` for aggregation, `U` for using and `C` for the class model. However, like classes, heuristics rarely act in isolation. Given the heuristics in Table 4—2, the next section describes how heuristics are grouped together to evaluate higher level design problems encountered by users performing OOD.

4.4.2 Catalogue categories

4.4.2.1 Concept categories

Individual heuristics touch upon a number of different design concepts when documenting a single design problem. However, when evaluating an OOD, designers want feedback upon how well they are employing encapsulation, feedback upon whether their inheritance hierarchies are well formed, some indication on the (de)centralised nature of their objects, and so on. A concept category groups heuristics together to provide feedback on a particular design principle. Table 4—3 summarises the resultant concept categories applied on a per class basis by designers evaluating their design documents.

Heuristic Id	Heuristic Description
CC1	Limit the number of methods per class
CC2	Limit the number of attributes per class
CC3	Limit the messages that an object can receive
CC4	Minimise complex methods
CC5	Limit enabling mechanisms that breach encapsulation
CC6	Hide all implementation details
CU1	Limit the number of collaborating classes
CU2	Restrict the visibility of interface collaborators
RU1	Identify and stabilise common place interface collaborators
CA1	The aggregate should limit the number of aggregated
CA2	Restrict access to aggregated by clients
RA1	Aggregation hierarchies should not be too deep
RA2	The leaf nodes in an aggregation hierarchy should be small, reusable and simple
RA3	Stability should descend the hierarchy from rich aggregates to their building blocks
CI1	Limit the use of multiple inheritance
CI2	Prevent over-generalisation of the parent class
RI1	The inheritance hierarchy should not be too deep
RI2	The root of all inheritance hierarchies should be abstract

RI3	For deep hierarchies upper classes should be type definitions
RI4	Minimise breaks in the type/class hierarchy
RI5	Strive to make as many intermediary nodes as possible abstract.
RI6	Stability should ascend the inheritance hierarchy
RI7	Inheritance is a specialisation hierarchy

Table 4—2 : The design heuristic catalogue

Experience teaching OOD has shown that these concept categories are the ones that inexperienced designers typically require feedback on. Grouping design heuristics into categories provides a uniform way of representing higher level design problems relating directly to essential paradigmatic concepts.

concept category	*what the category reports on*
inheritance	the use of abstract classes, the shape of the hierarchy, inheriting from concrete classes and issues of type vs. class.
encapsulation	the use of non-private data and mechanisms used to breach encapsulation.
collaborators	the behavioural content of the class, whether it centralises the design's behaviour and if it possess a high fan-out.
bandwidth	if low or high, is class cohesive, whether it is abstract or concrete and what is it's position in the inheritance hierarchy.
implementation	size of the implementation, contributions by user-defined types and whether its attributes well encapsulated.

Table 4—3 : Concept categories

4.4.2.2 Quality categories

These categories issue design reports according to important software qualities. Currently, the catalogue supports maintainability. This quality category produces a maintainability report based upon the frequency, type and context in which heuristics are breached within OO software. A quality category has an intimate knowledge of the inter-relationships and

dependencies between design heuristics. Using this knowledge, a quality category can report on the overall quality of a system based upon the absence and/or presence of breached design heuristics. Chapter 5 describes how these reports are generated and the results of their application are presented in chapter 6.

Heuristics have been used to document common design problems, providing a vocabulary for developers to communicate difficulties that arise during design evaluation in a uniform way. Heuristic forms are used to capture and comprehensively document these common design problems. The catalogue then structures the heuristics into a highly cohesive collection of design expertise. The next section describes the models upon which these design heuristics are founded.

4.5 The heuristic models

The heuristic catalogue [Gi97b] is a working document that provides a comprehensive description of every heuristic listed in Table 4—2. This section presents the models upon which design heuristics are based. Heuristic models draw upon knowledge gleaned from a number of pragmatic and theoretical sources regarding OOD evaluation. They represent an in-depth understanding of object technology learnt by constructing *and* measuring OO software systems.

The resultant heuristic models are informal. Figure 4—2 depicts how these models have built upon software measurement research, revisited elementary object technology and taken stock of the practical applications of objects delivered in the form of design hints. Heuristic models balance the theory of building maintainable objects documented by software metrics against how the objects are actually used to construct OO systems.

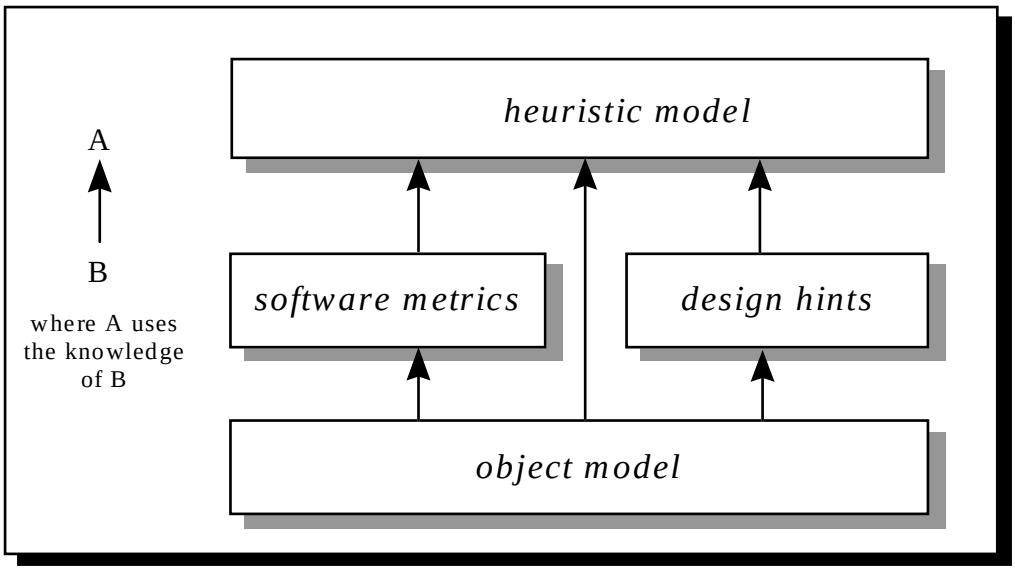


Figure 4—2 : The heuristic models' sources of design knowledge

Established design methods formalise objects to clarify what is and what is not an object. Using an object model, software metrics define what constitutes high quality objects and provides mechanisms to identify and enhance them. Chapter 3 reported that software metrics are neither readily available nor legible to *all* developers during *design*. In reality, hints, guidelines and tips are the usual methods employed during software design to convey experience from one developer to another. The heuristic model takes what software metrics believe to be quality objects and augments this knowledge with the more practical means of disseminating design expertise. The resultant heuristic models provides a pragmatic set of guidelines for building maintainable software objects based upon the well-founded concepts derived from measurement research.

The catalogue embodies four distinct yet collaborative heuristics models; one for each type of heuristic in the catalogue. The following sections describe the class, aggregation, using and inheritance heuristic models.

4.5.1 The class model

An object model informs the designer that a class is an abstraction that encapsulates data and exports well-defined behaviour. However, a class designer must reason carefully between a number of conflicting design

decisions that will ultimately decide how a class will be (re)used, to what extent and for how long. The class model considers every class in isolation and presents the designer with the essential characteristics for producing maintainable objects.

A class is responsible for producing well-encapsulated objects . This is achieved by restricting direct access to an object's implementation (`CC6`) and enforcing restrictions upon doing it indirectly [Wi87]. The use of enabling mechanisms to access an object's implementation (e.g. friends in `C++`) should also be restricted (`CC5`).

Export cohesive interfaces that support simple, observable behaviours . Cohesion has been defined as the amount of use that methods make of their instance variables [CK91][EKS94] or as the similarity of methods within a class [CL93] [MDS93]. As the number of class methods increases, the less likely that either of these conditions will hold (`CC1`, `CC3`). Furthermore, class behaviour should be defined in terms of a collection of composable primitive methods (`CC6`). This will reduce the effort required to maintain the class and permit individual methods to be reused in more diverse contexts.

Populate designs with single-minded, expressive, yet manageable objects. Occasionally, objects must embody complex state information. However, it is not good practice for an object to model complex states with large collections of instance variables [Ta93]. Object state is an aggregate that is determined by the consolidated values of all instance variables. The more states an object can assume, the more roles that its designers must cater for, which in turn increases the complexity of object interactions within a system.

By reducing the number of instance variables within a class, the designer can control the number of roles an object may assume. Furthermore, reducing the number of instance variables in a class minimises the degree of co-ordination required by an object to *correctly* maintain its complex state information.

4.5.2 The aggregation model

Aggregation expresses the *physical* containment of one class by another. It aims to assemble objects into richer and more useful wholes. However, aggregates still need to be maintained which involves modifying their contents. The heuristic model for aggregation moderates the black box view of aggregates against their realistic need for maintenance. To this end, aggregation hierarchies must structure and control the way in which aggregates *evolve*.

Manage the complexity of an aggregation hierarchy by shielding clients from its descendent classes. Consider the aggregation hierarchy depicted inFigure 4—3. Ideally, clients collaborating with `House` objects will do so without knowledge of its descendent classes. Martin [Ma94] believes that this way the complexity of an aggregation hierarchy is well managed.

However, designers building a maintainable aggregation hierarchy cannot ignore its shape or contents. The hierarchy for a rich aggregate should be both narrow (`CA1`) and deep (`RA1`). Narrow hierarchies ensure that any one aggregate does not have to manage a large number parts and deep hierarchies illustrate the reuse of the lower level aggregates.

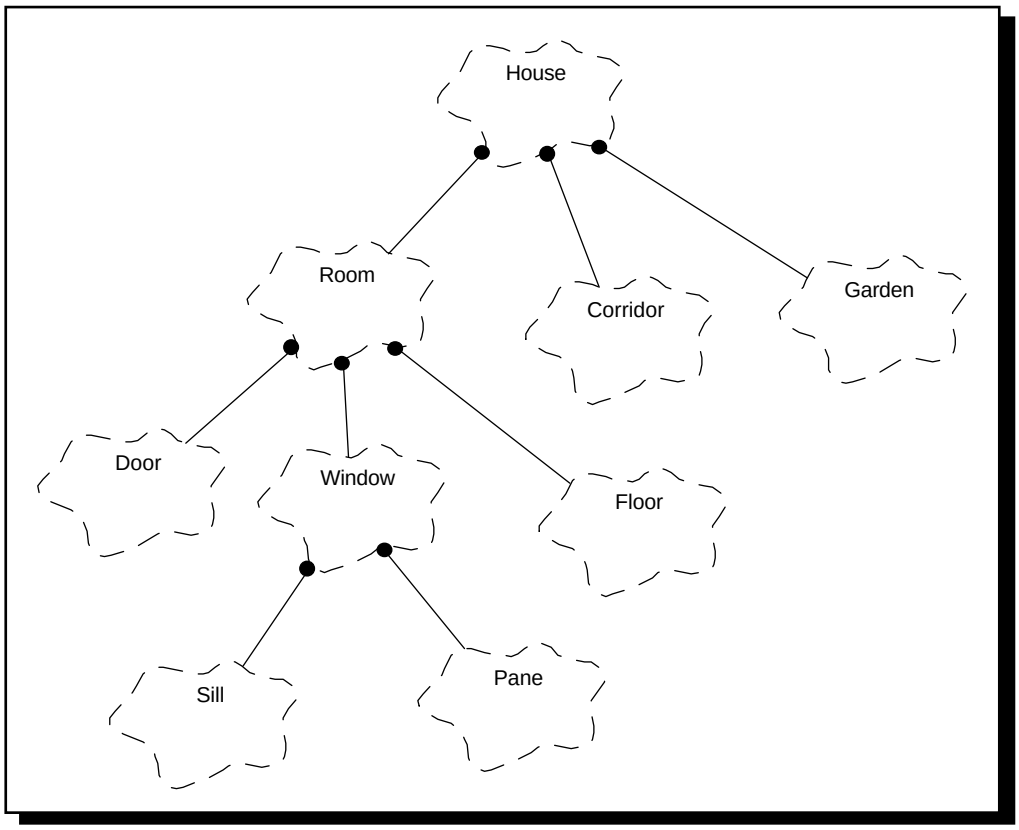


Figure 4—3 : Aggregation hierarchy for `House`

Stability should descend the aggregation hierarchy. The leaf nodes in an aggregation hierarchy represent the building blocks for rich aggregates. They should be small and simple if they are to be reused in other aggregation hierarchies *and* export primitive behaviours to the uppermost aggregates (RA2).

The stability of an aggregation hierarchy relies upon the stability of its leaf nodes and the extent to which its uppermost aggregates have encapsulated them. Re-consider the aggregate `House` in Figure 4—3. If both `Pane` and `Sill` were unstable, then changes made to them could effect modifications on `Window`, which in turn *could* propagate changes up through the aggregation hierarchy to `House` and its clients. To reduce the likelihood of propagating changes, stability should descend the aggregation hierarchy (RA3). However, if `Window` had taken precautionary steps to encapsulate `Sill` and `Pane`, the fact that they were unstable would be less of an issue to `Room` and its ascendent classes. Hence, if aggregates restrict clients from

accessing their aggregated parts (CA2) then propagating changes can be managed if and when they occur. To conclude, *stable* lower level aggregates that are *encapsulated* by upper level aggregates increase the maintainability of an aggregation hierarchy.

4.5.3 The using model

Where aggregation describes how classes are composed of other classes, the *using* relationship relates more to the interactions between classes at the interface level. By modelling the inherent client/server relationships that exists between classes, design heuristics aim to create a collection of well-mannered, decentralised objects that collaborate in small groups to perform higher level system tasks.

Produce decentralised objects . Objects collaborate with other objects in order to fulfil their responsibilities. However, an object can centralise system intelligence by taking a monolithic approach to object collaboration. Typically, this is characterised by a class supporting a large number of interface collaborators in its methods (CU1). In addition, the visibility of these interface collaborators has a direct bearing on the maintainability of the class (CU2). The greater the number of public interface collaborators, P, in a class C, the higher the probability that clients to C may need to be change if any of P are modified or removed.

Identify key objects and ensure that they are stable. The overall stability of a system is largely dependent upon the stability of its most *common* and *major* service-providing classes. The pre-emptive use of heuristics to identify pervasive service providers in a design and to stabilise them is an important task during software evaluation (RU1), for changes made to these classes ripple through a design architecture at a great cost to the system.

The using model places emphasis on the server to export reliable behaviours to the system. This ensures that the client is always in receipt of a stable set of services. The stability of client classes is defined in terms of its interface collaborators. The model looks to enhance the stability of a client's interface

collaborators by ensuring that they are loosely coupled, well-encapsulated, small and simple. The greater the number of (public) interface collaborators a class has, the more emphasis is placed upon stabilising them.

4.5.4 The inheritance model

The previous heuristic models are relatively small in comparison to the inheritance model. This is because the aggregation, using and class models, and their usage, are relatively well-defined in the literature [Bo94][Ri96] [Wi90]. However, inheritance presents a plethora of opportunities for practitioners to misuse it within their designs. The heuristic model captures both the good and bad applications of inheritance to guide designers towards building maintainable class hierarchies.

Inheritance is not a singleton. It embodies a number of*conflicting* techniques for building reusable, flexible and extensible design architectures. However, these *distinct* mechanisms for constructing *different* types of class hierarchy have *specific* objectives. Table 4—4 lists a number of these inheritance mechanisms which are typically applied in a synergistic manner. Unfortunately, without directly addressing the *independent* roles of inheritance, they will continue to be misunderstood and misapplied by the majority of class designers. The inheritance heuristic model strives to bring to the fore the different uses of inheritance captured in the form of common design problems and typical misapplications of them.

4.5.4.1 Terminology and objectives

Inheritance creates families of classes related by behaviour, data and/or interface. **Implementation inheritance** permits abstractions with similar representations to share data by reusing code within their ascendent classes. **Interface inheritance** reuses class protocol. A class specifies an (abstract) interface that is adhered to by all its descendent classes. Clients of a well-formed inheritance hierarchy are shielded from the complexities of concrete descendent classes by manipulating polymorphic objects through (abstract) interfaces defined by their ascendent classes. This separation of

implementation from interface is vital within OO systems so that objects can have:

- multiple interfaces for the same implementation;
- multiple implementations for the same interface.

Conceptual abstraction ; the parent class is abstract and provides an outline for concrete child classes to fill in, e.g. a dog is a mammal.
Specialisation ; involves refining a concept. The child class models a particular subset of all parent class objects, e.g. a manager is an employee.
Extension ; involves adding new behaviours to a child class while preserving parent class behaviours, e.g. a coloured point is a point.
Realisation ; a concrete child class provides implementation for an abstract parent class. It is possible to have multiple child classes each providing a different representation of an abstraction, e.g. a queue modelled using a linked list is a queue.
Mixin inheritance ; a child class inherits a set of behaviours disjoint from the abstraction that is represents in order to display a certain type of behaviour, e.g. a persistent database entry is a persistent object.

Table 4—4 : Some uses of inheritance [Ga97]

To enforce this level of separation requires designers to clearly differentiate between **class** and **type**:

- a *class* defines only the implementation of a set of methods and a data structure for object instances;
- a *type* specifies the interface that an object must provide *and* the behaviour that the object must exhibit.

In Smalltalk, types are implied and there are no constructs in the language for explicitly specifying them. The C++ language does not support the distinction between class and type. However, types can be modelled with abstract classes composed solely of pure virtual member functions that have no data members; a **type definition** In Java, explicit support for type

definitions is provided with the `Interface` language construct. In all three language models, behaviour conformance is implied, such that interface inheritance can only guarantee conformance to *interface*. That is `Athlete` and `Payroll` types may both be inclined to `Run`, but clearly their behaviours are incompatible and there is no language support to enforce this.

Type extension, often referred to as interface inheritance, permits designers to organise hierarchies by their interfaces. As shown in Figure 4—4 to enforce a clear separation of interface from implementation and manage the complexity of inheritance hierarchies, clients should manipulate objects through types [GHJ+94]. Inheritance heuristics take account of these concerns in trying to guide designers towards constructing flexible yet resilient class hierarchies.

4.5.4.2 The model

Create families of classes, create families of abstractions . The root class of an inheritance hierarchy specifies behaviour and interface for its descendent classes. The greater the size of the inheritance hierarchy the more important it is for the root to be abstract (`RI2`) and for deeper hierarchies for it to be a type definition (`RI3`). A concrete root binds its descendent classes to a particular representation which reduces the flexibility of the hierarchy by impeding the separation of interface from implementation.

Deep and narrow vs. broad and shallow hierarchies; a compromise. It is well-documented that narrow and deep class hierarchies result from making good reuse of existing classes [Ri96][Fi95]. This model argues whether deep inheritance hierarchies are warranted for all types of software. For example, consider a general reuse library whose class hierarchies are narrow and deep. Before clients can reuse these abstractions in their own systems, they must understand them. The deeper the hierarchy, the more effort is required to assimilate it *and* the design policies that it supports. Furthermore, the resultant system will have even deeper hierarchies that will be even more difficult to maintain.

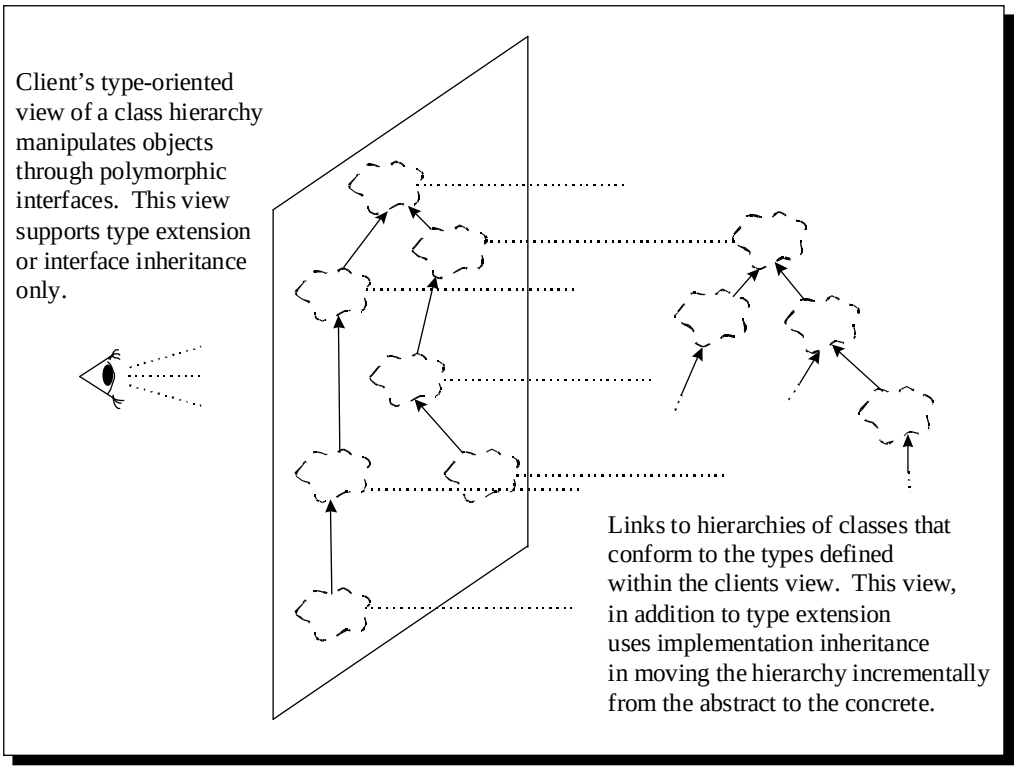


Figure 4—4 : A type-oriented view of an inheritance hierarchy

Coggins [Co90] suggests that reuse libraries should be broad and shallow to facilitate reuse. We contend that inheritance hierarchies should not *strive* to become deep and that depth is related more to the type of application that the class hierarchy is a part of. Inheritance heuristics recognise three types of software application: general reuse libraries; application frameworks; and complete working systems that employ combinations of the other two. Although the optimal depth of inheritance cannot be objectively determined, thresholds based upon the intended target domain and application type can be used to indicate *potentially* erroneous class hierarchies. This permits the developers to re-evaluate the hierarchy and determine whether it is too deep, which as Wang and Wang [WW94] report, is when the depth of hierarchy impedes reuse.

Broad hierarchies are also not beyond reproach. Upon first inspection, a parent class with a multitude of child classes may seemingly be making good use of inheritance to distribute commonality. However, if the child class has

little in common with its siblings and/or its parent class, then the distribution of commonality throughout the hierarchy may be at fault. Invariably, this results in class hierarchies whose levels of abstraction are skewed, indicated by the presence of over-generalised parent classes (CI2) or deep inheritance hierarchies.

Inheritance hierarchies are specialisation hierarchies. In support of the type-oriented client view of a class hierarchy illustrated in Figure 4—4 an inheritance hierarchy should be a specialisation hierarchy (RI7). Moving from the root class to its leaf nodes, behaviour, data and interface are added *incrementally*. Specialisation sees an inheritance hierarchy gradually progressing from the abstract to the concrete. Enforcing this model requires making as many intermediary nodes as possible abstract (RI5). More specifically, the uppermost classes are type definitions, progressing to abstract classes that embody increasing amounts of implementation, and finally down to concrete classes at the leaf nodes.

Although this approach to inheritance introduces extra complexity on behalf of the class designer, it reduces complexity from the clients' perspective as they only need to be aware of the type-oriented view. Even so, these additional complexities better equip class designers to carry changes to the inheritance hierarchy provided that their modifications conform to the client's type-oriented view of it.

In practice, the extra costs involved in constructing type-oriented class hierarchies are manageable [SD97]. However, the flexible and extensible nature of a class hierarchy requires particular attention to be paid to the ways in which implementation inheritance is employed. Clearly it is not practical to dismiss the benefits of code reuse delivered by implementation inheritance. Nevertheless, these benefits need to be balanced against the long term goals of a class hierarchy that interface inheritance can bring.

Non-public inheritance is a common mechanism for code reuse. However, it creates breaks in the type/class hierarchy preventing clients from

manipulating objects polymorphically through type references (RI4). Substitutability [Li88][Ma96] amongst descendent objects shields clients from the complexities of an inheritance hierarchy and permits types to support an unbounded number of implementations. Removing substitutability from an inheritance hierarchy severely restricts its ability to adapt in the face of changing requirements.

Aggregation can be used in place of non-public inheritance where inheritance is employed simply to reuse the internal representation of ascendent classes. In circumstances where implementation inheritance involves methods, delegation can be used as a design alternative. In all instances, the heuristic model dictates that these different approaches to implementation inheritance be used when the designer is trying to realise the full benefits of code reuse. The constructive use of aggregation and delegation will ensure a certain level of interface and implementation separation within a class hierarchy whilst gaining some of the benefits of code reuse. However, the model clearly stipulates that the upper levels of (large) class hierarchies should distribute type-oriented information and that implementation inheritance be employed progressively more upon descending an inheritance tree.

The heuristic models presented in this section described how heuristics attempt to impose their view of maintainable design architecture upon the class designer. The next section outlines the automatable design heuristic catalogue that serves to objectively deploy heuristics during OOD.

4.6 The automatable design heuristic catalogue

Figure 4—5 illustrates that the three phases of design evaluation are:

- problem detection;
- problem feedback;
- solution guidance.

The first phase deploys heuristics to establish where in the architecture *potential* problems reside. Upon detecting a problem, design evaluation reasons about why a particular heuristic has been breached. If during this problem feedback phase the evaluation process concludes that the breached heuristic is actually a design problem, solution guidance is provided. The remainder of this section expounds the roles of problem detection, problem feedback and solution guidance when informally evaluating OOD documents with heuristics.

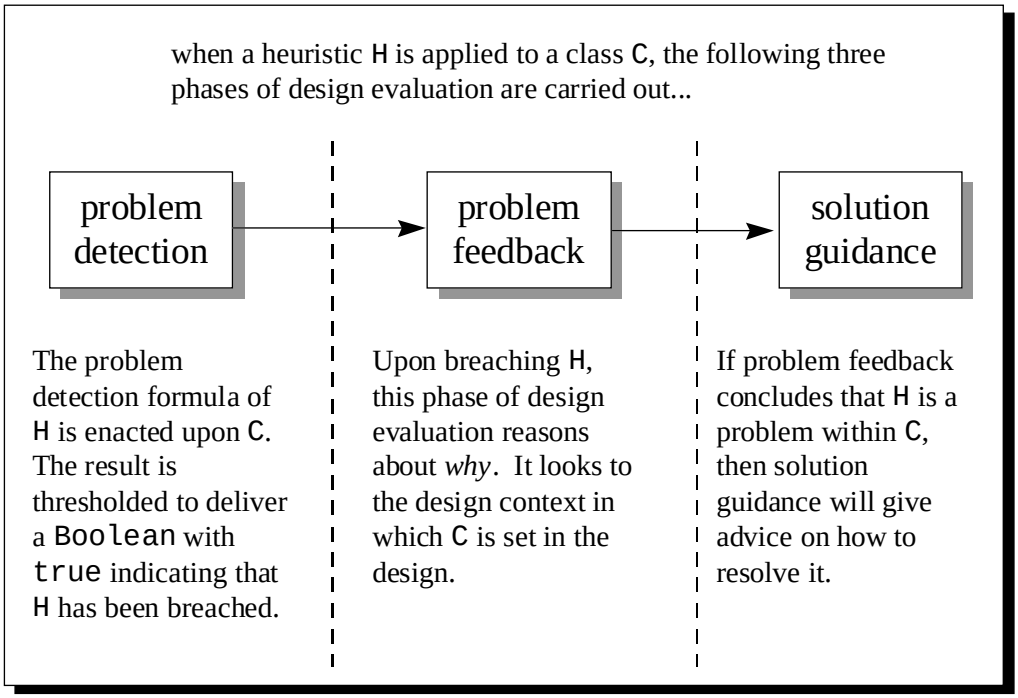


Figure 4—5 : Informal design evaluation using heuristics

4.6.1 Problem detection

Every heuristic has a mathematical expression that is interpreted during the problem detection phase of design evaluation. The delivered heuristic result is thresholded, presenting a `Boolean` value that determines whether it has been breached. Thresholds for most design heuristics have an upper *and* lower bound. For example, the class heuristic to *limit the bandwidth of an object* could have an upper bound of 15 and a lower bound of 3. The upper

bound conveys to the designer that the class is behaviour-rich and the lower bound suggests that it is behaviour-poor.

This kind of problem detection is static. It represents design evaluation *without context*. This coarse level of evaluation assumes that *all* classes are equal and that *all* classes are used in exactly the same manner. Such sweeping generalisations of classes and their uses are made by a number of metric suites when assessing software designs [CL93][CK94][HW93]. Heuristics add more to the evaluation process by reasoning *why*.

The environment in which a class is set provides vital information regarding why a problem might have occurred. The environment may be privy to details of a known design anomaly `D`, to suggest that `D` is not really a design problem at all. Instead, `D` may result from a common/idiomatic arrangement of classes within certain a design context; a *pattern*. Breached heuristics use problem feedback to aid designers to reason about identified design anomalies by taking into consideration the context in which classes are implemented.

4.6.2 Problem feedback

Consider breaching the lower bound of the class bandwidth heuristic which suggests to the designer that a class `C` is behaviour poor. The problem feedback for this heuristic knows that not all classes with a low message bandwidth are detrimental to a design. Problem feedback is charged with taking a finer grained approach to evaluation by trying to establish the design context of `C`. This involves examining paradigmatic concepts such as inheritance and modularity and reasoning why a low object bandwidth on `C` would be applicable within a given design context.

For example, what if `C` is a type definition or an abstract class? What if `C` is the root class in an inheritance hierarchy? Clearly, in these cases `C` is being used to distribute behaviour incrementally throughout the class hierarchy. `C` might be a concrete root for numerous classes in a design indicating that it is being used as a mixin class. Problem feedback on this occasion would outline

the appropriate advantages and limitations of this approach. Alternatively, `C` might not be part of an inheritance hierarchy but information regarding its presence within using graphs and/or aggregation hierarchies could present yet more reasons for its low bandwidth [Gi97b].

Problem feedback may also call upon software metrics before passing judgement. Supporting metrics could determine whether `C` is composed of complex and/or unstable interface collaborators. This would suggest that `C` was performing complex operations within a small number of methods. Supporting metrics provide valuable input on the *properties* of a class that are used by the heuristic to reason more clearly about a given design problem.

Problem feedback takes stock of the common and largely learned ways in which classes are used within an OOD document. Problem detection is an important prerequisite to problem feedback, informing the designer that a class(es) have been implemented in an unexpected way. Problem feedback builds upon this potential design problem by comprehensively assimilating its design context. Upon identifying the problem and fully diagnosing it, the evaluation process sets about instituting a solution.

4.6.3 Solution guidance

Every heuristic has a generic and specific solution. Generic solutions can be documented as design patterns or less formally with ad-hoc descriptions that demonstrate ways in which to approach a given design problem. Specific solutions use generic solutions as templates. They reify these generic descriptions with elements taken directly from the design under evaluation.

Consider the inheritance heuristic that stipulates that the majority of intermediary nodes should be abstract. This implies that child classes should not inherit from concrete parents [Me95][St95]. As a solution, the designer could be presented with an ad-hoc description of how and why they should promote commonality up into an abstract parent class for use by these concrete classes. However, as shown in Figure 4—6, this does not go as far as depicting the original arrangement of classes in the design *and* how they

can be revised. By formalising the solution as a design pattern, the pattern becomes reusable and another form of documentation within the automatable heuristic catalogue.

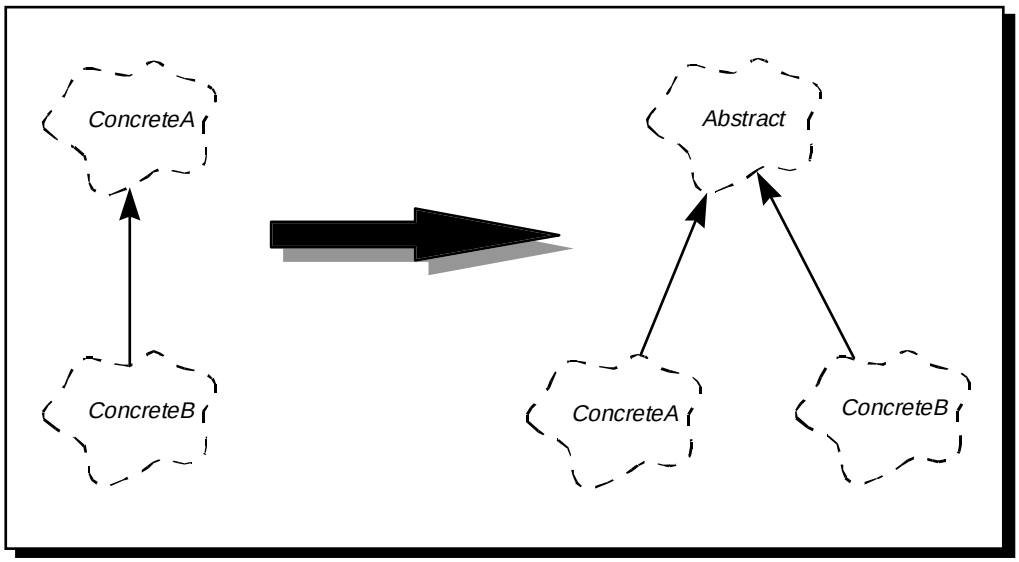


Figure 4—6 : Do not inherit from concrete classes pattern

Specific solutions are specialised versions of generic solutions. They assimilate contextual information derived from problem feedback to present a solution annotated with the classes, methods and instance variables taken directly from the design under evaluation. Every class has a unique (specific) solution that is based upon on their generic solution. Specific solutions are therefore *familiar* to the designer, increasing their awareness of the problems *they* encounter during the evaluation of*their* OOD documents.

Chapter 6 and Appendix B demonstrate how heuristics are used throughout design evaluation process. The next section describes how a number of heuristics in the catalogue have been prioritised to provide a structured approach to resolving problems found during evaluation.

4.6.4 Heuristic Priorities

Heuristic priorities provide an effective mechanism for designers to manage and schedule how they approach fixing design problems presented during evaluation. Ultimately breached heuristics are ordered according to their perceived impact on a design's maintainability. This section outlines the basis upon which heuristics are assigned their priority values and how they are applied.

Priority values are composed of two elements such that:

```
heuristic priority = static priority + dynamic priority
```

Table 4—5 presents a nominal ordering of the class-based heuristics within the catalogue. These heuristics have been categorised and assigned a *static priority* according to the concepts that they address. Level 4 heuristics address encapsulation, level 3 coupling, level 2 cohesion/size and level 1 heuristics do not directly address a specific concept. This partial ordering of concepts was reported in chapter 2 based upon a large body of metric research. The same ordering has been used to assign static priorities to class-based heuristics.

A dynamic priority is used to increase the importance placed upon a heuristic, given the design context in which it was breached. Consider a class `C` that breaches an encapsulation-based heuristic. The dynamic priority of `C` would promote the importance of this heuristic if, for example, `C` is an abstract class with a large number of descendent classes, or if `C` is a common type used throughout the design. Under these circumstances, modifications to `C` could incur mass changes to all its clients and being unencapsulated increases the likelihood of this prospect.

The dynamic priority of a heuristic supplements its static priority. Upon breaching a heuristic `H` on a class `C`, the problem detection phase assigns a static priority to `H`. During problem feedback, this static priority is added to the dynamic priority to get the final heuristic priority for `H`. Dynamic priority

has a number of context-dependent factors that are checked for. A design priority value is assigned according to how many of these factors occur within the offending class. Consider, for example that H is an encapsulation-based heuristic that was breached in class C. If C is abstract with a number of descendent classes and a common type, then a dynamic priority of 2 would be assigned to H. This figure together with the encapsulation-based static priority of 4 would result in a final heuristic priority of 6 for H.

Static Priority	Heuristic
4 [high]	CC6 Hide all implementation details CC5 Limit enabling mechanisms that breach encapsulation CA2 Restrict access to aggregated by clients
3	CU1 Limit the number of collaborating classes CI2 Prevent the over-generalisation of the parent class
2	CA1 The aggregate should limit the number of aggregated CC1 Limit the number of methods per class CC2 Limit the number of attributes per class
1 [low]	CC4 Minimise complex methods CI1 Limit the use of multiple inheritance CU2 Restrict the visibility of interface collaborators

Table 4—5 : Ordering of heuristics

Heuristic priorities serve as a management tool for designers during evaluation. They permit designers to focus upon the problems that are deemed most damaging to a system's maintainability.

4.7 Summary

Design heuristics provide a vocabulary for common problems occurring in OOD documents. Heuristic forms provide a template for documenting these heuristics and an effective means of communicating design problems amongst developers. The heuristic models conform to an interface-specific view of classes. This enables them to be used as early as late analysis through to implementation and beyond. Although the type of design document changes upon moving from OOA to OOD to OOP, the nature of the design

problems do not, thereby allowing heuristics to be applied in a timely, instructive and uniform manner.

The catalogue structures design heuristics along four vectors: class, inheritance, using and aggregation. These heuristics can be used in isolation, or synergistically to describe higher level problems through concept and quality categories. More specifically, these categories document the explicit links between maintainability and paradigmatic concepts, and the design heuristics that support them. The heuristics in the catalogue have been objectified to automate their detection within software designs. Heuristic automation comprehensively analyses OOD documents, annotating the generic feedback documented within heuristic forms with information gleaned directly from the design under evaluation.

The next chapter describes a prototypical tool that implements the automatable design heuristic catalogue. Chapter 6 reports on how design heuristics were received by students and how they can be used to review large OO software. In chapter 7, based upon the results gleaned in chapter 6, a number of conclusions are drawn and areas for future research outlined.

Chapter 5

The TOAD system

5. Overview

This chapter describes how TOAD (Teaching Object Analysis and Design) implements the automatable design heuristic catalogue presented in chapter 4. The objectives of TOAD are set out in section5.1 and their requirements described in section 5.2. Using TOAD, design documents produced at both the method and language level can be evaluated and feedback presented to the user. Section5.3 details how designs produced by different OOD methods and different OO programming languages can be evaluated within TOAD.

Section 5.4 presents the TOAD model upon which the design heuristic catalogue and class role migration are founded. Class role migration is a novel model of maintainability that aims to pinpoint which design classes are most likely to undergo future changes and at what cost to the rest of the system. Finally, in section 5.5 an architectural overview of the TOAD system reports on how the models presented in this chapter inter-relate to form a single coherent design evaluation tool. The material presented throughout this chapter has been published in [GLH96][GH96a][Gi97a].

5.1 TOAD Objectives

The implementation of TOAD strives to meet the following goals:

- to provide a tool that can readily be incorporated into existing design processes;
- to implement the automatable design heuristic catalogue;

- to provide a design diagnostic environment with solution support for both novice and experienced designers alike;

- to evaluate the extent to which TOAD can objectively articulate small to medium sized designs.

This chapter addresses the first two objectives leaving chapter 6 to focus upon the remaining two. In light of these objectives, initial progress made towards TOAD aimed to establish both its system and user requirements. In the next section answers to the following questions were sought: what design problems do developers typically encounter? Are there common solutions to these recurring problems? How should the tool present/enforce its ideas?

5.2 Soliciting TOAD requirements

A series of self-evaluative procedures were carried out prior to the development of TOAD [GLH96]. Their aim was to solicit from TOAD's inaugural audience, students, the way that they perform design and their grasp of object technology. These procedures took the form of mailing groups, design questionnaires and laboratory appraisals. Our primary concern was to gain a clear understanding, from the user's perspective, of what TOAD should deliver and how.

A number of design problems were posted periodically to a mailing group entitled `g52obj`. Its members were students undertaking the OO methods course, OBJ, at the University of Nottingham. The aim of `g52obj` was to determine in what aspects of object technology students were weakest and the way that students articulated OO concepts. An analysis of `g52obj` mail messages highlighted the technological areas that students found difficult and also presented, by examining the vocabulary of `g52obj` mail messages, the level at which TOAD should pitch its design solutions.

Laboratory sessions saw learners doing OOD. The need for immediate feedback on both design problems and their potential solutions was made clear. Questionnaires completed by students after the semester one, 95/96

OBJ course, asked whether they thought a tool could provide any sort of support during design. 92% of the students believed that tool support would be beneficial. Of the detractors, the consensus was a fear of design invasion and/or persuasion. They believed that the unsuspecting learner might always be tempted to adapt their designs, possibly for the worse, to support the tool's view on design correctness.

Taking the results of these evaluation procedures into account a short-list of TOAD's requirements was drafted:

- design feedback, both positive and negative, should be expressed in the learners' vocabulary;
- problem detection, problem feedback and solution guidance should be informative, informal and simple;
- the tool should fully annotate design alternatives giving in-depth concrete examples where possible;
- a flexible approach to evaluation should be adopted [VJT92] such that design enhancements are seen as suggestions that are acted upon at the learner's discretion;
- heuristics must be available during design if their suggestions are to be applied by the learners;
- to reduce the amount of effort necessary to incorporate the tool into existing design processes, it must support a certain level of method/language/platform independence.

As chapter 4 surmised, the last two requirements rely upon the presence of an interface-specific model of classes. TOAD implements this view of a design as the **abstract design model** This model makes no assumptions about the underlying implementation, focusing primarily upon the interfaces that exists between classes. The abstract design model embodies the static,

class-based properties of an OOD in supporting both a method and language independent view of classes and their inter-relationships.

The next section describes how the abstract design model is captured in a platform independent manner using the **class description language** Section 5.4 illustrates how TOAD articulates the abstract design model in a dynamic manner to pinpoint potential maintenance problems.

5.3 Class description language

Figure 5—1 illustrates some of the many different types of design document and how they are converted into the abstract design model prior to evaluation within TOAD. The class description language (CDL) adopts an ASCII flat file format in providing a platform independent way of representing the abstract design model. The CDL parser, upon reading design documents represented as CDL files, initialises the structures within the abstract design model. The CDL parser was specified using the UNIX meta tools `yacc` and `lex` [LMB90]. `yacc` and `lex` were used generate a portable C version of the CDL parser that permits any architecture that supports a C compiler to parse CDL-compliant files.

Converting design documents into the abstract design model enables TOAD to evaluate designs at different phases of the life cycle and those produced by different OOD methods. However, as section 5.3.2 conveys, method and language independence does not come without cost.

5.3.1 Abstract Design Model

The abstract design model views an OOD as a collection of collaborating class clusters. Each cluster is comprised of a set of classes and the relationships between them: inheritance, using and aggregation [Bo94]. The class-based nature of the abstract design model means that only the static properties of OOD documents are modelled. It is the TOAD model described in section 5.4 that is responsible for modelling classes in a behaviour-driven manner. The

contents of the abstract design model are discussed in more detail in Appendix A.

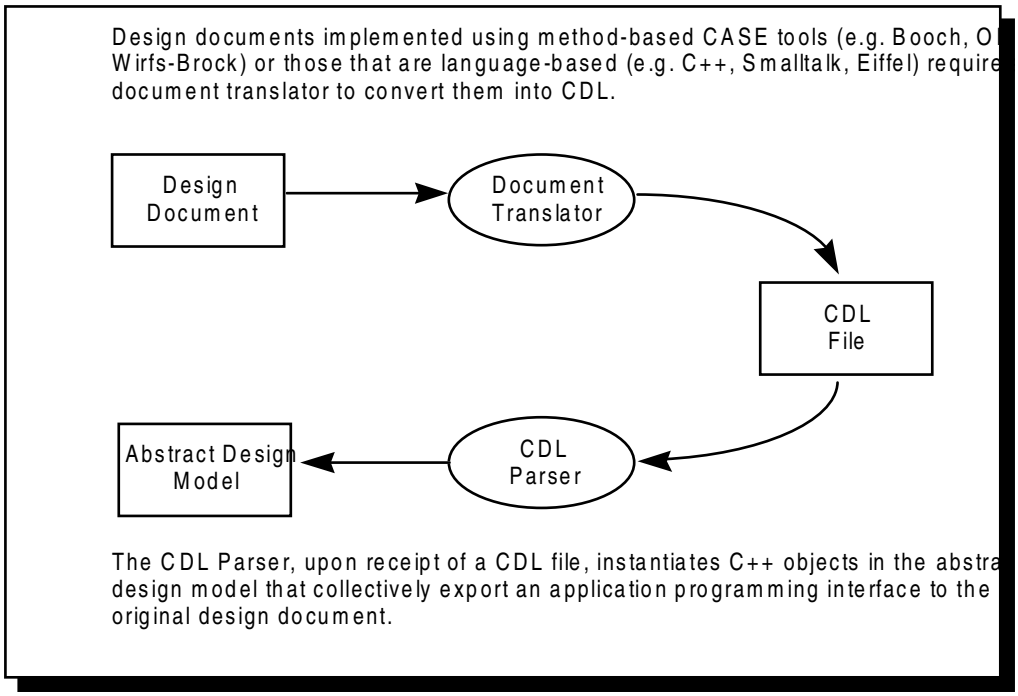


Figure 5—1 : From design document to abstract design model

Figure 5—1 depicts how the abstract design model results from parsing a CDL file. However, it is the document translator that is responsible for generating a CDL file from a design document. In the next section we demonstrate that it is the document translator that affords much of the effort required to convert a design document into the abstract design model.

5.3.2 Document translators

Consider the unilateral printer language PostScript™. Every application that wants printing facilities must provide the means to convert its internal models into PostScript. This way the onus is placed upon the individual application to support this well-defined language interface. The same model, illustrated in Figure 5—1, has been adopted by TOAD. Method and language independence necessitates that every design document provide its own translator to convert it into CDL. The next two sections give a brief

description of some of the problems encountered implementing C++ and CASE-based document translators.

5.3.2.1 Source code document translators

The OBJ course saw students implementing their designs in C++. This necessitated the presence of a C++ document translator before students could use TOAD to evaluate their designs. C++ is an expressive and complex language that warrants the employment of sophisticated tools and intelligent grammars to parse it. A number of approaches were taken, each with their reported advantages and limitations.

gen++

AT&T licensed a C++ tool entitled gen++ for this research to evaluate how effective and/or comprehensive it was at querying C++ applications. gen++ programs are implemented using a lisp-like specification language in which C++ program elements are represented as gnodes in a gtree. gen++ programs iterate over the gnodes in the gtree which are organised according to the grammar rules implemented by the AT&T C++ compiler.

The expressive nature of C++ syntax resulted in complex gen++ programs that embodied lengthy list-traversals to extract the design attributes needed by the CDL. This resulted in:

- increased memory usage: gnodes are created dynamically and with no garbage collection mechanisms or language features to delete them, a proliferation of redundant gnodes ensued;

- decreased performance: as the number of iterations and depth of recursion increased to parse complex C++ code, redundant gnodes and complex list traversals presented uncompromising performance hits.

Before gen++ could deploy its query scripts it was required to build the *complete* gtree. For CDL, where the required design information resides at the interface and not the implementation level, this unnecessary overhead

proved too restrictive for large C++ systems. Although AT&T acknowledged that gen++ was tailored towards smaller C++ programs with limited functionality, their next release was not forthcoming. The proprietorial nature of other C++ parsing tools saw research directed towards customising and implementing grammars to extract the relevant CDL design attributes from C++ program source.

C++ grammars: The lightweight approach

A lightweight approach to extracting design details from C++ class definitions was first adopted. A yacc-specified grammar was implemented that focused on the interface-specific attributes of a design and ignored all aspects of the implementation. However, C++ permits programmers to embed implementation details within the class definition. Therefore, the grammar needed to differentiate between interface and implementation, otherwise problems resulting from parenthesis mismatch and token ambiguities would ensue. This extraneous implementation knowledge was captured as additional grammar rules that saw the lightweight approach adopting heavyweight properties.

Techniques to suppress unnecessary tokens being delivered to the parser were implemented using multiple states within the lexical analyser. The lexical analyser turned off token streaming to the parser when it encountered implementation details in the class definition and recommenced streaming upon departure. The lexical analyser was now responsible for keeping track of numerous tokens pertaining to parenthesis, curly braces, string and character literals constants, and so on. In retrospect, the lexical analyser was temporarily assuming the role of the parser for which it was inadequately equipped.

The lightweight approach highlighted that *incomplete* grammars can not be used to parse *complete* C++ programs. Instead, research took to customising heavyweight grammars to perform lightweight functions.

C++ grammars: The heavyweight approach

A yacc-specified C++ grammar written by Roskind [Ro88] did not incorporate many of the new language features (e.g. templates) but provided a basis for implementing a C++ design document translator. The 90% yacc-specified C++ grammar released as part of the Gnu g++ compiler supplied the missing grammar rules for these new C++ language features. However, in both cases the grammars were incomplete and attempts to augment them yielded grammars that were easily broken.

A C++ grammar implemented at NeXT using the Purdue Compiler Construction Tools Sets [Pu97] (an alternative to`yacc/lex`) was adapted and ported to UNIX by Lilley [Li97]. Upon modifying this grammar and implementing a number of supporting tools, Figure 5—2 illustrates how TOAD converts C++ design documents into CDL-compliant files.

In summary, grammar-based design document translators for non-standard programming languages such as C++ need to account for language nuances at the compiler level which could result in a plethora of (un)supported document translators. However, we contend that such design document translators need only be used for the post-analysis of existing software by TOAD. The true benefits of TOAD come from the ability to adapt software designs prior to implementation. To this end, document translators at the design level should populate development environments so that the full benefits of TOAD can be realised. Typically this occurs within a CASE-based environment.

5.3.2.2 CASE-based document translators

As alluded to in the previous section, a large amount of OOD is performed within a CASE environment. However, CASE tool design models are typically closed to software developers wishing to gain access to design attributes. Industrial demands by software practitioners [Me96] saw CASE tool vendors exporting application programming interfaces(APIs) to their internal design models. The content of these CASE tool APIs was addressed

at a OOPSLA'95 workshop [NB95] in which participants short-listed the requirements of software that needed to manipulate OOD documents within a CASE environment. Typically, these tools were either evaluating, extending or forward/reversing engineering OOD models within CASE.

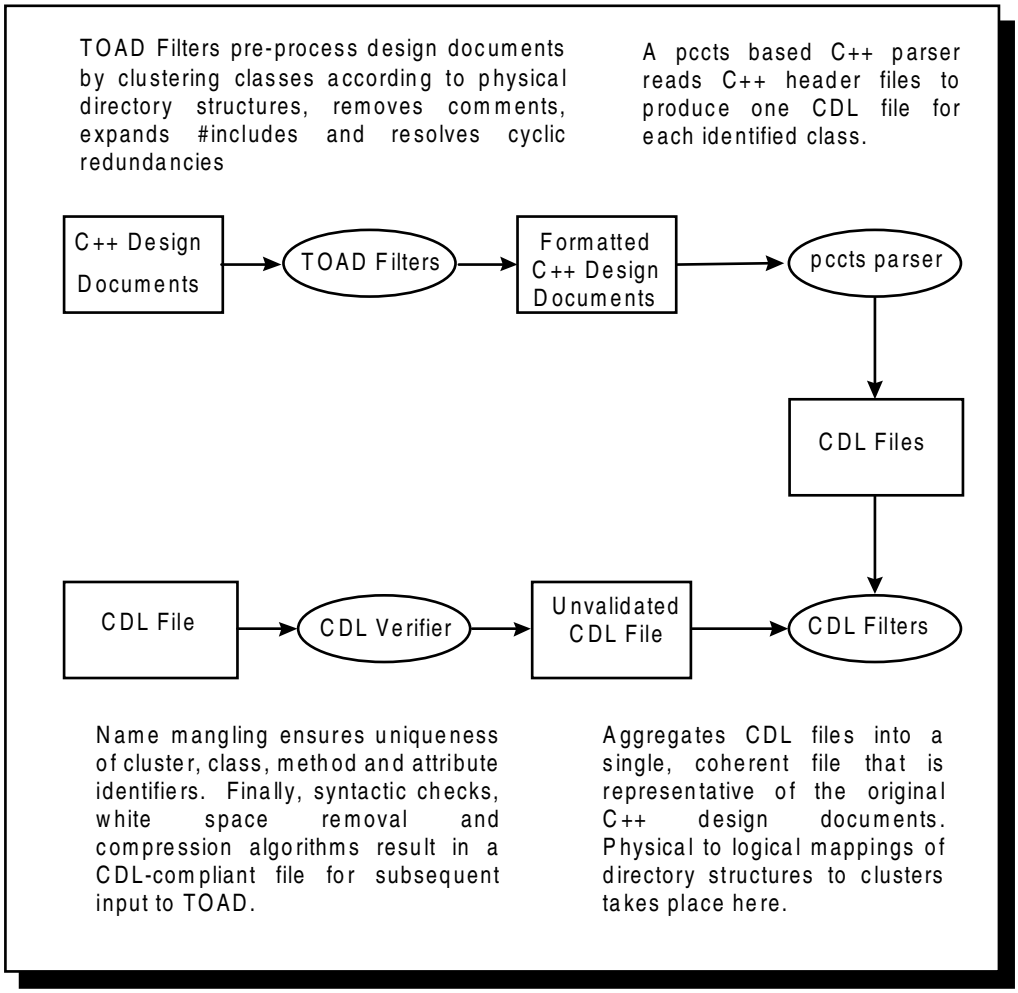


Figure 5—2 : pccts and supporting tools

A beta version of a CASE tool that exported an API to its internal design model was licensed for use by TOAD. Using this API, TOAD extracted the required design attributes from the CASE tool's internal design model needed for inclusion within CDL. The API was implemented using a variant on the Visual Basic scripting language. The design document translator for this CASE tool ran under Windows95 and vastly reduced the effort required to convert CASE design models into CDL. Chapter 6 describes how student

designs were captured using this CASE tool. Once in the CASE tool, document translators transformed the student designs into CDL-compliant files for subsequent evaluation within TOAD.

To date, document translators for Java, C++ and the aforementioned CASE tool which supports the Booch, OMT and UML OOD methods have been implemented as part of TOAD. Document translators output CDL files that are parsed to initialise the abstract design model which in turn provides the foundation for the TOAD model.

5.4 The TOAD model

A thorough and formal definition of the TOAD model is given in [Gi97a]. This section presents a concise description of the model's rationale and the concepts employed to deliver its solutions.

TOAD models the static properties of an OOD in a dynamic manner. It builds directly upon the abstract design model presented in section5.3.1 by accentuating the inherent client/server relationship that exists between classes. A behaviour-driven model of an OOD views classes in terms of the services they provide and/or depend on. Based upon this model a class in TOAD can assume one of four roles, that of a **server**, **actor**, **agent** or **standalone**. Class role migration (section 5.4.2) and the TOAD system (section 5.5) articulate the well-defined behaviours exhibited by these roles to enhance the maintainability of OOD documents.

5.4.1 Assigning class roles

Class roles are assigned within the context of a class relationship: inheritance, aggregation or using. Consider, for example, the `Earthling` inheritance hierarchy in Figure 5—3. `Minotaur` is considered an *actor* because it uses the services of its parent classes `Man` and `Bull` but does not distribute any behaviour of its own. Conversely, `Earthling` serves only to distribute commonality amongst its child classes and is therefore labelled as a*server*. The *agent* `Bird` uses the services of `Earthling` and distributes behaviour to

child classes `Vulture` and `WanderingAlbatross`. Finally, the *standalone* `Ewok` does not participate in the inheritance relationship within system X. In general, root nodes in a class hierarchy are servers, intermediary nodes are agents and leaf nodes are actors.

At the cluster level only the using relationship exists between groups of cohesive classes. A cluster P is said to use the services of cluster Q if it depends upon any of the classes defined within Q. In this instance, P is the actor and Q the server. A cluster R is an agent if, for example, it uses the services of P and provides services to Q.

When assigning roles, the component (class or cluster) under scrutiny is known as the target, C_i, and the components with which it collaborates directly are placed within its response set(s). Servers to C_i are placed within its server response set, or SRS_i, and actors within its actor response set, ARS_i. Returning to Figure 5—3, the roles within system `X` are thus:

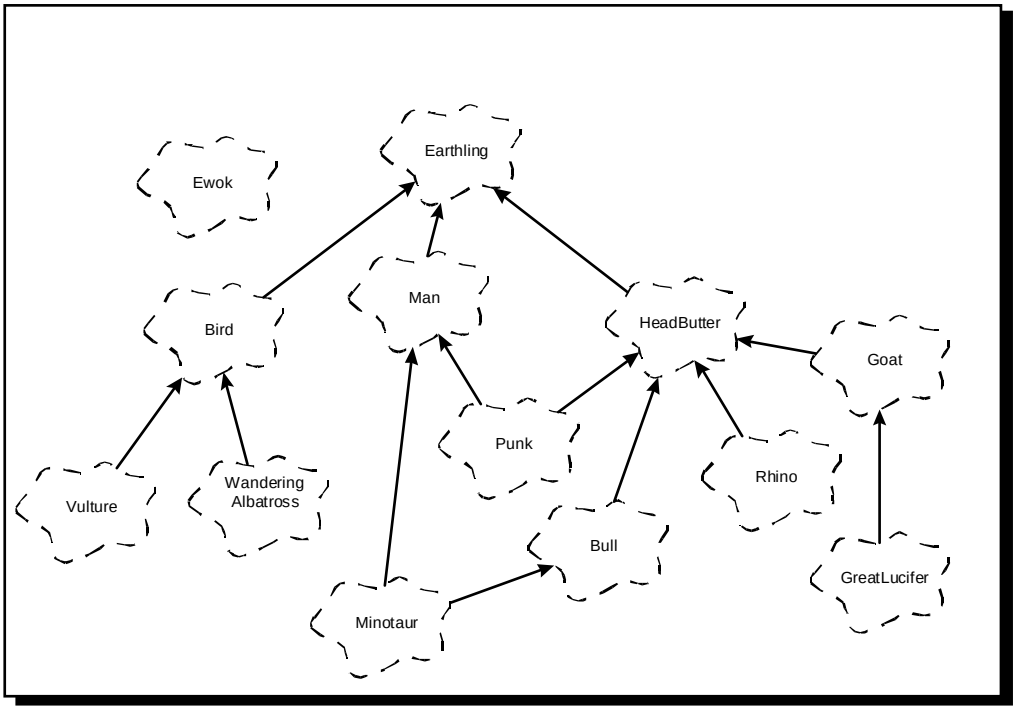


Figure 5—3 : The Earthling inheritance hierarchy for system X

```
servers     = { Earthling }
actors      = { Vulture, WanderingAlbatross, Minotaur,
                Punk, Rhino, GreatLucifer }
agents      = { Bird, Man, HeadButter, Bull, Goat, }
standalones = { Ewok }
```

and examples of SRS_i and ARS_i for the target classes `Goat`, `Earthling` and `Minotaur` are thus:

```
Goat       ⇨   SRSi     = { HeadButter }
               ARSi     = { GreatLucifer }

Earthling  ⇨   SRSi     = {}
               ARSi     = { Bird, Man, HeadButter }

Minotaur   ⇨   SRSi     = { Man, Bull }
               ARSi     = {}
```

Figure 5—4 depicts the target class `HeadButter` that is an agent within the `Earthling` inheritance hierarchy. Applying the TOAD model, a server does not have an *SRS*, an actor does not have an *ARS*, where an agent has both. Furthermore, because roles are assigned on a per class relationship basis within TOAD, a class could have different roles for every class relationship that it participates in. Collectively, the target class and its response sets are known as a TOAD class.

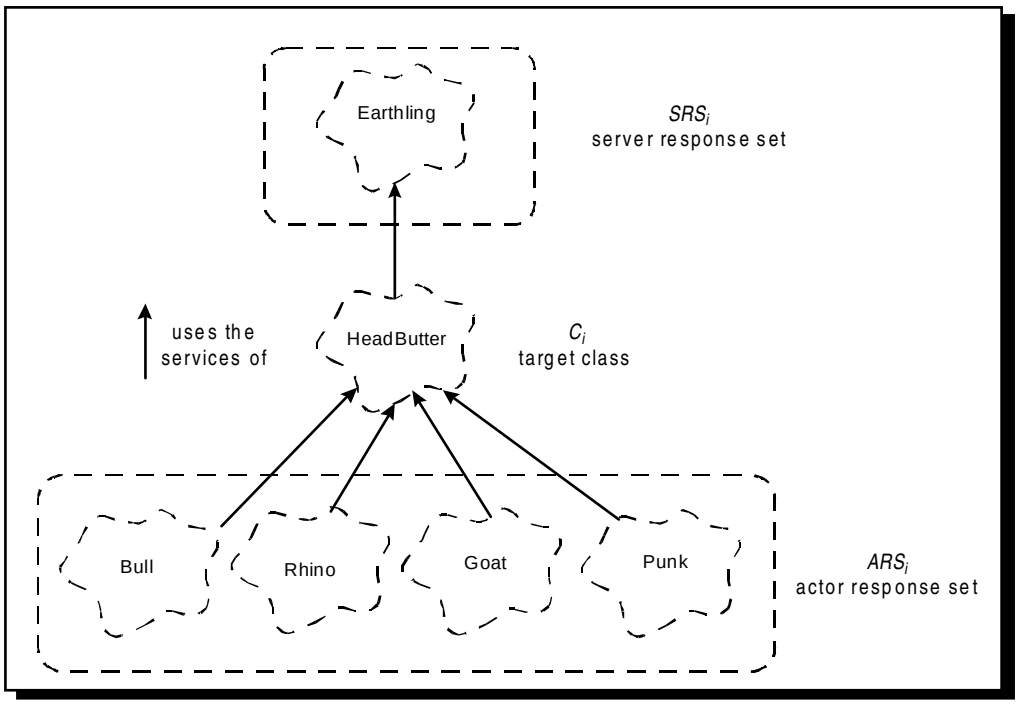


Figure 5—4 : HeadButter class within the TOAD model

5.4.2 Class role migration

By extrapolating the properties of servers, actors and agents, role migration highlights those classes most likely to undergo future modifications. Applying class role migration at the design phase not only highlights those classes destined to change but also those classes that initiate and/or are affected by those changes. The basic premise underlying class role migration is that designing for change at design time is less costly than implementing that change during the coding/maintenance phases.

Consider again the inheritance hierarchy in Figure 5—3. Extending this hierarchy through its leaf nodes represents the actor/agent role migration for the inheritance relationship. For example, the subclassing of `Punk` with `NeoPunk` sees the actor `Punk` adopting the agent role. Gibbon [Gi97a] comprehensively documents the causes, effects and consequences of the most important role migrations for each class relationship. TOAD applies these role migrations to every class in a design to determine the likelihood and cost of future system changes.

Preliminary results from applying class role migration as means for pinpointing change prone classes are presented in appendix E. The promising results received to date have suggested new directions for future research into this model of design maintainability within OO software.

5.5 The TOAD system

5.5.1 Platform selection

The choice of platform to implement TOAD on was influenced by the availability of tools for specifying language grammars (CDL), development libraries for building graphical user interfaces and the author's familiarity with the environment. However, the overriding factor was the platform that the inaugural users of TOAD were most familiar with. As the UNIX platform proved to be the only common denominator amongst students taking the OBJ

course at Nottingham, it was deemed appropriate to implement TOAD within the same operational environment.

TOAD was written using Gnu C++ v2.7.2 and has been ported to Linux, SunOS, Solaris and Silicon Graphics Unices. Its graphical user interface was implemented using OSF Motif v1.2 and release 5 of the X windows system. The UNIX meta tools `yacc` and `lex` that are released at part of Solaris v5.5 were used to specify the grammar for the class description language.

5.5.2 Architectural overview

Figure 5—5 gives a simplified overview of the TOAD system and its constituent components. Each of these are described in turn:

- the `Abstract Design Model` exports both a method and language independent static class model as a framework of C++ classes;
- the `TOAD Model` builds upon the abstract design model by assigning each TOAD class a role for every class relationship it participates in;
- a series of `TOAD Analysers` perform queries over the TOAD model on behalf of components higher up in the system architecture;
- the automatable `Heuristic Catalogue` implements a collection of cohesive quality classes in which related design heuristics are placed;
- `Class`, `Cluster` and `Relationship Reports` utilise these quality classes to deliver diagnostic feedback, generic solutions, and context-driven help to the designer;
- the `TOAD System` exports a well-defined C-Style API to the underlying report/heuristic structures;
- the `GUI Components` are C++ wrappers for Motif/X widgets whose registered callbacks invoke functions specified within the TOAD API;

- the `TOAD User Interface` co-ordinates its subordinate models and presents a graphical view of OOD documents and the results of their evaluation.

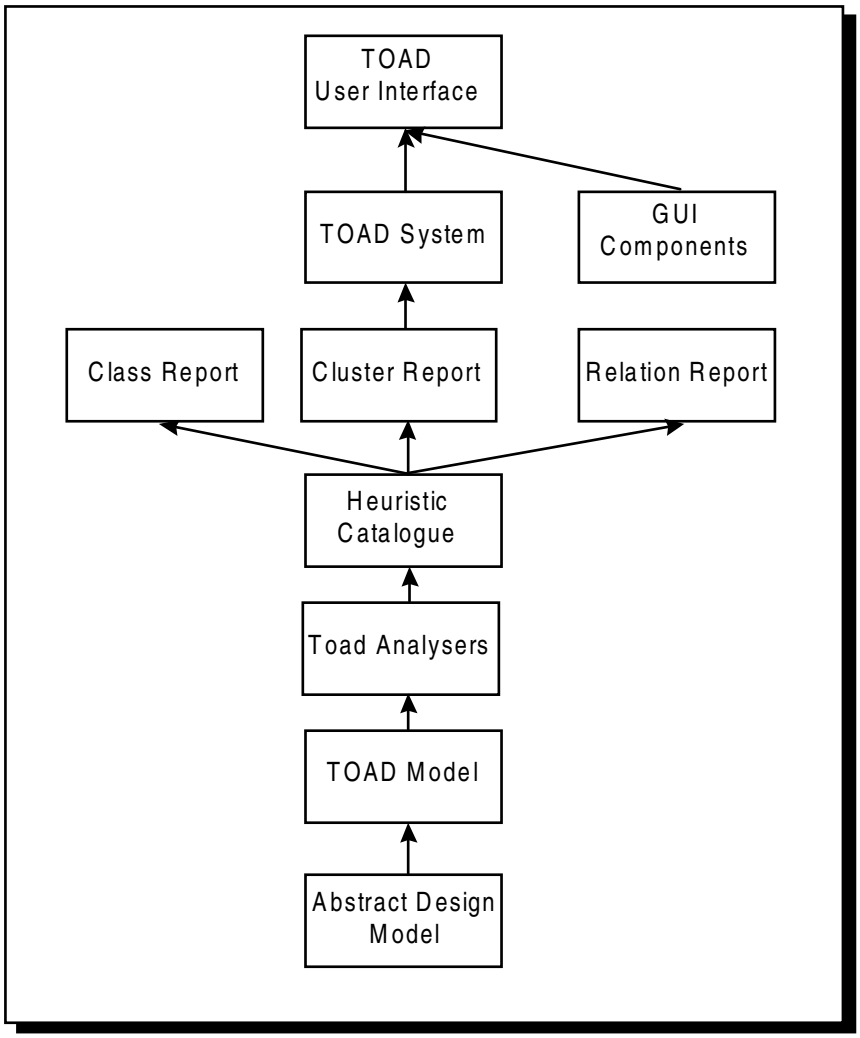


Figure 5—5 : Architectural overview of the TOAD system

5.5.3 The working TOAD

The `Heuristic Catalogue` supports a family of quality classes that each house a collection of related design heuristics. Figure 5—6 illustrates the hierarchy of quality classes that map directly onto the structure of the design heuristics within the catalogue (see section4.4).

In TOAD, users interrogate their designs through reports. A class report: issues statistical feedback on the class; highlights which heuristics have been

breached and retrieves their associated rationale; annotates design concepts such as encapsulation, inheritance, collaborations etc. with respect to the current class; displays design hints in the form of concrete examples and/or software patterns. This same feedback process holds for the cluster and relationship reports. Quality classes hold all the information on design heuristics which in turn utilise TOAD analysers to interrogate the underlying TOAD model. The process by which users evaluate their design documents with TOAD is detailed in appendix B.

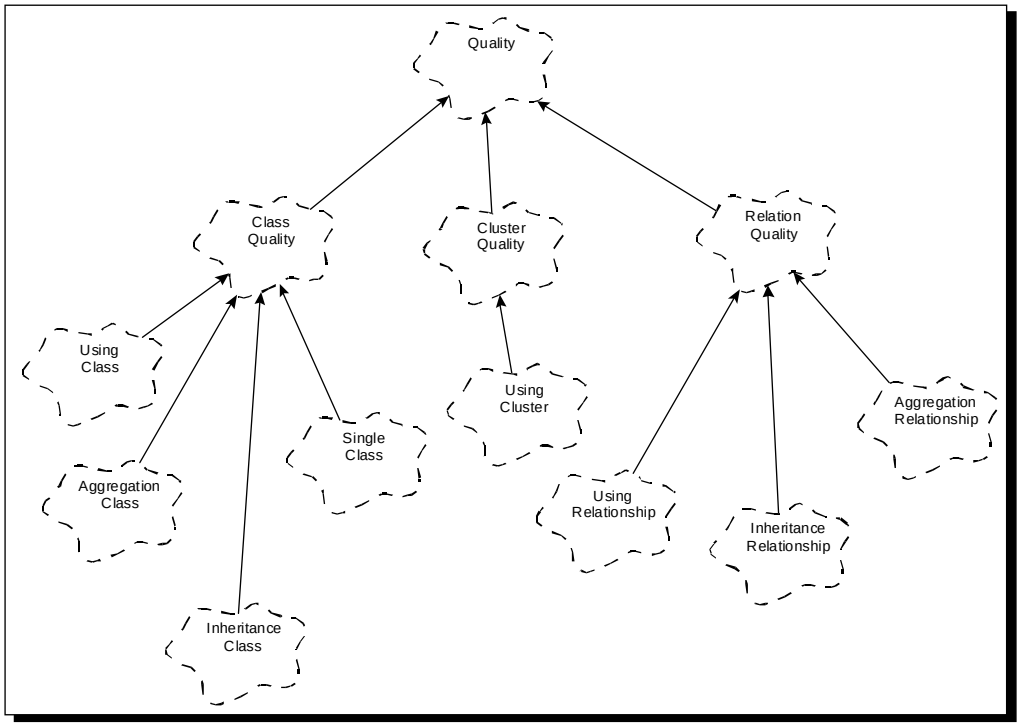


Figure 5—6 : Mapping quality onto heuristics

5.6 Summary

This chapter presented a prototypical tool for evaluating OOD documents and outlined its implementation. TOAD permits software designs produced by different OOD methods and/or different OO programming languages to be evaluated. To achieve this, a class description language is used to capture the essential class-based properties of an OOD.

The class description language is an ASCII flat file that document translators produce upon parsing class models. This affords the required level of method and language independence to permit TOAD to be used within a variety of teaching methods or design processes.

The next chapter reports on how TOAD was received by students performing OOD for the first time. It also investigates the advantages and limitations of TOAD as a design reviewer. Chapter 7 draws some important conclusions from this research and presents ongoing and future development directions for TOAD.

Chapter 6

Results

6. Overview

This chapter presents the results of applying TOAD to OO software. First and foremost, TOAD is an automated, interactive design educator. In this role we report which design heuristics proved most useful to learners, recurring problems in their designs, what solutions TOAD suggested, whether this advice was applied and how useful it was to the learner.

Second, TOAD's ability to evaluate medium to large OO software is also reported. Here, the user is considered an experienced designer who employs TOAD to pinpoint structural design weaknesses with the clarity and precision of an *objective* OOD expert. In both cases, we aim to meet the goals presented in section 6.1 and determine whether heuristics are effective vehicles for software design evaluation.

6.1 Objectives

Section 5.1 outlined TOAD's objectives and fulfilled two of them. This chapter addresses the remaining two:

- to provide a design diagnostic environment with solution support for both novice and experienced designers alike;
- to evaluate the extent to which TOAD can objectively articulate software designs.

Addressing these objectives, section 6.2 reports on the novice designer implementing small to medium sized designs and section 6.3 appraises TOAD in its role as a reviewer for medium to large sized designs.

6.2 TOAD: a design educator

> *"Most software tools are moronic assistants that know what to do but do not understand the purpose of the objects they manipulate or how their tasks fit into the development process. In other words, they know the how but do not understand the why!" Vessey et al [VJT92]*

For TOAD to be considered as an intelligent design educator, where the whys take precedence over the whats, wheres and hows, it needs to embrace the concepts and principles of the underlying object model. As reported by Gibbon and Higgins [GH96b], design heuristics implemented within TOAD fulfil both these essential educational and paradigmatic requirements for teaching OOD by example. Section6.2.1 documents the experimental details. It describes how design data was collected and the way that students perform OOD in the presence of design heuristics. An analysis of the designs evaluated both using TOAD as an intelligent automated design educator and without are discussed in sections 6.2.2, 6.2.3 and 6.2.4. A more in-depth breakdown of the material presented in this section can be found in [GLH96][GH97b].

6.2.1 Experimental details

6.2.1.1 Test subjects

The evaluation of design heuristics as educational aids spanned two successive academic years at the University of Nottingham. The test subjects were second year computer science undergraduate students taking the Object-Oriented Methods course entitled OBJ. The 95/96 OBJ student intake were presented with design heuristics in lectures and told to use them throughout the design process. TOAD was only made available to the 96/97 OBJ students.

A questionnaire was completed by the OBJ students before performing their design task. The aim of the survey was to ascertain the students'

programming and design background and what their views on software quality were. Of the 51 students on the 95/96 OBJ course, 44 of them replied, whereas 47 from a possible 62 completed the questionnaire on the 96/97 OBJ course.

Figure 6—1 illustrates that the majority of students taking the OBJ course have little or no design experience. More to the point, except for a first year C++ programming course that taught students how to program with abstract data types, they had encountered virtually no elements of object technology.

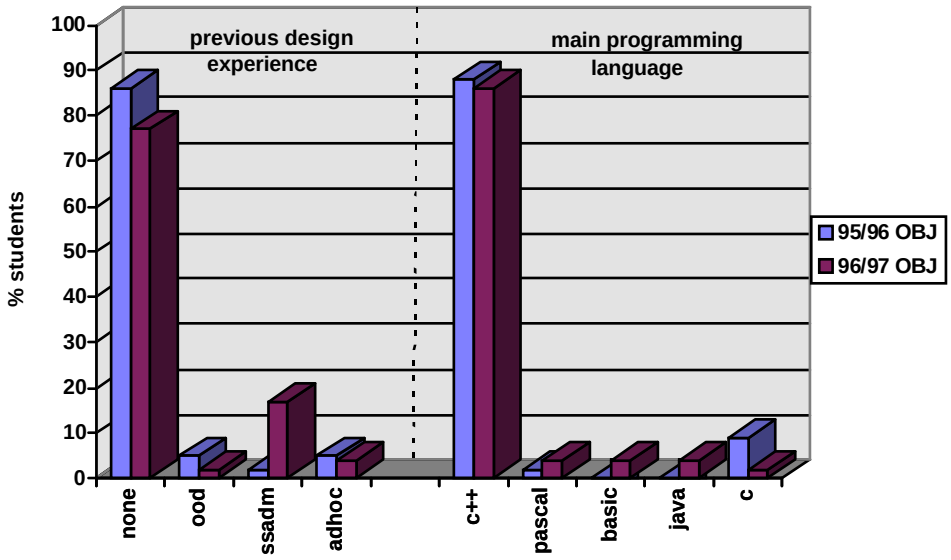


Figure 6—1 : Comparing the 95/96 and 96/97 test subjects

Not only were the test subjects devoid of design experience but asFigure 6—2 clearly depicts, they also had limited experience in building applications with more than five classes.

When quizzed on issues pertaining to software quality, the questionnaire revealed their lack of awareness of, for example, how design affects maintainability (see Figure 6—3). Although expected, these important gaps in their understanding was a major influence for building such knowledge into the design heuristic catalogue.

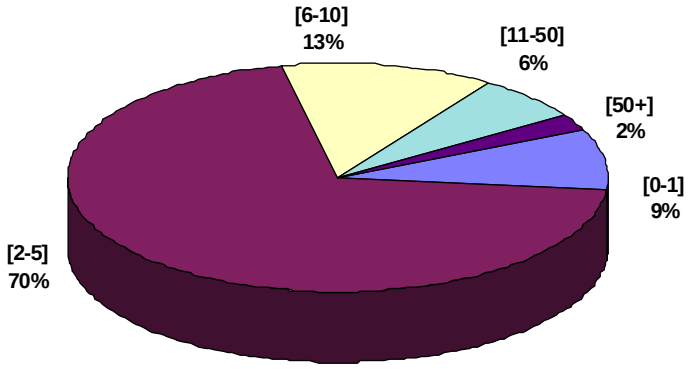


Figure 6—2 : How many classes did your largest application contain?

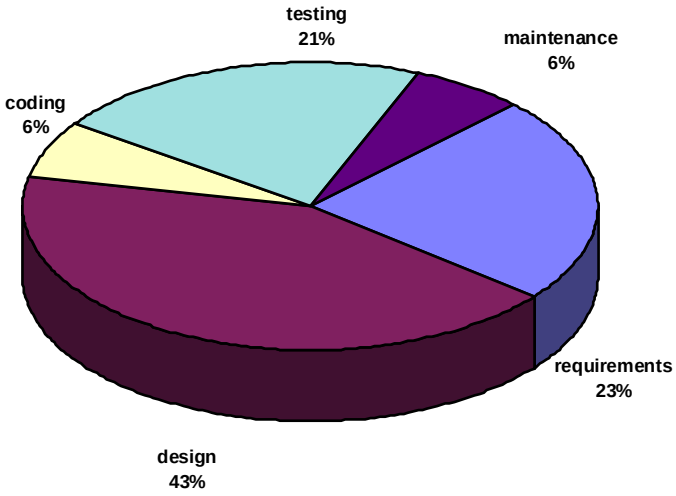


Figure 6—3 : In what phase does software spend most of its life?

6.2.1.2 Test data

The same eight problem statements, summaries given in Table 6—1, were distributed evenly amongst the OBJ students for both academic years. The nature of the supplied problem statements gave rise to design solutions that comprised of approximately 20-25 classes. The deliverables for the 95/96 and 96/97 OBJ course, listed in Table 6—2, highlight which class models and at what stage they were captured using a commercial OOD CASE tool. Note

that the 96/97 OBJ students performed an extra design maintenance task that served to extend their first OOD class model. The deployment of heuristics by TOAD was made available to the 96/97 OBJ students *once* they had submitted their first OOD.

Car Fleet; a company car pool system that manages the assignment of vehicles to members of an organisations, monitors their usage and commits them to repair at the more cost-effective garage.
Atom Video; a video rental system responsible for the day to day management of videos, members and borrowers.
University Library System; a library system responsible for maintaining the books and their borrowers that are all members of the university, for example staff, students, research assistants and so on.
Abbey Hotel; a hotel management system that keeps track of rooms usage, guests, handles check-in/check-out and customer billing.
Payroll; a system to automate the payment of employees within an organisation together with the addition/removal of them, union deductions and support for different methods of payment.
The Baring-Up Bank; a system to carry out the day to day customer transactions within a bank which includes deposits, withdrawals, statement printing, open/close account, compound interest and report facilities for rogue accesses/overdrawn accounts.
University Registration System; models the structure of a university for registering departments, courses and students. In addition, the system reports on all examination details within a department.
The Arches; stock taking system for parts within a garage system. The system must keep track of all suppliers, their parts and prices in maintaining an up to date stock level and automate the re-supplying procedure when parts are used during repairs.

Table 6—1 : Design problem statements for OBJ course

In addition to the deliverables outlined in Table 6—2, post-design questionnaires contributed important information to the appraisal of TOAD. By using the same eight problem statements, compare and contrast evaluation procedures were performed on the 95/96 and 96/97 design deliverables.

The 95/96 OOD deliverables will henceforth be referred to as `9500D`, their 96/97 equivalents as `9600D` and the 96/97 OOD documents submitted after their maintenance task as `96FIN`.

	Requirements Phase	OOA	OOD	OOD after maintenance tasks [96/97 only]
Coursework deliverables	system charter use cases	class diagrams(s) scenarios analysis reports	class diagram(s) object diagrams design reports	class diagram(s) object diagrams design reports

Table 6—2 : 95/96 and 96/97 deliverables for the OBJ course

6.2.1.3 Definition of success

Chapter 3 argued that students are not receiving timely and instructive feedback on design documents produced during the software development process. One of TOAD's major *and* ongoing aims is to answer students' continual requests for ways to educate them in exactly what constitutes good design practice and whether they are achieving it in their software designs. Taking those student requests into consideration and the thesis objectives set out in chapter 3, success is defined in terms of the following core questions:

- What are the common design mistakes made by students?
- Are they soluble?
- Can we automate their detection?
- Can we offer assistance during the design process to improve student designs?
- Can all this be incorporated into the teaching method with minimal effort for both educators and learners alike?

The next section presents the results of applying design heuristics and TOAD within OO curricula. Section 6.2.4 subsequently addresses these core questions.

6.2.2 Design heuristics as educational aids

The problems encountered by students undertaking OOD for the first time fall into two distinct categories: process-oriented and product-oriented. Gibbon and Higgins [GH97b] concur with the documented difficulties regarding the process of teaching OOD [Mc93][Ds92][GM96]: finding the initial design classes; performing OOD in an iterative and recursive manner; and assigning/distributing responsibilities amongst classes in a decentralised manner. Although these process-oriented design difficulties form a large part of the teaching problem, this section focuses directly upon the common design errors and anomalies learners have constructing and enhancing OO class models.

6.2.2.1 Common design problems

The majority of the heuristics breached in the students' designs were class based owing to the small to medium size of their solutions. Figure 6—4 illustrates the most commonly breached heuristics by the OBJ students. The heuristic bar in the graph represents a count on the number of *designs* that had at least one class that breached that heuristic. It bears no indication of the number of times that heuristic was breached within a particular design (see Table 4—2 for the heuristic descriptions).

Figure 6—4 clearly depicts that the majority of student designs were comprised of class(es) that had large numbers of attributes (`CA1`, `CC2`) and methods (`CC3`, `CC4`). Furthermore, is was common for student designs to contain unencapsulated classes (`CC6`, `CA2`). The thresholds for the class heuristics illustrated in Figure 6—4 are listed in Table 6—3. As described in section 4.6.1, thresholds represent upper bounds above which the heuristic is considered breached. For example, from the heuristic descriptions in Table 4—2 and the thresholds given in Table 6—3, to breach `CU1`, a class must have more than 4 interface collaborators. These thresholds were set based upon the design solutions we implemented to the said problems, thresholds set in metric literature and our experience developing OO software systems.

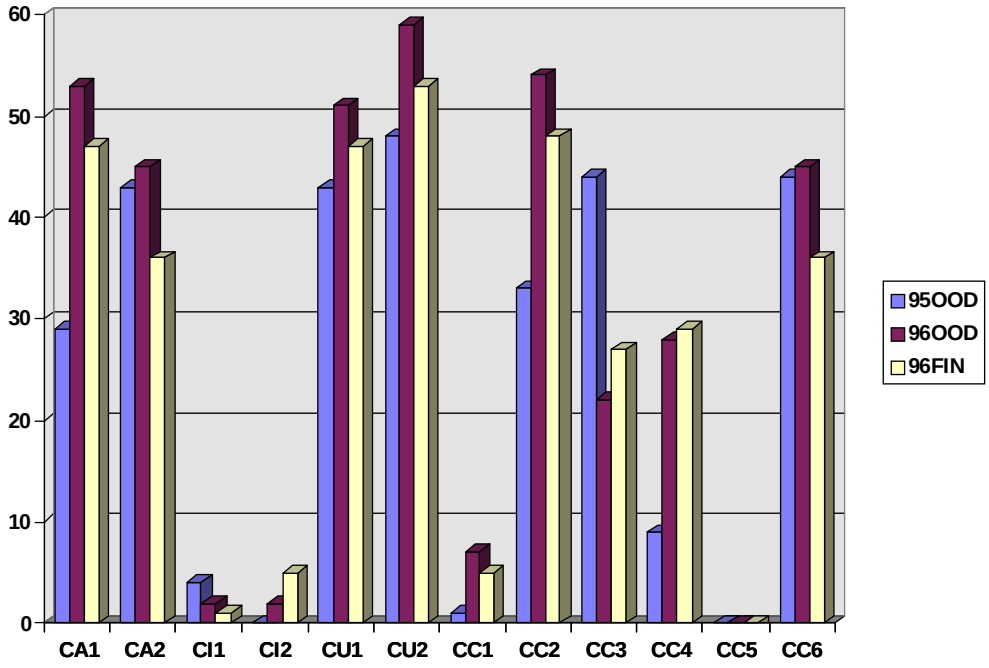


Figure 6—4 : Commonly breached heuristics within the student designs

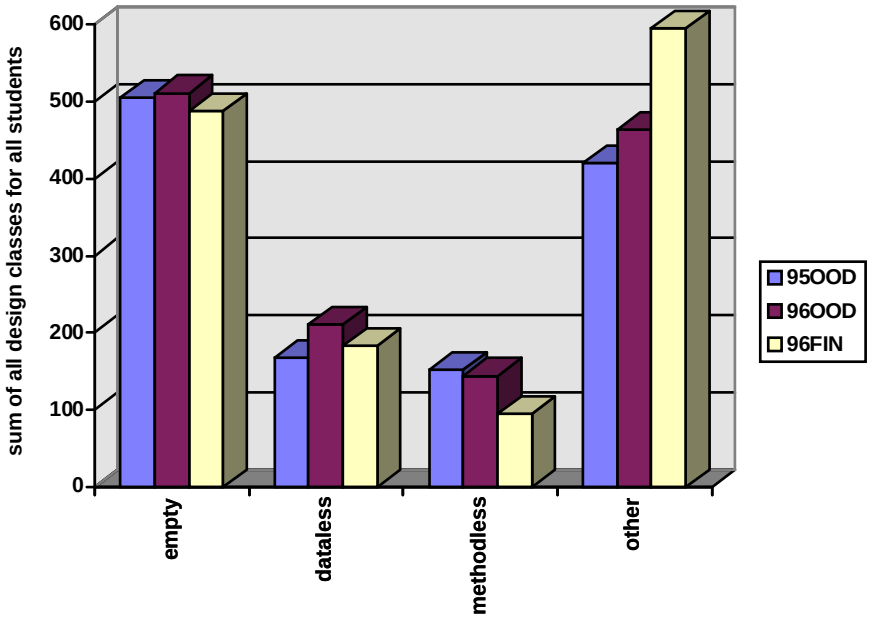
The two most common design problems students encountered were with classes that were either behaviour-rich (**God** classes [Ri96]) or behaviour-poor (**inert** classes [GH97b]). God classes collaborate with numerous system classes to fulfil their responsibilities often resulting in a centralised design architecture. The high count on the `CU1` and `CU2` heuristics in Figure 6—4 showed that the majority of the designs contained at least one class that was behaviour-rich.

CA1	CA2	CI1	CI2	CU1	CU2	CC1	CC2	CC3	CC4	CC5	CC6
4	0	3	4	4	90% public	15	5	10	0	0	0

Table 6—3 : Table of thresholds for class-based heuristics

Conversely, inert classes resemble data sinks that are devoid of behaviour, exporting simple get and set interfaces for use by third party classes. Inert classes stemmed from students building designs in a procedural manner. The resultant inert classes tended to be rich in data and poor in behaviour.

Figure 6—5 illustrates the strong presence of *empty*, *dataless* and *methodless* classes within student designs. Empty classes have neither instance variables nor methods. Typically, they represent classes to be implemented in the future or those provided for by the target programming language. Conversely, classes depicted as *other* represent those that possess both instance variables and methods. However, it is the presence of *methodless* and *dataless* classes that suggests problems in a design. Methodless classes highlight the possible presence of inert classes that are being used as data sinks, where dataless classes tend to be acting as controllers for co-ordinating the behaviour of numerous (disparate) objects. Problems surrounding the use of dataless and methodless classes are



discussed in section 6.2.2.2.

Figure 6—5 : Empty, dataless, methodless and other classes

A well-documented yet recurring design problem faced by all newcomers to object technology is deciding when *and* when not to apply inheritance. Typically, students favoured implementation inheritance over interface inheritance which resulted in classes that made minimal use of

polymorphism and the subsequent creation of inert classes. The visibility of instance variables throughout the class hierarchy, the use of accessor methods to instance variables and the different types of inheritance (*private*, *protected* and *public*) were familiar problems faced by the students.

In general, maintenance tasks performed on a `96OOD` design increased the number of classes in the resultant `96FIN` design by 3 to 4 classes. Despite this, Figure 6—4 illustrates that `96OOD` designs were more likely to breach design heuristics than the `96FIN` designs. This was a clear indication that the students were applying the heuristics *and* the advice suggested by TOAD. For example, Figure 6—5 illustrates how the `96FIN` classes in possessing fewer dataless, methodless and empty classes than `96OOD` classes, acknowledged and applied techniques for keeping data with the methods that act upon them. Although these figures indicate that the students were actively applying the heuristics and the information suggested by TOAD, *did they find it useful?*

6.2.2.2 Were design heuristics useful?

Although heuristics proved useful *inside the classroom* for describing and solving common OOD problems, only restricted use of them *outside the classroom* was made by a small number of students on the 95/96 OBJ course. Gibbon and Higgins [GH97b] concluded that the manual application of a diverse set of highly inter-related design heuristics to frequently changing class models requires more effort than the learners care to expend. As a result, the majority of 95/96 OBJ students did not even attempt to apply the design heuristics, and those that did, only applied them with minimal effort. Nevertheless, the 95/96 OBJ designs, as reported in the previous section, presented invaluable insights into the common design problems faced and committed by students.

In stark contrast to the previous year, the 96/97 OBJ students readily applied heuristics to their design documents when provided with a tool to automate their deployment. TOAD minimised the effort required by students to locate

design anomalies thus giving them more time to focus on improving the quality of their class models. As with the 95/96 OBJ students, prior to the application of TOAD, the majority of the 96/97 students did not attempt to use the design heuristics.

Questionnaires were completed by students to ascertain which heuristics *they* thought they frequently breached and those that *they* found the most useful. In general, the most frequently breached heuristic was also considered to be the most useful.

By far the most commonly breached heuristics were `CU1` and `CU2` that determined a class's connection with its architecture. The student found them useful for highlighting candidate God classes and presenting feedback on how to fragment them. `CC1` and `CC2` were the second most commonly breached heuristics that proved to be as equally useful as `CU1` and `CU2`. Breaching both `CC1` and `CC2` illustrated those classes that were large and potentially God classes. Firm evidence of a centralised architecture and the contributing God class(es) resulted from violating all four of these design heuristics within a single class.

The students admitted to having a great tendency to construct large classes, in particular large classes with numerous attributes. Previous experience from using classes as convenient vehicles for constructing large data structures was often cited as the reason for this. Nevertheless, students found that the `CU1`, `CU2`, `CC1` and `CC2` heuristics highlighted the limitations of this approach to class design. These heuristics outlined why decentralised architectures were preferable and directed the students towards this objective.

`CC6` was also frequently breached and ideally suited for espousing the benefits of well-encapsulated objects. Students submitted that although in lectures they were fully cognisant of the concept of encapsulation, the advantages attributed to its proper enforcement were not forthcoming until modelled directly within they own designs. It wasn't until experiencing how

many unrelated objects in their design could (inadvertently) affect the state of their unencapsulated objects, that the objective of encapsulation became clear. Prior to this, the advantages of encapsulation remained*deep within their lecture notes* and not a powerful, accessible design mechanism for use during software development.

Of those heuristics that were not frequently breached, *minimise complex methods* (CC4) and *limit the number of user-defined types within a class* (CA1) proved useful for distributing design intelligence. Typically, complex methods were found in large classes and used to assign state to all instance variables with a single message send. As expected, the re-design of large classes resulted in a reduction of instance variables thereby making these complex methods redundant.

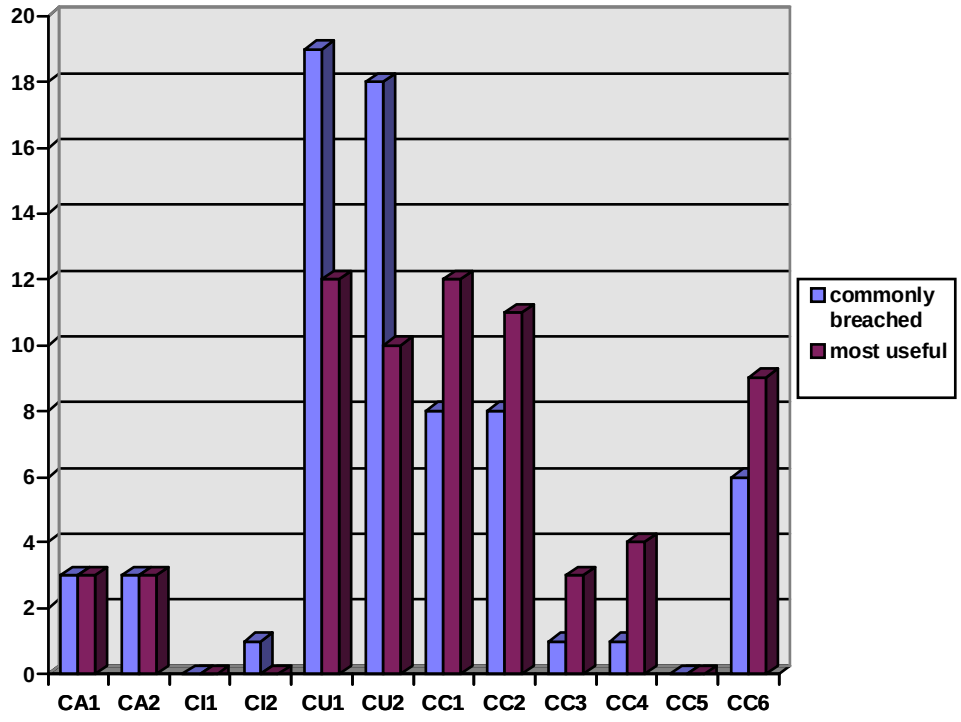


Figure 6—6 : 96/97 OBJ students useful and commonly heuristics

Students singled out the identification of dataless and/or methodless classes within TOAD as an important part of their design feedback. The*keep data*

with methods message that the design reports reiterated upon discovering dataless and/or methodless classes saw a change in attitudes amongst students regarding class design. The students did recognise, albeit towards the *end* of design process, the reason why classes should contain data that is frequently operated upon by the methods that it exports. This was yet another message delivered in lectures that was not assimilateduntil problems resulting from its misapplication or ignorance of it were committed by students in the context of their*own* designs.

6.2.3 Was TOAD useful?

A detailed questionnaire was completed by the 96/97 OBJ course students soliciting their opinions on TOAD as an automated design educator. Figure 6—7 illustrates the apportioning of the 56 replies.

It was apparent from the students' feedback that TOAD should place more emphasis on disseminating its solutions than on explaining the identified design problems. Students favoured a more concise breakdown of design difficulties and the presentation of a more diverse set of design alternatives. Furthermore, the solutions should be well-documented and expressed in terms of the design classes under evaluation.

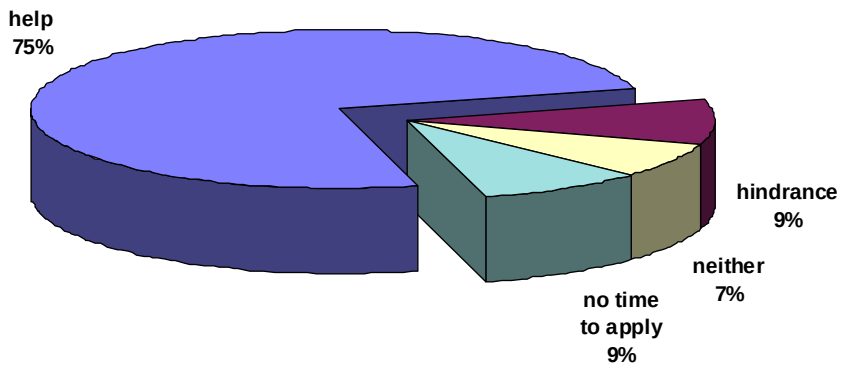


Figure 6—7 : Was TOAD useful during OOD?

However, although enhancing the solution aspect of TOAD is an important objective, we believe that the students' dismissive attitude towards understanding the problem was in favour of a quick fix to the design. A similar situation arises within programming environments in which the *edit-compile-debug* cycle sees the developer using the compiler as a debugger. This type of *debugging* at the design level is not the aim of TOAD and is a recognised, yet unavoidable problem that occurs when empowering developers with unrestrained access to rich evaluation tools [VJT92].

The students explicitly asked for a partial ordering on design heuristics. They felt that by prioritising the breached heuristics they would save time and effort during evaluation by first addressing those problems that were considered the most damaging to the design. Heuristic priorities described in section 4.6.4 were incorporated into TOAD to support this requirement.

The few students that found TOAD a hindrance had implemented designs in which only a few heuristics were breached. These students had prior design experience and implemented structurally sound designs in which TOAD offered limited feedback. Although TOAD provides positive feedback, it was insufficient for those designs that did not present many design problems. However, it should be noted that the lack of breached heuristics could result from an incomplete design, and that a design which does not breach any heuristics can still be a semantically poor one.

6.2.4 Discussion

In answer to the core questions presented in section6.2.1.3, we have shown the common problems that newcomers to object technology encounter, those design heuristics which can automate their detection, the feedback they receive and the advice that was suggested. It was also interesting to find certain combinations of breached heuristics that collectively highlighted specific types of (problem) classes. For example, (`CU1`, `CU2`, `CC1`, `CC3`) for God classes and (`CC4`, `CC1`, `CA1`) for inert classes.

The majority of students found TOAD useful during the design process and actively applied its advice to their class models. Furthermore, the use of TOAD did not disrupt the teaching method in supplying an *additional* aid for educating students in elements of object technology. TOAD was viewed as an optional design aid that virtually all students (91%) applied at their discretion.

The students found that because the design heuristics were based around specific design problems, they were easy to understand and therefore apply. They reified the paradigmatic concepts presented in lectures using examples taken directly from the context of their own designs. A few of students, based upon their inaugural experiences designing object-oriented software, suggested a number of design heuristics that they thought would be useful to them and future learners of object technology, a few of the more informative suggestions are listed below:

> *"When designing a class, design its attributes and* ***probable*** *methods at the* ***same*** *time and add them to the design."*
>
> *"Try and split classes up into smaller ones early."*
>
> *"Spend a lot of time on object design diagrams with perhaps a less detailed class diagram and only add new methods if absolutely necessary."*
>
> *"Beware of God classes and don't ignore them."*
>
> *"Always check for the repetition of methods in different classes. If many different classes have similar functionality, the quality of the design is not as good as it could be and should be re-thought."*

Upon reflection, the common design message that *classes should keep data with the methods that act upon them* is an important one and should have been documented as a design heuristic. The removal of this from the reporting facilities in TOAD and into the heuristic catalogue is left for future research.

Design heuristics need to be automated to guarantee their continued use throughout software development. TOAD was able to uncover problems that individuals failed to identify either due to the size and complexity of their class models *or* through their lack of design experience.

The benefits of design heuristics also continued off-line. As the student designs progressed and they made more use of the design heuristics, they began expressing their problems in terms of *heuristics*. The design heuristics provided a *vocabulary* for describing design problems. The students voiced their design concerns in lectures using heuristics which provided a richer and more descriptive vehicle for articulating problematic design models. Students were unwittingly discussing designs at the design level and gradually viewing objects and their classes outside the context of a particular programming language.

Finally, on the qualitative side of evaluation, a rough idea of the expected size of the final system, typically 20-25 classes, factored highly into how students' approached a design problem. This presented them with an invisible ceiling that served as a rough guide for determining how close they were to completing the system.

6.3 TOAD: a design reviewer

This section evaluates two medium to large OO systems using TOAD. The first system is TOAD itself. The identified problems and suggested solutions are moderated by our intimate knowledge of the TOAD architecture and the design policies behind its class structures. The second system is the Java programming language v1.0. We demonstrate that TOAD is an effective tool for reviewing OO software and establish the limits of its application by clarifying what TOAD provides for the designer *and* how the designer makes use of its feedback.

6.3.1 Reviewing TOAD with TOAD

For clarity, the system will be referred to as TOAD and the classes used to implement it as design *T*. A summary of *T* in section 6.3.1.1 is followed by a breakdown of which class-based and which inheritance-based heuristics were breached in sections 6.3.1.2 and 6.3.1.3 respectively. An architectural overview of *T* is given in Figure 5—5.

6.3.1.1 An overview of TOAD

Design *T* has 120 classes of which 105 are organised around 11 inheritance hierarchies. The design report facility in TOAD (see Appendix B) summarised a number of common, system-wide problems occurring within *T*. Firstly, there were no *methodless* classes and 4 concrete *empty* classes that were subsequently identified as belonging to third-party libraries used by *T*. For the 28 concrete *dataless* classes, TOAD noted that all 28 were part of an inheritance hierarchy suggesting that they had inherited their behaviour and/or attributes from their ascendent classes. These initial observations highlighted the prominent use of inheritance throughout *T*.

The design report marked classes `CString` and `Widget` as common interface collaborators *and* parts, where common was defined as influencing 10%-20% of the total classes within *T*. Upon lowering the definition of common to 5%-10%, a number of problems surfaced regarding *T* and the feedback given by TOAD.

The omission of pervasive generic classes such as `UnBoundedBagPtr` and `UnBoundedSetPtr` (see Figure 6—8) as common parts and/or interface collaborators signalled an inaccuracy with the feedback mechanisms in TOAD. Although inheritance recognises contributions by generic classes, only their instantiated versions can participate in aggregation and using relationships. Consider, for example, a generic class `C` that has instantiated classes C_i, C_j and C_k. For `C` to be recognised as common requires either C_i, C_j or C_k to be common. However, it is `C` that is responsible for providing the interface *and* the implementation for classes C_i, C_j and C_k. Ultimately, generic classes are not perceived as common interface collaborators or common parts within TOAD. Furthermore, generic classes do not participate in the evaluation procedures for the aggregation and using relationship heuristics. As a result, the generic classes `UnBoundedBagPtr` and `UnBoundedSetPtr` are important key classes in design *T* that were not indicated as such by TOAD.

A number of expected common interface collaborators did appear within *T*. These included `TTOADClass` and `TIterator`: the former used to model classes within *T* and the latter for iterating over data structures such as bags, sets and lists. The presence of `DesignClass` as a common collaborator highlighted a problem in *T*. `DesignClass` is a low-level implementation class that `TTOADClass` purportedly encapsulates. The fact that it appears as a common interface collaborator suggests that`TTOADClass` failed to achieve its design objective. Finally, TOAD reported that 3 out of the 13 root classes in *T* were concrete. These concrete root classes together with`DesignClass` provided the starting point for further analysis with TOAD.

6.3.1.2 Reviewing class structures

Table 6—4 lists the class-based heuristics that were breached in*T*. The third column displays the number of classes that were breached by the given heuristic, where the last column gives the threshold, above which the heuristic is considered to be breached.

Id	Heuristic	# of classes that breached the heuristic	Threshold
CA1	The aggregate should limit number of aggregated	10	4
CA2	Restrict access to the aggregated	3	0
CC1	Limit the number of class methods per class	21	15
CC2	Limit the number of attributes	7	6
CC3	Limit the number of messages an object can receive	34	10
CC4	Reduce the number of complex methods per class	1	0
CC6	Hide all implementation details	3	0
CI2	Prevent over-generalisation on the parent class	1	6
CU1	Limit the number of object collaborators per class	11	4
CU2	Restrict the visibility of interface collaborators	11	90% public

Table 6—4 : Class-based heuristics in TOAD

The statistics for the average class is listed inTable 6—5. The average class in *T* falls within all the appropriate thresholds given inTable 6—4.

Average number of aggregated per class	1.54
Average number of messages an object can receive	8.86
Average number of attributes per class	1.91
Average number of collaborators per class	3.31
Average number of methods per class	11.56

Table 6—5 : Average class statistics within TOAD

From the design report `CString` and `Widget` were seen to be both common parts and common interface collaborators. Table 6—6 lists those heuristic statistics from Table 6—4 that were affected by declaring `CString` and `Widget` as simple types[1]. In doing so the designer is declaring that `CString` and `Widget` are as reliable and stable as simple types in the target programming language.

Id	# of classes that breached the heuristic	# of classes that breached the heuristic excluding the `CString` and `Widget`
CA1	10	3
CC1	21	20
CC3	34	33
CU1	11	2
CU2	11	2

Table 6—6 : Class based heuristics excluding simple types

The removal of `CString` from the design had a profound effect on the `CA1`, `CU1` and `CU2` heuristics. This lent more credence to the design report's suggestions to ensure that `CString` was stable and well-encapsulated to prevent modifications to `CString` incurring large system-wide changes to *T*. From an evaluation perspective, in the absence of trusted types such as `CString`, TOAD will be able to concentrate solely upon user-defined (complex) classes in *T*.

The generic classes `UnBoundedBagPtr` and `UnBoundedSetPtr` were responsible for breaching heuristic `CU1`. Instigating the recommendations of

`CU1` to reduce their collaborations would have been too costly to enforce due to the pervasive nature of these classes. However, the suggestions given by `CU2` to remove some of the collaborators of these generic classes from the public scope proved to be a more cost-effective design solution.

3 user-defined classes used inheritance to share data amongst their descendent classes (`CA2`, `CC6`). `BasicComponent` shares GUI elements amongst its descendent components (seeFigure 6—9), where `UnBoundedBag` and `UnBoundedSet` share their `List` implementations amongst their descendent containers (see Figure 6—8). `BasicComponent` applied the feedback given by the design heuristics`CA2` and `CC6` which was to provide an accessor method to its internal representation.

[1] It should be noted that `Widget` did not have any affect upon these figures as its use was confined to the user interface classes within *T*.

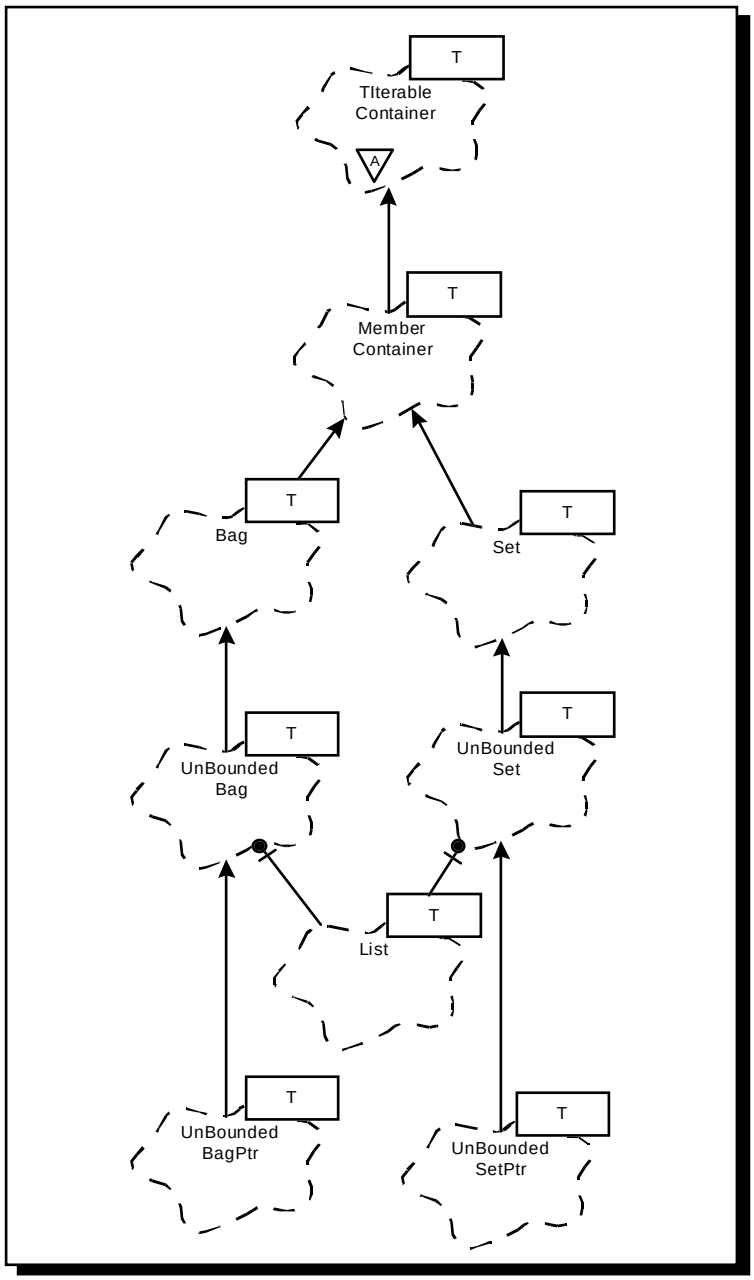


Figure 6—8 : A hierarchy of generic, iterable containers

`UnBoundedBag` and `UnBoundedSet` instead chose to apply the advice suggested by heuristic `RI5` and removed their `List` instance variable. This meant that all descendent classes of`UnBoundedBag` and `UnBoundedSet` had to add `List` as an instance variable to their definitions *and* implement the appropriate behaviour. However, it also meant that these descendent classes were free to change their internal representation using alternative data structures without incurring changes upon `UnBoundedBag` and

`UnBoundedSet`. Moreover, new descendent classes can now be defined by reusing purely the interfaces of `UnBoundedBag` and `UnBoundedSet` which are no longer bound to a specific (`List`) implementation.

Of the remaining class-based heuristics that were breached, classes with large bandwidths (`CC1`, `CC3`) and numerous attributes (`CC2`) were typically GUI components in *T*. These classes were rich in behaviour and maintained complex state information inherent in the GUI components that they modelled. However, the breaching of `CC2` and *not* `CA2` (user defined parts) suggested that these classes were not making effective use of aggregation (see section 4.5.2).

Although the GUI classes in TOAD made good use of inheritance to incrementally add data and behaviour, the resultant leaf node classes ultimately comprised of numerous *simple* instance variables. According to the heuristic class model (see section 4.5.1), `CC2` documents that modelling complex state with large numbers of instance variables gives rise to intricate interactions between class attributes and introduces dependencies between the class methods that modify them. Instead, the designer should create smaller, reusable abstractions that group together instance variables when modelling complex state information. This would increase the maintainability of the original class, reduce the amount of implementation inherited from ascendent classes and build a more robust aggregation hierarchy. The GUI classes within TOAD presented a good example of where the apparently good intentions of inheritance were to the detriment of design and where aggregation would have been a more effective design technique.

6.3.1.3 Reviewing inheritance structures

`TAnalyser`, `TSimpleClassItem` and `BasicComponent` are concrete roots within *T*. The first two classes applied TOAD's suggestions and promoted their interface-specific information into abstract classes. From Figure 6—9 we note that `BasicComponent` did not take this approach. `BasicComponent`

provides default behaviour for all GUI components, such that its methods are polymorphic but not abstract. However, in the absence of an abstract method, the C++ class `BasicComponent` could not be labelled as abstract. On this occasion, limitations in the language caused TOAD to misinterpret the intentions of `BasicComponent`.

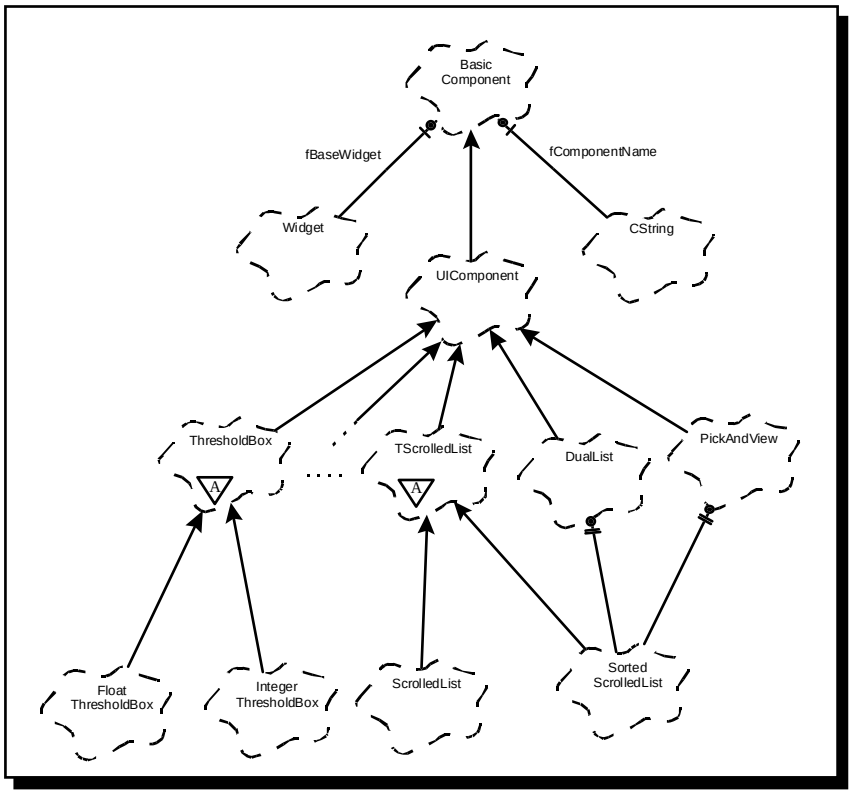


Figure 6—9 : A hierarchy of GUI components

`UIComponent` was the only over-generalised parent class in *T*. A child class of `BasicComponent`, `UIComponent` provided additional default behaviour for all its 23 descendent classes. Given the design context, `UIComponent` was not an over-generalised parent class but an interface that all GUI components used within *T*, and like `BasicComponent`, should have been abstract.

This section highlighted a number of key problem areas that TOAD encountered during its evaluation of *T* and some of the design solutions it suggested. Not all the advice given by TOAD was applied to *T* and some

advice that could have been given was not. However, TOAD offered an objective and detailed perspective on *T* for second checking its class structures. A number of oversights and unexpected class collaborations were identified and appropriately dealt with. The next section evaluates the Java programming language v1.0. TOAD is used to interrogate its design architecture to gain a deeper understanding of those problems designers might encounter using Java. Finally, section 6.3.3 discusses the broader issues that have been raised by evaluating medium to large OO software with TOAD.

6.3.2 Reviewing Java v1.0.2 with TOAD

The Java programming language is comprised of a rich set of APIs and class frameworks. This section applies TOAD to Java to determine whether design heuristics are useful for reviewing intricate class structures.

6.3.2.1 An overview of Java

51 of the 59 concrete *dataless* classes in Java are descendent classes of `Throwable`. `Throwable` embodies state information and behaviour needed for reporting errors that occur within executing Java applications. Its descendent classes typically have two methods responsible for creating their objects at which time their state information is passed in as arguments. The resultant `Throwable` hierarchy is comprised of dataless classes and a single *empty* class, `ThreadDeath`.

The remaining 8 dataless classes were found in the `java.util` and `java.lang` packages. These classes export methods belonging to third-party APIs. Because these APIs encapsulate their own state, classes that import API functions need only provide data for the methods that they*define*. With the majority of the methods in these 8 dataless classes declared as API functions, the was no need for them to have instance variables of their own.

Java has six common interface collaborators (`Image`, `Dimension`, `String`, `Object`, `InputStream` and `OutputStream`) and one common part (`String`). It has no *methodless* classes.

6.3.2.2 Reviewing class structures

Table 6—7 lists the class-based heuristics that were breached within Java.

Id	Heuristic	# of classes that breached the heuristic	Threshold
CA1	The aggregate should limit number of aggregated	6	4
CA2	Restrict access to the aggregated	19	0
CC1	Limit the number of class methods per class	42	15
CC2	Limit the number of attributes	30	6
CC3	Limit the number of messages an object can receive	54	10
CC4	Reduce the number of complex methods per class	12	0
CC6	Hide all implementation details	47	0
CI1	Limit the use of multiple inheritance	24	0
CI2	Prevent over-generalisation on the parent class	11	6
CU1	Limit the number of object collaborators per class	23	4
CU2	Restrict the visibility of interface collaborators	23	90% public

Table 6—7 : Class-based heuristics for Java

222 classes and types define Java of which `Object` is the root of a single inheritance hierarchy. The statistics from Table 6—8 illustrate that the average class within Java does not breach the appropriate heuristics listed in Table 6—7. However, from Table 6—7 we also note that a number of class-based heuristics were breached.

TOAD reported that a large number of Java classes failed to conceal their internal representation (`CA2`, `CC6`) and that their implementation comprised of numerous instance variables (`CA1`, `CC2`). Problem feedback illustrated that the majority of the public instance variables were in fact *constants.* In Java, everything must belong to a class and constants cannot be private if they are to be visible to other classes. TOAD assumed that all attributes belonging to a class were instance variables which caused the design heuristics `CA1`, `CA2`,

CC2 and CC6 to be breached in circumstances when they should not have been.

Average number of aggregated per class	0.96
Average number of messages an object can receive	7.69
Average number of attributes per class	2.72
Average number of collaborators per class	2.90
Average number of methods per class	9.14

Table 6—8 : Average class statistics in Java

13 out of the reported 47 *unencapsulated* Java classes were found to be exporting public instance variables. For example, the`Dimension` class had two public instance variables `height` and `width`. However, `Dimension` also provides accessing methods to these public instance variables thereby presenting conflicting mechanisms for modifying/reading the state of its objects.

The design heuristics (CC6, CA2) brought attention to Java's extensive use of protected instance variables for sharing code amongst its descendent classes. It would appear that Java does not enforce encapsulation for inheriting clients. Nevertheless, changes made to the implementations of these unencapsulated classes can have costly repercussions on their descendent classes. Viewing encapsulation for inheriting clients in the same way as a instantiating clients is becoming an increasing popular design practice [WW89][Sn89][Ri96] and is even more important within frameworks, such as Java, that deal extensively with inheritance.

TOAD does not explicitly model *types*. Instead, TOAD models types as abstract classes devoid of implementation that contain purely abstract methods. As a result, 24 classes breached the multiple inheritance heuristic CI1. Although problem feedback within TOAD recognised that these classes were in fact implementing multiple types and inheriting from a single class, such knowledge should reside in the model and not the tool.

The CI2 heuristic that determines whether a class is over-generalised again brought issues regarding class and type to the fore. For example, as the root in Java, Object has numerous child classes. However, TOAD reported another 10 classes that were considered to be over-generalising on the parent class. Upon closer inspection with TOAD, 3 of these classes were types that were legitimately exporting interfaces for classes to implement. Of the remaining 7 potential problem classes, 2 were abstract classes providing default behaviours to their descendent classes and the remaining 5 were used to group errors and exceptions within Java. By not explicitly modelling types within TOAD, the evaluation process created unnecessary noise for the CI1 and CI2 design heuristics.

In general, two kinds of Java classes were considered by TOAD to be highly communicative: GUI components and networking classes. These classes had a large number of (abstract) interface collaborators (CU1, CU2) and no instance variables (CC2). For these classes, state information was being taken out of the class and passed in as parameters to its methods (cf. The flyweight design pattern [GHJ+94]).

6.3.2.3 Reviewing inheritance structures

Object is the only root class in Java. However, because TOAD models types as abstract classes, type hierarchies also appeared alongside Object as root classes. These type hierarchies did not breach any of the inheritance based heuristics because the underlying heuristic model described in section 4.5.4 conforms to this method of inheritance. Applying this model sees class hierarchies defining the upper most classes as type definitions. Upon descending the hierarchy, classes incrementally add implementation details. This sees the gradual specialisation of types to intermediary abstract classes with data and finally into concrete classes at the leaf nodes. Table 6—9 lists the inheritance based heuristics breached within the Object hierarchy.

Id	Heuristic description	Breached

RI1	The inheritance hierarchy should not be too deep	yes
RI2	The root of all inheritance hierarchies should be abstract.	yes
RI3	For deep hierarchies upper classes should be type definitions	no
RI4	Minimise breaks in the type/class hierarchy.	no
RI5	Strive to make as many intermediary nodes as possible abstract.	yes

Table 6—9 : Inheritance-based heuristic automated within TOAD

Although the `Object` inheritance hierarchy has 176 descendent classes, its root class is concrete (`RI2`). `Object` provides default behaviour that all its descendent classes exhibit but reasons for being concrete were not forthcoming. Furthermore, its also appears that the `Object` hierarchy is composed of numerous concrete classes (`RI5`). Breaching this heuristic suggests that the inheritance hierarchy is not a specialisation hierarchy. However, the `Throwable` classes were creating a lot of noise during the evaluation of the `Object` hierarchy. `Throwable` classes contribute limited behaviour to Java, providing only a standardised mechanism for documenting errors and exceptional conditions within Java programs. By temporarily removing them from the evaluation process, the `Object` hierarchy was shown to possess an equal number of intermediary abstract and concrete classes (see Table 6—10). As such, the `Object` hierarchy would not have breached the `RI5` heuristic.

Descendent classes of the `Throwable` and `Component` hierarchy caused the `Object` hierarchy to reach a depth of 6 (`RI1`). For `Throwable` classes this enabled the comprehensive classification of errors that can occur within a Java program. Where deep `Component` classes are representative of rich GUI components that had been produced incrementally.

The heuristics in Table 6—9 suggest that Java makes good *strategic* use of inheritance. However, the class-based heuristics highlight *tactical* problems in the hierarchy such as parent classes not encapsulating their implementation and classes that inherit from concrete parents. An example

of the latter sees `Stack` inheriting from the concrete parent `Vector`. `Vector` has a large number of methods and upon realising that `Stack` has only a few, TOAD reported that `Stack` might not be a specialisation of `Vector`. Moreover, TOAD noted that the methods implemented within `Stack` were not overriding those within `Vector`. TOAD concluded that `Stack` is not a specialisation on `Vector`. In fact, `Stack` objects in Java can remove and add elements at any point in the stack which directly compromises the expected behaviour (`push` and `pop`) of a `Stack`. In pursuing the principles and concepts of object technology to their full conclusion, TOAD was able to identify and report on a serious discrepancy in the expected behaviour of `Stack` objects within Java.

	Non Leaf Nodes		**Leaf Nodes**	
	With Throwable classes	Without Throwable classes	With Throwable classes	Without Throwable classes
abstract	13	12	10	10
concrete	21	12	132	87
total	34	24	142	97

Table 6—10 : `Object` inheritance hierarchy

To summarise, TOAD reported that Java promoted a type-oriented view of interfaces and made good strategic use of inheritance. However, the unencapsulated nature of a large number of its classes (protected instance variables) and behavioural anomalies (e.g. `Stack`) demonstrate that there is still much more effort that needs to be invested into the Java classes.

6.3.3 Discussion

The use of design heuristics for reviewing large OO systems places less emphasis on learning during evaluation, focusing more upon problem detection and feedback. TOAD reports on the strategic and tactical design policies within a system by analysing heuristics both individually and collectively. For example, tactical problems associated with GUI classes

pertain to large class bandwidths (`CC1`, `CC3`), numerous instance variables (`CC2`, `CA1`) and over-generalised parent classes (`CI2`). However, viewing these breached heuristics collectively reveals the domain-specific nuances of GUI components regardless of the type of OO system. Such domain-specific information needs to be modelled directly within TOAD by recognising how certain groups of heuristics interact.

Design heuristics document a finite number of solutions. As a result, the suggestions presented during evaluation became repetitive once the designer had fully assimilated the rationale behind the heuristic and how to effectively apply its solution(s). As expected, the benefits attributed to educating designers diminished once the designer had learnt the given design concept that the heuristic embodied. To the reviewer, a design heuristic is an effective problem detection/feedback mechanism to help them reason objectively about their design by returning to OO first principles.

Design heuristics, in the assumed role of problem detector, on occasion indicated problems that were actually the idiomatic arrangement of classes or common patterns applied during software design. Consider, for example, the use of the flyweight pattern [GHJ+94] in a number of the GUI components in Java. This design pattern takes state information out of the class and passes it in as arguments to its methods. In design heuristic terms, this results in a large number of (public) interface collaborators (`CU1`, `CU2`) within the protocol of dataless classes (`CA1`, `CC2`). Without knowledge of how design patterns are applied within software, TOAD will continue to ignore patterns and instead present them as a collection of breached design heuristics.

Design heuristics provoke a closer examination of specific class structures. They pinpoint where in the hierarchy further analysis should commence and provide reasons why these problems are occurring in software. The automation of design heuristics within TOAD accentuated their inherent benefits. Automation permits an unbiased and more detailed inspection of all class structures. By mechanising the deployment of heuristics, large software

designs were assimilated quickly and feedback on the located problems presented in a concise manner. This way (un)familiar class models can be articulated in a comprehensive and interactive manner.

6.4 Summary

In this chapter design heuristics were shown to be effective vehicles for novices learning OOD and for experts reviewing large software designs. Furthermore, TOAD proved to be an important tool for automating their deployment during the design evaluation process. The next chapter presents our conclusions and future directions for our research.

Chapter 7

Future research and conclusions

7. General conclusions

A novel system has been conceived, designed and built that allows learners to evaluate object-oriented designs. Design evaluation not only involves identifying potential difficulties in software but receiving timely and instructive feedback of these discovered problems and directions on how to set about rectifying them. The system also empowers experienced developers with an intelligent tool to help them reason about common design problems within large and complex OO software.

The design heuristic catalogue and the TOAD system developed during this research demonstrate how design experience can be transferred from the expert to the novice in an effective manner. An informal approach to the design evaluation problem resulted in a system that automated the deployment of heuristics that was both practical and useful for evaluating OO software. Design heuristics document common problems and their respective solutions within OO systems. They are founded upon models of maintainability stemming from metric research that aim to contain change within software designs. The resultant catalogue structures and categorises heuristics by providing a handbook of small, simple and legible design problems encountered by software developers performing OOD.

The TOAD model focuses purely upon aspects of interface to ensure the timely and constructive evaluation of software *designs*. We submit that the inter-modular properties of classes provide expressive and comprehensive information to designers on both the structural integrity and architectural properties in OO systems.

Our results illustrate that the majority of learners that used TOAD found it useful for detecting problems, issuing feedback and suggesting solutions. For the expert reviewing OO systems, TOAD proved to be an effective tool for pinpointing design weaknesses and providing objective perspectives on design architecture at both the strategic and tactical level.

The following sections highlight some of the important issues raised during this thesis together with areas that are considered innovative.

7.1 The software design evaluation problem

Chapter 3 defined design evaluation and addressed its associated problems. We contend that design evaluation embodies elements of learning. This requires certain degrees of experience to be present before the process can reason about the design under evaluation. We demonstrated that experience needed to be captured as a series of small, simple and legible fragments of design expertise so that they could be assimilated in their entirety by the learner and applied incrementally. Metrics are not the ideal vehicles for documenting design expertise in this manner (see section3.4.1). Although heuristics have the potential, they rarely deliver on this promise in the literature (see section 3.4.3). Our research has further clarified the relationship between metrics and heuristics. More importantly, this thesis has taken preliminary steps to not only identify, document and automate design expertise but to evaluate how well it was transferred to learners in the form of heuristics. We contend that design heuristics are effective in this role.

7.1.1 Metrics and heuristics

Heuristic use metrics. Metrics provide heuristics with answers to questions such as is x complex, is y large and is z maintainable. Heuristics articulate this feedback with knowledge of object technology and experience of how objects are actually applied during software design. Heuristics provide the judgement on what course of action to take in light of x being complex, y being large and z being maintainable. In doing so, heuristics balance the

theoretical ideas supported by metrics against the way objects are actually used by software designers.

7.1.2 An informal approach

Chapter 4 presented the design heuristic catalogue and outlined an informal approach for applying heuristics: problem detection, problem feedback and solution guidance. Problem detection, in treating all classes within a design as equal, identifies potential difficulties in the software. Problem feedback assimilates the context of the class before judging whether the detected anomaly is really a problem in the design. Solution guidance is subsequently employed depending on whether the feedback phase decides that the identified anomaly is actually a design problem.

Upon reflection, the problem feedback phase that reasons about these potential problems would be better served by an expert system that embodies the design experience required to support this objective. An expert system would simplify the *maintenance* of design knowledge and be more effectively implemented using an appropriate programming language such as Prolog or LISP, or an expert system shell.

7.1.3 Integrating design heuristics

This research has produced a *handbook* of common design problems that learners can refer to when performing OOD. This handbook documents design heuristics [Gi97b] and is constantly being updated as new design problems are found and existing ones are modified. Heuristic forms provide a standardised way to capture design expertise. These forms minimise the amount of effort required to document future design problems and are an important starting point for automating their deployment within tools.

The TOAD system is a pragmatic design aid for use by those learning the technology and/or reviewing large class models. It has been successfully incorporated into the teaching process at the University of Nottingham and the vast majority of students that applied TOAD have reported that it aided

them during their design work. In light of the success met with automating heuristics, section 7.3.2 reports on extensions to TOAD to improve its use during design and to make it available to those institutions currently teaching object technology.

7.2 Discussion: meeting our objectives

In this section we determine whether we have met our thesis objectives set as a series of core questions in chapter 1. Firstly, we have shown design heuristics to be effective during software design. Chapter 6 reported on how students successfully applied heuristics and how they can be used by developers to better articulate large OO systems. In addition, we have demonstrated how heuristics are automated within TOAD and outlined an informal design evaluation process that oversees their deployment (see section 4.6).

Secondly, as reported in section 7.1.1, we have elucidated the relationship between metrics and heuristics. However, it is not a case of what advantages heuristics have over metrics, or vice-versa. Instead, each has benefits and limitations as documented in chapter 3. We have found that informal mechanisms of evaluation are better suited for driving informal processes. However, these informal techniques benefit greatly from formal evaluation mechanisms; in our research this saw heuristics employing software metrics.

Chapter 6 also reported on how students began to express design difficulties in terms of heuristics. This saw students thinking at the design level and concentrating on how objects collaborate, rather than the means by which to implement them. This subtle shift in attitudes signalled that heuristics do provide a good vehicle for documenting and expressing design problems and that the success met by users applying TOAD illustrates that heuristics can be objectively specified.

7.3 Future work

There are a number of related research areas that would benefit by extending our work. These are described in conjunction with ways that additional implementation efforts invested in TOAD could produce a feasible, working commercial system. Section 7.3.1 describes how extensions to the heuristic catalogue would directly improve its design evaluation capabilities. Section 7.3.2 reports on how integrating certain features into TOAD would make it a marketable and more effective system for use during software development. Finally, section 7.3.3 describes current research efforts to visualise *and* evaluate class models within collaborative virtual environments.

7.3.1 Extensions to the heuristic catalogue

The heuristic catalogue is a working document that provides a foundation upon which future research efforts can build [Gi97b]. Appendix D details the collaborative research efforts that continue to review and enhance the design heuristic catalogue. Based upon our experience applying and writing heuristics, this section highlights the principal areas of interest that need to be addressed to improve the catalogue's evaluation capabilities and to ensure that their deployment during OOD is both comprehensive and effective.

7.3.1.1 Mapping heuristics onto software patterns

The majority of the heuristics in the catalogue document their solutions using ad-hoc descriptions. A more comprehensive approach to implementing problem and solution pairs would see every heuristic, where possible, linked to a *design pattern*. Research by Riel [Ri96] describes design transformation patterns, or meta-patterns, that are comprised of a source pattern, motivating heuristic and target pattern as illustrated inFigure 7—1.

Design transformation patterns use the *motivating heuristic* to highlight potential design problems in OO software. The breaching of a motivating heuristic signals to the designer that a certain arrangement of classes occurs within the software which is described by the *source pattern*. The *target*

pattern documents how the classes should ultimately be organised in the final design. Unfortunately, the design transformation patterns presented by Riel are largely qualitative.

The work in this thesis has both mechanised the process of applying motivating heuristics and during problem feedback, exacted in-depth analysis procedures over the identified source pattern. However, our research has made limited use of design patterns for documenting solutions to the identified problems reported by heuristics. Future work into documenting more design solutions as software patterns is an important requirement for improving the usability of heuristics within the catalogue.

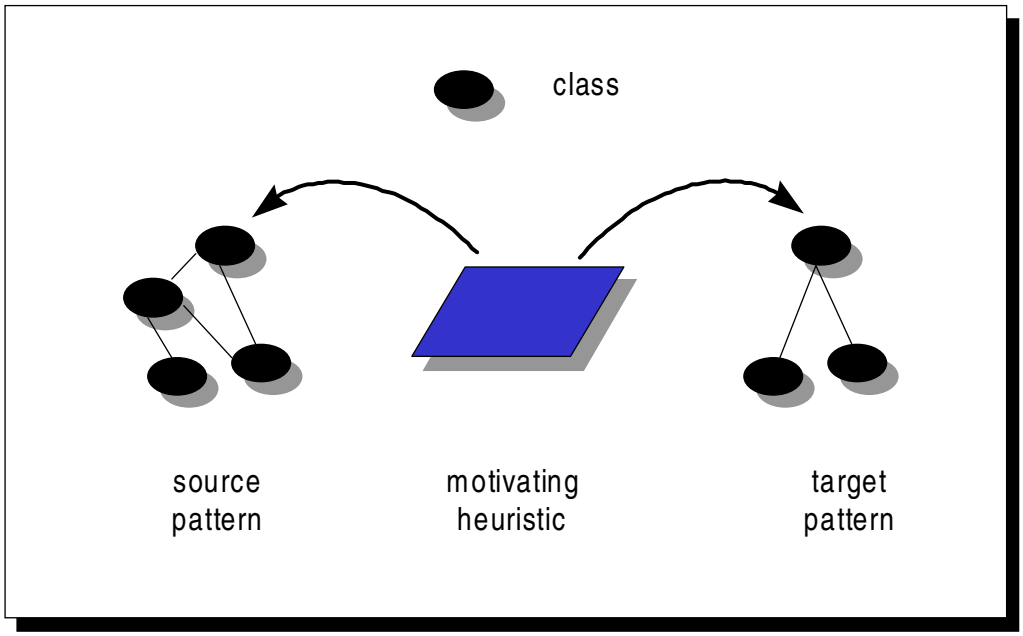


Figure 7—1 : A design transformation pattern

7.3.1.2 Heuristics for class clusters

The design heuristics in the catalogue are pre-dominantly based at the class level. Class heuristics evaluate design problems pertaining to a*single* class and relationship heuristics focus upon groups of classes participating in the *same* class relationship. At present, there are few heuristics aimed at the cluster level. A cluster is a group of highly cohesive, logically related classes

with a common functional goal. Classes within a cluster can partake in more than one type of class relationship, thereby making cluster level heuristics the most complex type of heuristic to document.

Cluster heuristics evaluate the communication paths between classes within a single cluster (intra-cluster) and how clusters interact with their peers (inter-cluster). The modelling of clusters becomes progressively more important as the size of the OO systems increases. Cluster heuristics, in providing a coarser level of evaluation, serve to supplement the finer-grained class and relationship heuristics. Support for clusters within TOAD has been implemented in readiness for the inclusion of cluster heuristics to evaluate their presence within OOD documents.

7.3.1.3 Vocabulary for heuristic inter-dependencies

Modelling inter-dependencies between heuristics was achieved through *concept* and *quality* categories reported in section 4.4.2. These categories within TOAD illustrated how heuristics can be applied collectively but the inter-relationships between them were expressed in an ad-hoc manner. Furthermore, maintaining these categories involves constantly adapting code deep within TOAD. Is there a means by which the communications between design heuristics can be described more definitively? Can a heuristic language be specified that offers developers a way to describe groups of collaborating heuristics? If so, can this heuristic language provide a logical representation of the coded categories, thereby permitting users of the catalogue to define their own heuristic inter-dependencies without knowing the intricacies of TOAD's implementation? These are interesting questions for future research to address.

Specifying a heuristic language would permit higher level and more involved design problems to be expressed in terms of primitive heuristics with minimal effort. For example, TOAD categories defined using a heuristic language would be more adaptable/extensible by third party developers using the heuristic catalogue. This would enable the users of TOAD to express design

problems that *they* commonly encounter in terms of *existing* heuristics. Inter-connecting heuristics involves knowing their expected inputs and delivered outputs and interpreting what is considered good and what is not. Research efforts into specifying a heuristic language are left for future research.

7.3.2 Extensions to TOAD

In light of some of these future extensions to the heuristic catalogue and user demands for more intelligent environments for evaluating software designs, this section presents some proposals for enhancing TOAD to meet these requirements.

7.3.2.1 From tools to environments

TOAD is a prototypical tool for evaluating OOD documents, it is not a learning environment. The migration path from *tool* to *environment* has a number of requirements:

- greater access to background material on object technology such as methodology, products, case studies, and so on;
- improved feedback mechanisms with more design examples complete with annotated solutions;
- multi-media resources and links;
- collaborative, group-oriented features.

A number of learning tools are emerging that immerse learners in rich, informative and interactive environments for developing and adapting OO software with the focus on education [Go97]. A move to address these requirements and integrate their ideas within TOAD would allay concerns voiced by educators about providing learning environments as opposed to tools [Do96], where tools are viewed as only a partial solution to the teaching problem.

7.3.2.2 More support for heterogeneous design processes

The degree of method and language independence provided by TOAD could be improved within additional research efforts. The class description language that describes the static, class-based properties of an OOD document resulted from examining the Booch, OMT and Wirfs-Brock design methods and the C++ and Java programming languages. The class description language generalised the class properties that these methods and languages modelled. The only way a more comprehensive version of the class description language can be realised is by implementing document translators for yet more OO programming languages (e.g. Smalltalk, Eiffel, CLOS) and OOD methods (e.g. UML, Coad/Yourdon, Catalysis). In the short term this will also increase the number of design methods and languages supported by TOAD and the number of developers that could then gain access to the automatable design heuristic catalogue. In the long term, these new additions would produce a robust generic model of the static, class-based properties of an OOD.

7.3.2.3 Round-trip development

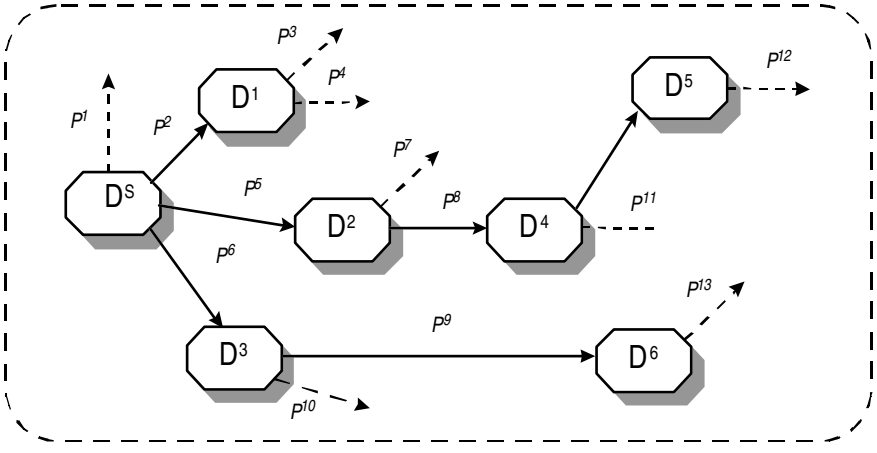
Recent advances in CASE has seen software vendors specifying application programming interfaces (APIs) to their tools. These APIs permit developers to access design model attributes, modify the tool's user interface and build third party, plug-in components to extend its original functionality; CASE APIs permit developers to implement what vendors omit.

An important ongoing project is to implement TOAD as a plug-in component for a number of CASE tools. Currently, evaluation mechanisms within CASE tools are largely syntactic in nature, exercising simple grammatical rules between design elements and their relationships. For example, the notification of an illegal placement of an inheritance relationship between an instantiated class and its generic template. A more accessible, semantically aware evaluation system would empower the class designer with design expertise in the form of heuristics. The implementation of pluggable design

expertise with TOAD has far reaching consequences. In short, the designer would not have to leave their current development environment or learn new tools in order to make use of the design experience that heuristics embody.

Another important consideration is that design evaluation with TOAD is read only; a class model is presented, interrogated and feedback presented. Yet the design under evaluation is never modified. Consider a design solution space, illustrated in Figure 7—2, that has the original class model as its initial solution. Upon breaching a heuristic the developer then chooses a particular solution and applies it to the model, thereby moving it into a different state within the design space. Evaluation now proceeds using this modified class model. Using this technique, a number of different solutions can be arrived at *and* (re)evaluated within the design solution space. The designer is now at liberty to choose the most appropriate design that they arrive at within this space.

The design solution space



D^S	the initial design presented for evaluation
D^n	a new design to which a series of patterns have been applied to D^S
P^n	a specific implementation of a design pattern

When a pattern P^n is applied to a design D^n, it moves it through the design solution space. Every D^n is evaluated using heuristics which suggest new design solutions and/or design patterns to apply.

Figure 7—2 : Heuristics and patterns in the design solution space

Plug-ins permit design evaluation to take place within the *developers'* modelling environment. Furthermore, if design solutions are documented as design patterns, the automation of "…parameteurised high-level coding buildings blocks" [Co96] could see developers applying specific solutions for specific design patterns.

Recent publications [EGY97][BFV96][Wi96] in this research area are promising. They describe specialist ways to codify a design pattern within limited contexts, thereby permitting developers to progressively move their class models through the design solution space. It should be noted however, that these tools present only a few of the possibly unlimited ways in which a design pattern can be applied [Co96], thus limiting the number of solutions developers can arrive at within the design solution space.

Although these tools are prototypical, they do highlight just one area in which TOAD could benefit greatly from in the future, by research conducted now into connecting, documenting and automating the links between design heuristics and design patterns.

7.3.3 Visualising Large Software Designs

Large scale software design is inherently a group undertaking best performed off-line. In moving software designs to an on-line medium, group design using CASE tools invariably takes place in a one designer/one machine operational environment. Such pseudo-group design foregoes the collaborative and interactive benefits afforded to groups designing off-line.

Initial research by Fraser [Fr97] into collaborative programming within virtual environments produced a single workspace for multiple developers performing OO software design. POODLE (Programming Object Oriented Design Language Environment) is a prototypical system that constructs 3D models of OO software. Using TOAD to export class models and aspects of the automatable heuristic catalogue to POODLE, developers can view and

evaluate software designs, on-line, in groups, in an interactive and collaborative manner [Gi96].

In addition to the well-documented benefits attributed to virtual environments and CSCW (co-operative support courseware)[BBF+95][Ha96], visualisation also aids developers to conceptualise and heighten their assimilation rates of unfamiliar software [SK96]. Software's invisible and intangible properties are given a physical presence within a virtual design workspace, displacing unforeseen group ambiguities in delivering a single, coherent model of software architecture.

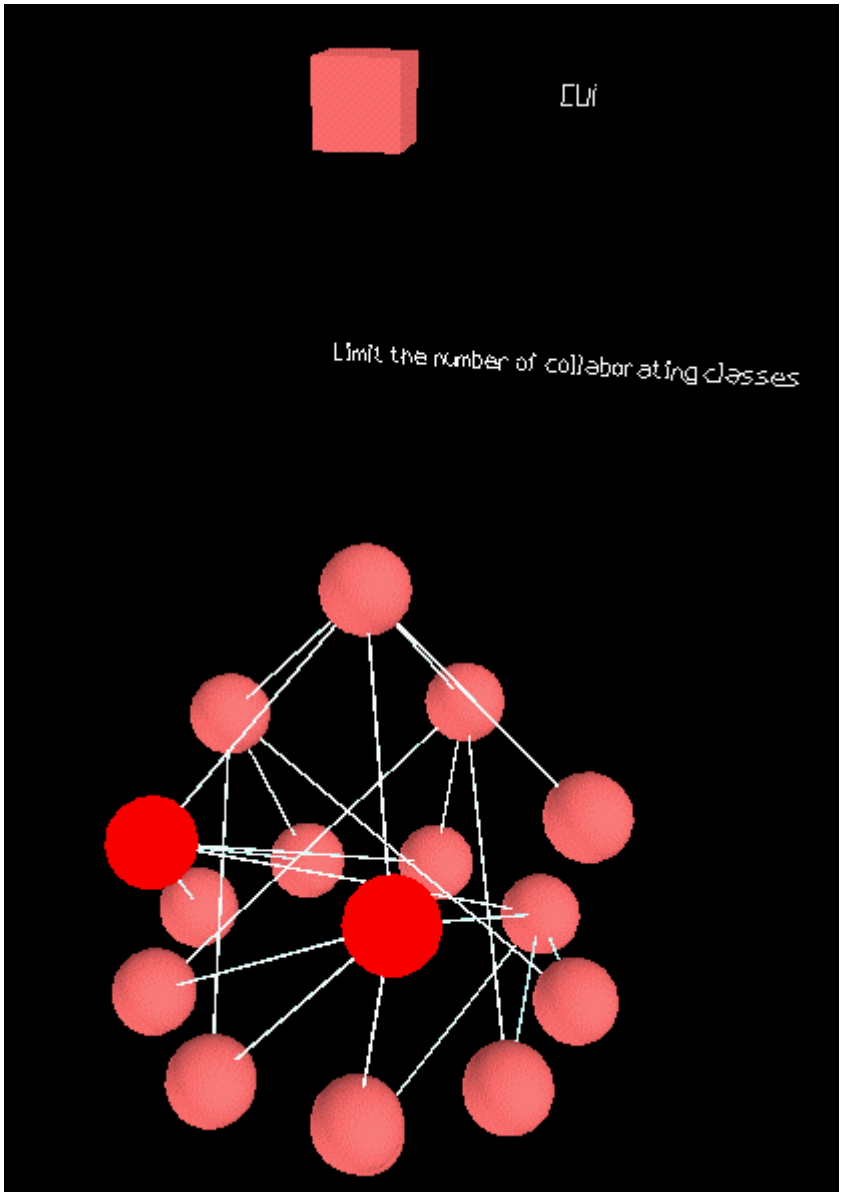


Figure 7—3 : Visualising class models with POODLE

To summarise, using both the collaborative and interactive elements of virtual environments, and TOAD''s knowledge of class models and design expertise, POODLE provides a virtual design arena, for geographically distributed developers, to review and modify large software designs in a group-oriented manner. Research efforts in this area are currently being undertaken by the Communications Research Group at the University of Nottingham.

7.4 Closing remarks

This research has demonstrated the advantages of heuristics during design. They provide an effective vehicle for describing common design problems and their respective solution(s) and a vocabulary for developers to communicate difficulties that arise in OO software. Nevertheless, learners were unwilling to apply heuristics unless provided with tool support to automate their deployment. We conclude that the use of TOAD accentuated the inherent benefits provided by design heuristics.

Finally, it has been said that good judgement comes from experience and experience from poor judgement. Fortunately, if we can harness design expertise in the form of heuristics, the poor judgement need not be that of the learner.

Bibliography

[Ab94] Abreu, F.B. "MOOD: Metrics for object-oriented design", OOPSLA'94 Workshop Paper Presentation, 1994.

[AC94] Abreu, F.B. and Carapuca, R., "Candidate metrics for object-oriented software within a taxonomy framework", Journal of Systems and Software, 26(1), July 1994, pages 87-96.

[Ab79] Albrecht, A.J., "Measuring application development productivity", Proceedings of IBM Application Development Symposium, 1979, pages 83-92.

[AM93] Avotins, J. and Mingins, C., "Metrics for object-oriented design", Internal Paper, Object Technology Group, Department of Software Development, Monash University, 1993.

[Av96] Avotins, J., "Towards an OO metric modelling method", OOPSLA'96 Workshop on product metrics, October 1996.

[BS93] Barnes, G.M. and Swim, B.R., "Inheriting software metrics", Journal of Object-Oriented Programming, 6(7), 1993, pages 27-34.

[BT75] Basili, V. and Turner, A.J., "Iterative enhancement: a practical technique for software development", IEEE Transactions on Software Engineering, 1(4), December 1975, pages 390-396.

[BSP83] Basili, V.R., Selby, R.W. and Philips, T.Y., "Metric analysis and data validation across FORTRAN projects", IEEE Transactions on Software Engineering, 9(6), November 1983, pages 652-663.

[BE81] Belady, L.A. and Evangelisti, C.J., "System partitioning and its measure", Journal of Systems and Software, 2, 1981, pages 23-29.

[BBF+95] Benford, S. D., Bowers, J., Fahlen, L. E., Greenhalgh, C.M., Mariani, J. and Rodden, T.R., Networked virtual reality and co-operative work presence: teleoperators and virtual environments, 4(4), MIT Press, ISSN1054-7460, Fall 1995, pages 364-386.

[Bi92] Bieman, J., "Deriving measures of software reuse in object-oriented systems", Proc. BCS Workshop on Formal Aspects of Measurement, 1991.

[BK93] Bieman, J.M. and Karunanithi, S., "Candidate reuse metrics for object-oriented and Ada software", IEEE Transactions on Software Engineering, 1993, pages 120-128.

[BL93] Bilow, S.C. and Lea, D., "Workshop report - processes and metrics for object-oriented software development", OOPS Messenger, 5 April 1994, pages 95-98.

[Bo83] Boehm, B., Software engineering economics, Englewood Cliff, NJ: Prentice-Hall, 1981.

[Bo86] Booch, G., "Object-oriented development", IEEE Transactions on Software Engineering, 12(2), February 1986.

[Bo95] Booch, G., "Rules of thumb", Report on Object Analysis and Design, 2(4), Nov-Dec 1995, pages 2-3.

[Bo89] Booch, G., "What is and what isn't object-oriented design", American Programmer, 2, Summer 1989, pages 7-8 and 14-21.

[Bo94] Booch, G., Object-oriented analysis and design with applications, Benjamin-Cummings, 1994.

[Bo84] Bowen, J.B., "Module Size: A standard or heuristic", Journal of Systems Software, 4, 1984, pages 327-322.

[Br75] Brooks Jr., F.P., The mythical man month: essays on software engineering, Reading, MA: Addison-Wesley, 1975.

[BFV+96] Budinsky, F.J., Finnie, M.A., Vlissides, J.M. and Yu, S., "Automatic code generation from design patterns", IBM Technical Journal, 35(2), 1996.

[Bu97] Buschmann, F., "On pattern forms", Object Expert, 2(4), May/June 1997, pages24-26.

[By94] Byard, C., "Software Beans: class metrics and the mismeasure of software", Journal of Object-Oriented Programming, 7(5), 1994, pages 32-34.

[CG90] Card, D. And Glass, R., Measuring software design quality, 1990.

[CA88] Card, D.N. and Agresti W.W., "Measuring software design complexity", Journal of Systems and Software, 8(3), 1988, pages185-197.

[Ca91] Card, D.N., "What makes a software measure successful", American Programmer, September 1991, pages 2-8.

[CCA86] Card, D.N., Church, V.E. and Agresti, W.W., "An empirical study of software design practices", IEEE Transactions on Software Engineering, 12(2), February 1986, pages 264-271.

[CPM85] Card, D.N., Page, G.T. and McGarry, F.E., "Criteria for Software Modularisation", Proc. of the 8th Inter. Conference on Software Engineering, August 1985, pages 372-377.

[CL93] Chen, J-Y and Lu, J-F., "A new metric for object-oriented design", Information and Software Technology, 35(4), 1993, pages 232-240.

[CS91] Cherniavsky, J.C. and Smith, C.H., "On Weyuker's axioms for software complexity measures", IEEE Transactions on Software Engineering, 17, 1991, pages 636-638.

[CK94] Chidamber, S. and Kemerer. C., "A metrics suite for object-oriented analysis and design using the Booch Method", A Rational Technical Paper, Rational Software Corporation, 1994.

[CK91] Chidamber, S. and Kemerer. C., "Towards a metrics suite for OO design", Proc. of OOPSLA'91, pages 197-211.

[CSM95] Chonoles, M.J., Shcardt, J.A. and Magrogan, P.J., "OO systems from a qualitative perspective", Report On Object Analysis and Design, 2(4), 1995.

[CM92] Chung, C-M. and Lee, M-C., "Inheritance-based metric for complexity analysis in object-oriented design", Journal of Information Science and Engineering, 8, 1992, pages 431-447.

[CY91] Coad, P. and Yourdon, E., Object-oriented analysis, Second Edition, Prentice-Hall, 1991.

[Co90] Coggins, J.M., "Designing C++ libraries", In USENIX C++ Conference, 1990, pages 25-35.

[Co82] Cook, M., "Software metrics: an introduction and annotated bibliography", Software Eng. Notes, 7(2), 1982, pages 41-60.

[Co96] Coplien, J., "Column with no name: code patterns", C++ Report, 10, 1996.

[CC92] Coppick, J.C and Cheatham, T.J., "Software metrics for OO systems", Proc. of CSC, 1992, pages 317-322.

[CBO+88] Cote, V., Bourque, P., Oligny, S. and Rivard, N., "Software metrics: an overview of recent results", Journal of Systems and Software, 8, 1988, pages121-131.

[De79] DeMarco, T., Structured analysis and system specification, Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

[Do96] Dodani, M., Chairperson for the OOPSLA'96 Educators' Symposium, San Jose, California, 1996.

[Ds92] D'Souza, D., "Teacher! teacher!", Journal of Object-Oriented Programming, 5(2), 1992, pages 12-17.

[DU79] Dunn, R.H. and Ullman, R.S., "Modularity is NOT a matter of size", Proc. on the Annual Reliability and Maintainability Symposium, 1979, pages 342-345.

[DB93] Durnota, B. and Mingins, C., "Measuring implementation reuse in inheritance hierarchies", 1993.

[DB92] Durnota, B. and Mingins, C., "Tree-based coherence metrics in object-oriented design", TOOLS Pacific, 1992.

[EGY97] Eden, A.H., Gil, J. and Yehudai, A., "Automating the application of design patterns", Journal of Object-Oriented Programming, 10(2), May 1997, pages 44-46.

[EKS94] Eder, J., Kappel, G. and Schrefl, M., "Coupling and cohesion in object-oriented systems", Technical Report, 1994.

[Em84] Emerson, T.J., "A discriminant metric for module cohesion", In Proceedings of the seventh international conference on software engineering, 1984, pages 294- 303.

[Fe90] Fenton, N., "Deriving structurally based software measures", Journal of Systems and Software, 12, 1990, pages 177-187.

[Fe91] Fenton, N.E., Software Metrics: A Rigorous Approach, 1991.

[Fi89] Fiedler, S.P. "Object-oriented unit testing", Hewlett-Packard Journal, 1989, pages 69-74.

[Fi84] Finkelstein, L., "A review of the fundamental concepts of measurement", Measurement, 2(1), 1984, pages 25-34.

[Fi95] Firesmith, D., "Inheritance guidelines", Journal of object-oriented programming", May 1995, pages 67-72.

[Fr97] Fraser, M.C., "POODLE: Programming Object-Oriented Design Language Environment", Internal Report, Communications Research Group, University of Nottingham, UK, 1997.

[GHJ+94] Gamma, E., Helm, R., Johnson, R. And Vlissides, J., Design patterns, Addison-Wesley, 1994.

[Ga97] Gardner, T., "Inheritance: the concepts behind the construct", BCS-OOPS Newsletter, 29, Winter 1997, pages 4-7.

[GM96] Gehringer, E.F. and Manns, M.L., "OOA/OOD/OOP: What programmers and managers believe we should teach", Journal of Object-Oriented Programming, 9(6), October 1996, pages 52-60.

[GH97a] Gibbon, C.A. and Higgins, C.A., "An informal approach to software design evaluation", INSPIRE'97, BCS SIGS, IVF, Sweden, Gothenburg, 18-20 August, 1997.

[GH96a] Gibbon, C.A. and Higgins, C.A., "Teaching object-oriented design with heuristics", SIGPlan Notices, 31(7), July 1996, pages 12-16.

[GH97b] Gibbon, C.A. and Higgins, C.A., "The case for design heuristics in OO curricula", Educators' Symposium: OOPSLA'97.

[GH96b] Gibbon, C.A. and Higgins, C.A., "Towards a learner-centred approach to testing object-oriented designs", Proc. of the APSEC'96, Seoul, South Korea, December 4-7, 1996.

[Gi96] Gibbon, C.A., "Evaluating and visualising object-oriented designs", Doctoral Symposium, OOPSLA'96, San Jose, California, 1996.

[Gi97a] Gibbon, C.A., "Modelling system dynamics with classes, Technical Report", NOTTCS-TR-97-2, Department of Computer Science, University of Nottingham, UK, 1997.

[Gi97b] Gibbon, C.A., "Object-oriented design heuristics: a working document", Internal Report, Department of Computer Science, University of Nottingham, UK, 1997.

[GLH96] Gibbon, C.A., Lovegrove, G.L. and Higgins, C.A., "Tools, heuristics and techniques to assist OO education", Educators' Symposium: OOPSLA'96, San Jose, California, October 1996.

[Gi97c] Gibbon, C.A., On-line Reference to TOAD research material and heuristics, http:/www.cs.nott.ac.uk/~cag/phd/toad.html.

[Gi88] Gilb, T., Principles of software engineering management, Addison-Wesley, Wokingham, UK, 1988.

[Gi77] Gilb, T., Software metrics, Cambridge, MA: Winthrop, 1977.

[Gl87] Glass, R.L., "Standards and enforcers: do they really help achieve software quality", Jour. of Systems Software, 7, 1987, pages 87-88.

[Go97] Goldberg, A., "LearningWorks", Educators' Symposium, OOPSLA'97, Atlanta Georgia, keynote address, 1997.

[Ha96] Hagsand, O., "Interactive Multiuser virtual environments in the DIVE System", IEEE Multimedia, Spring 1996, Vol. 3(1), IEEE Computer Society, ISSN 1070-986X, page 30-39.

[Ha77] Halstead, M.H., Operating and programming systems: elements of software science, New York: Elsevier, 1977.

[HF82] Hamer, P. and Frewin G., "Halstead's software science - a critical examination", Proc. of the 6th International Conference on Software Engineering, 1982, pages 197-206.

[Dh94] Dhama, H., "Quantitative models of cohesion and coupling software", In Annual Oregon Workshop on Software Metrics, 10-12 April 1994.

[HC87] Harrison, W. and Cook, C., "A micro/macro measure of software complexity", Journal of Sys. and Software, 7, 1987, pages 213-219.

[HMK+82] Harrison, W., Magel, K., Kluczny, R. and DeKock, A., "Applying software complexity metrics to program maintenance", Computer, 15(9), September 1982, pages 65-79.

[HE94] Henderson-Sellers, B. and Edwards, J.M., "MOSES: A second generation object-oriented methodology", Object Magazine, 4(3), June 1994, pages 68-71.

[He96] Henderson-Sellers, B., Object-Oriented Metrics, Sydney, Australia: Prentice Hall, 1996.

[HTM94] Henderson-Sellers, B., Tegarden, D. and Monarchi, D., "Tutorial notes: metrics for object-oriented software", OOPSLA'94.

[HK81] Henry, S. and Kafura, D., "Software structure metrics based on information flow", IEEE Transactions on Software Engineering, 7(5), September 1981, pages 510- 518.

[HW93] Henry, S. And Li, W., "Maintenance metrics for the object oriented paradigm", In First International Software Metrics Symposium. IEEE Computer Science Press, 1993.

[HS90] Henry, S. And Selig, C., "Predicting Source-Code Complexity at the Design Stage", IEEE Software, March 1990, pages 36-43.

[HDH81] Henry, S., Kafura, D. and Harris, K., "On the relationship among three software metrics", Performance Evaluation Review, 10(1), Spring 1981, pages 81-88.

[He79] Henry, S.M., "Information flow metrics for the evaluation of operating systems' structure", PhD dissertation, Iowa State University, Ames, IA, 1979.

[Ho94] Hopkins, T.P., "Complexity metrics for quality assessment of object-oriented design", Software Quality Management II, Vol. 2 Building Quality into Software, 2, Computational Mechanics Press, 1994, pages 467-481.

[HB85] Hutchens, D.H. and Basili, V.R., "System structure analysis: clustering with data bindings", IEEE Transactions on Software Engineering, 11(8), August 1985, pages 749-757.

[In91] Ince, D., Chapter in Software metrics: a rigorous approach by Fenton, N., Chapman-Hall, 1991.

[Ja92] Jacobson, I. et al, Object oriented software engineering: A Use Case driven approach, Addison Wesley, 1992.

[JF88] Johnson, R. and Foote, B., "Designing reusable classes", Journal of Object-Oriented Programming", 1(2), 1988, pages 22-35.

[Jo94] Jones, C., "Estimating and measuring object-oriented software", Software Productivity Research, Inc., 1994.

[Jo88] Jones, T.C., A short history of functions points and feature points, ACI Computer Services, 1988, 45 pages.

[KST+86] Kearney, J.K., Sedlmeyer, R.L., Thompson, W.B., Gray, M.A. and Adler, M.A., "Software Complexity Measurement", CACM, 29(11), November 1986, pages 1044-1150.

[Ke87] Kendall, P.A., Introduction to system analysis and design: a structured approach, second edition, WCB publishers, 1987.

[KP74] Kernighan, B.W. and Plauger, P.J., The elements of programming style, Bell Telephone Laboratories, N.J., 1974.

[Ko93] Kolewe, R., "Metrics in object oriented design and programming", Software Development, Oct 1994, pages 53-62.

[LKS+81] Lassez, J-L., van der Knijff, D., Shepherd, J. and Lassez, C., "A critical examination of software science", Journal of Systems and Software, 2, 1981, pages 105-112.

[Le80] Lehman, M., "Programs, life cycles, and laws of software evolution", Proc. of the IEEE 68(9), 1980.

[LMB90] Levine, J.R., Mason, T. and Brown, D., Lex and yacc, O'Reilly and Associates, Inc., 1990.

[Le86] Levitin, A.V., "How to measure software size and how not to", In Proc. of IEEE Compsac'86, 1986, pages 314-318.

[Le87] Levitin, A.V., "Investigating predictability of program size", in Proceedings of IEEE COMPSAC, 1987, pages 231-235.

[LC87] Li, H.F. and Cheung, W.K., "An empirical study of software metrics", IEEE Transactions on Software Engineering, 13(6), June 1987, pages 697-708.

[LR89] Lieberherr, K.J. and Riel, A.I., "Contributions to teaching object-oriented design and programming", In OOPSA'89, pages 1-6.

[LBS91] Lieberherr, K.J., Bergstein P. and Silva-Lepe, I., "From object to classes: algorithms for optimal object-oriented design", Journal of Software Engineering, 6(4), 1991, pages 205-228.

[Li97] Lilley, J., On-line reference, John Lilley's C++ Grammar, http://www.empathy.com/pccts/index.html.

[Li88] Liskov, B., "Data Abstraction and Hierarchy", SIGPlan Notices, 23(5), May, 1988.

[Li82] Lister, A., "Software science - the emperor's new clothes?", Australian Computer Journal, May, 1982, pages 66-70.

[LK94] Lorenz, M. and Kidd, J., Object oriented software metrics. Prentice Hall, 1994.

[Lo90] Loy, P.H., "A comparison of object-oriented and structured development methodologies", ACM SIGSoft Software Engineering Notes, 15(1), January 1990, pages 44-48.

[Ma95] Mancl, D., Panel on: tailoring OO analysis and design methods, OOPS Messenger, 6(4), October 1995.

[Ma96] Martin, R., "The interface segregation principle: one of many principles of OOD", The C++ Report, 1996.

[MB89] McCabe, T.J. and Butler, C.W., "Design complexity measurement and testing", Communication of the ACM, 32(12), December 1989, pages 1415-1425.

[Mc76] McCabe, T.J., "A complexity measure", IEEE Transactions on Software Engineering, 2(4), December 1976, pages 308-320.

[Mc93] McKim, J.C., "Teaching object-oriented programming and design", Journal of Object-Oriented Programming, 6(1), March/April 1993, pages 32-39.

[Me87] Meyer, B., "Reusability: The case for object-oriented design", IEEE Software, March 1987, pages 50-64.

[Me88] Meyer, B., Object-oriented software construction, Prentice-Hall, 1988.

[Me96] Meyers, G., Private Email Communication.

[Me95] Meyers, S., More effective C++ : 35 new ways to improve your programs and designs, Addison-Wesley, 1995.

[MDS93] Mingins, C., Durnota, B. And Smith, G., "Collecting software metrics data for the Eiffel class hierarchy", In TOOLS Proceedings, 1993, pages 427-435.

[MD89] Moreau, D.R. and Dominick, W.D., "Object-oriented graphical information systems: research plan and evaluation metrics", Journal of Systems and Software, 10, 1989, pages 23-28.

[My77] Myers, G.J., "An extension to the cyclomatic measure of program complexity", ACM SIGPLAN Notices, 12(10), 1977, pages 61-64.

[My76] Myers, G.J., Software reliability: principle and practice, John Wiley & Sons, New York, 1976, pages 88-109.

[NB95] Narayanaswamy, K. and Blakey, A., "Workshop Report On: Are Object-Oriented CASE Frameworks ready for Prime Time?", OOPS Messenger, 6(4), October 1995, pages 155-158.

[PCR92] Patel, S., Chu, W. and Baxter, R., "A measure for composite module cohesion", In 15th International Conference on Software Engineering, ACM, 1992, pages 28-48.

[Pu97] Purdue Compiler Construction Tool Sets, On-line reference, http://www.ocnus.com/pccts.html, 1997.

[RL92] Rajaraman, C. and Lyu, M.R., "Reliability and maintainability related software coupling metrics in C++", In IEEE Proceedings of the 3rd International Symposium on Software Reliability, 1992, pages 303-311.

[Re84] Reynolds, R.G., "Metrics to measure the complexity of partial programs", Journal of Systems and Software, 4, 1984, pages 75-91.

[Ri96] Riel, A., Object-oriented design heuristics, Addison-Wesley, 1996.

[RB89] Robillard, P. and Bolouix, B., "The interconnectivity metrics: A new metric showing how a program is organised", Journal of Systems and Software, 10, 1989, pages 29-39.

[Ro88] Roskind, J., On-line reference for LALR(1) C++ grammar, http://www.empathy.com/pccts/roskind.html.

[RH68] Rubey, R.J. and Hartwick, R.D., "Quantitative measurement of program quality", In Proceedings of the 23rd National Conference, ACM Publications August, 1968, pages 671-677.

[Ru91] Rumbaugh, J. et al, Object-Oriented Modelling and Design, Prentice-Hall, 1991.

[SK96] Schank, R.C. and Kass, A., "A goal-based scenario for high school students", Communication of the ACM, 39(4), 1996, pages 28-29.

[SH79] Schneidewind, N.F. and Hoffmann, H., "An experiment in software error data collection and analysis", IEEE Transactions on Software Engineering, 5, January 1979, pages 276-286.

[Se89] Selby, R., "Quantitative studies of software reuse", In Biggerstaff, T. and Perlis, A.J., Software Reusability Vol. II Applications and Experiences, Addison-Wesley, 1989, pages 213-233.

[SCD83] Shen, V.Y., Conte, S.D. and Dunsmore, H., "Software science revisited: A critical analysis of the theory and its empirical support", IEEE Transactions on Software Engineering, 9(2), March 1983, pages 155-165.

[Sh88] Shepperd, M., "An evaluation of software product metrics", Journal of Information and Software, 30(3), 1988, pages 177-188.

[SGB83] Silverman, J., Giddings, N. and Beane, J., "An approach for design for maintenance", In Proceedings of Software Maintenance Workshop, 1983.

[SD97] Snoeck, M. and Dedene, G., "Object behaviour and the IS-A relationship", Birds of a feature session, Object Technology conference, Oxford, 1997.

[Sn89] Snyder, A., "Encapsulation and Inheritance in Object Oriented Programming Languages", In OOPSLA Sigplan Notices, 1986, pages 38-45.

[SP96] Soloway, E. And Pryor, A., "Log on education: The next generation in human-computer interaction", Communication of the ACM, 39(4), April 1996, pages 16-18.

[SMC74] Stevens, W.P., Myers, G.J., and Constantine, L.L., "Structured Design", IBM System Journal, 4(2), 1994, pages, 115-139.

[St81] Stevens, W.P., Using structured design: how to make programs simple, changeable, flexible and reusable, John Wiley and Sons: A Wiley-Inter-Science Publication.

[SHR95] Stiglic, B., Hericko, M. and Rozman, I., "How to evaluate object-oriented software development", ACM SIGPLAN Notices, 30(5), May 1995, pages 3-10.

[St95] Stroustrup, B., "Why C++ is not just an OO programming language", OOPS Messenger, 6(4), 1995, pages 1-13.

[St92] Stump, Laine. Writing reusable classes: Five top tips on C++ class design. EXE: The Software Developer's Magazine, 6(9), 1992, pages 59-61.

[Ta93] Taivalsaari, A., "Object-oriented programming with modes", Journal of object-oriented programming, 6(3), 1993, pages 25-32.

[TSM92] Tegarden, D.P., Sheetz, S.D. and Monarchi, D.E., "Effectiveness of traditional software metrics for object-oriented systems", In Proceedings HICSS-92, IEEE, 1992.

[TZ81] Troy, D.A. and Zweben, S.H., "Measuring the quality of structured designs", Journal of Systems and Software, 2, 1981, pages 113-120.

[VJT92] Vessey, I., Jarvenpaa, S.L. and Tractinsky, N., "Evaluation of vendor products: Case tools as methodology companions", Communication of the ACM, 35(4), April 1992, pages 90-105.

[Wa79] Walsh, T.J., "Software reliability study using a complexity measure", In Proceedings of the National Computer Conference, New York: AFIPS, 1979.

[WW94] Wang, B.L. and Wang, J., "Is a deep class hierarchy considered harmful?", Object Magazine, 4(7), Nov-Dec 1994, pages 35-36.

[WH93] Wei, L. And Henry, S., "Object-oriented metrics for predicating maintainability", Journal of Systems and Software, 23(2), November, 1993, pages 111-122.

[We92] Wei, L., "Applying software maintaining metrics in the software development life cycle", Master's Thesis, MIT, 1992.

[We70] Weinberg, G.M., PL/I Programming: A Manual of Style, McGraw-Hill, New York, 1970.

[We74] Weissman, L., "Psychological complexity of computer programs: An experimental methodology", ACM SIGPLAN Notices, 9(6), June 1974, pages 25-36.

[We96] West, M., "Taking the measure of metrics", Object Expert, 1(2), Jan-Feb, 1996, pages 43-45.

[We88] Weyuker, E.J., "Evaluating software complexity measures", IEEE Trans. on Software Engineering, 14(9), September 1988.

[Wh92] Whitmire, S., "Measuring the complexity of object-oriented software", In Third international conference on the application of software measures, La Jolla, California, 1992.

[Wh96] Whitty, R., "Object-oriented metrics: A status report", Object Expert, 1(2), Jan.-Feb. 1996, pages 35-40.

[Wi96] Wild, F., "Instantiating code patterns", In Dr. Dobb's Journal, June 1996.

[Wi90] Wirfs-Brock et al., Designing object-oriented systems, Prentice-Hall, 1990.

[WW89] Wirfs-Brock, A. and Wilkerson, B., "Variables Limit Reusability", Journal of Object-Oriented Programming, 2(1), May/June 1989, pages 34-40.

[WJ90] Wirfs-Brock, R. and Johnson, R.E., "Surveying current research in object-oriented design", Communication of the ACM, 33(9), September 1990, pages 105-124.

[WSD81] Woodfield, S., Shen, V. And Dunsmore, H., "A study of several metrics for programming effort", Journal of Systems and Software, 2, June 1981, pages 97-103.

[WHH79] Woodward, M., Hennel, M. and Hedley, D., "A measure of control flow complexity in program TEXT", IEEE Transactions on Software Engineering, 5, January 1979, pages 45-50.

[YC85] Yau, S.S. and Collofello, J.S., "Design stability measures for software maintenance", IEEE Transactions on Software Engineering, 11(9), September 1985, pages 849-856.

[YW78] Yin, B.H. and Winchester, J.W., "The establish and use of measure to evaluate the quality of software design", In Proceedings of ACM Software Quality Assurance Workshop, volume 3, 1978, pages 45-52.

[YC79] Yourdon, E. and Constantine, L.L., Structured design, Englewood Cliffs, NJ, Prentice-Hall, 1979.

[Zu92] Zuse, H. "Properties of software measures", Software Quality Journal, 1, 1992, pages 225-260.

[Zu96] Zuse, H., "The History of Software Measurement", On-line reference: http://irb.cs.tu-berlin.de/zuse/3-hist.html, 1996.

Glossary

abstract design model	a model of the static, class-based properties of an object-oriented design.
actor	an entity that uses the services of other entities but never provides any services of its own.
afferent stability	an indication of the degree to which a system can incur changes upon a single entity (e.g. class/cluster).
agent	an entity that uses the services of other entities and exports services of its own.
aggregate	an object composed of one or more other objects [Bo94].
aggregated	objects that compose an aggregate object [Bo94].
API	application programming interface.
application framework	a collection of classes that provide a set of services for a particular domain; these classes define an off the shelf, customisable, working system.
class description language (CDL)	an ASCII-based language used by TOAD to specify all the elements within the *abstract design model* .
class role migration	a model of maintainability that articulates the static properties of an object-oriented design in dynamic manner by extrapolating the inherent client/server relationships that exist between classes (Appendix E).
classical metric	a metric that generates a large amount of publications that serve to refute or substantiate its ability to measure software.
code metric	metrics that are enacted upon program source.
cohesion	the strength of dependencies within an entity.
control flow metric	a metric that assesses the control path through a program.

coupling the strength of dependencies between entities.

design evaluation the means by which a user articulates a design product to gain a deeper understanding of both its good and bad features. The user not only learns from the discovered problems but equally from the observed solutions presented within the context of their own design.

design heuristic a small, simple and legible piece of design expertise that delivers experience from the expert to the novice in the most effective manner.

design pattern a generic solution to a recurring design problem.

efferent stability the potential for changes to a system due to amendments made to a single entity (e.g. class/cluster).

external product attribute an observable quality in software that both management and clients strive to engineer into their application products, such as maintainability, reliability and efficiency.

fan-in fan-in of a module `M` is the number of *local flows* that terminate at `M`, plus the number of data structures from which information is retrieved by `M`.

fan-out fan-out of a module `M` is the number of *local flows* that emanate from `M`, plus the number of data structures that are updated by `M`.

general reuse library a collection of classes that provide abstractions to be reused principally through inheritance.

God class a behaviourally rich class that centralises system intelligence within a design.

hybrid metric a combination of *inter-modular* and *intra-modular* metrics.

implementation inheritance defining one object's representation in terms of another object's representation— code reuse.

inert class a class that is data rich but behaviourally poor, typically these classes act as data sinks or unintelligent, database-oriented structures.

information flow metric a metric that assesses how information is being consumed and/or passed throughout a program by examining the assignment of data to variables.

interface collaborator a type/class/object that is part of the interface of another type/class/object.

interface inheritance the definition of subtypes such that *substitutability* can occur between objects that have inherited interface from ascendent types.

inter-modular the properties/dependencies between a collection of software modules.

internal product attribute something that can be measured purely in terms of the product, e.g. *size*, *modularity*, *coupling*, *cohesion*, etc.

intra-modular the properties within a software module.

local flow the movement of information between modules.

modularity a property of software that gives rise to decomposable modules that are highly cohesive and loosely coupled modules and are also composable in other contexts.

object an instance of a *class*. It has state, behaviour and identity.

quality framework defines the inter-relationships between internal and external product attributes when determining how to measure quality.

role migration see *class role migration* .

server an entity that exports services but never uses the services of other entities.

structure metric a traditional metric that attempts to measure the modular properties within structured programs.

substitutability If for each object `o1` of type `S` there is an object `o2` of type `T` such that for all programs `P` defined in terms of `T`, the behaviour of `P` is unchanged when `o1` is substituted for `o2`, then `S` is a subtype of `T` [Li88].

teaching method a highly pragmatic and informal approach to educating learners in aspects of object-oriented design.

type a set of observable properties and behaviours to which a number of objects conform. A type specifies an interface that an object must provide and the behaviour the object must exhibit.

type extension the creation of a subtype from an existing*type.*

Appendix A

The Class Description Language

The class description language (CDL) documents the static, class-based properties within an object-oriented design. CDL-compliant files adopt an ASCII flat file format that permit them to be ported to any architecture. The class-based structures defined by the CDL are thus:

Class Description Language

[] : denotes optionality	(it is not part of the grammar)
| : denotes exclusive OR	(it is not part of the grammar)
* : denotes zero or more	(it is not part of the grammar)

```
cdlFile {
toadCluster*
class*
}

toadCluster {
clusterName theName
parentCluster theName
class*
}

class {
className theName
method*
base*
attribute*
[ template ] [ abstract ]
[ friend theClass ]
}
```

```
base {
baseName theName
scope public | protected | private
[ virtual ]
}

method {
[friend ] [ abstract ]
[ virtual | cdlDynamic ]
methodName theName
scope public | protected | private
parameters { attribute* }
}

attribute {
[ idName theName ]
type Name theType
scope public | protected | private
containment [ byValue | byRef ]
}
```

CDL files are composed of clusters and classes. The *inheritance*, *aggregation* and *using* relationships exist between classes and the *using* relationship between clusters.

A grammar is used to describe the contents of the CDL. The syntax for the CDL grammar is specified using the UNIX meta tools `yacc` and `lex`. `yacc` is used to generate a portable C-based parser that reads a stream of tokens presented to it by the lexical analyser tool `lex`. Upon input to the C-based parser, CDL files are used to instantiate objects within the **abstract design model** (see chapter 5).

The abstract design model is a framework of C++ classes that is representative of the design attributes contained within the CDL. Collectively, these classes provide an API to the design attributes within a CDL file. Figure A—1 depicts this framework of classes that define the abstract design model.

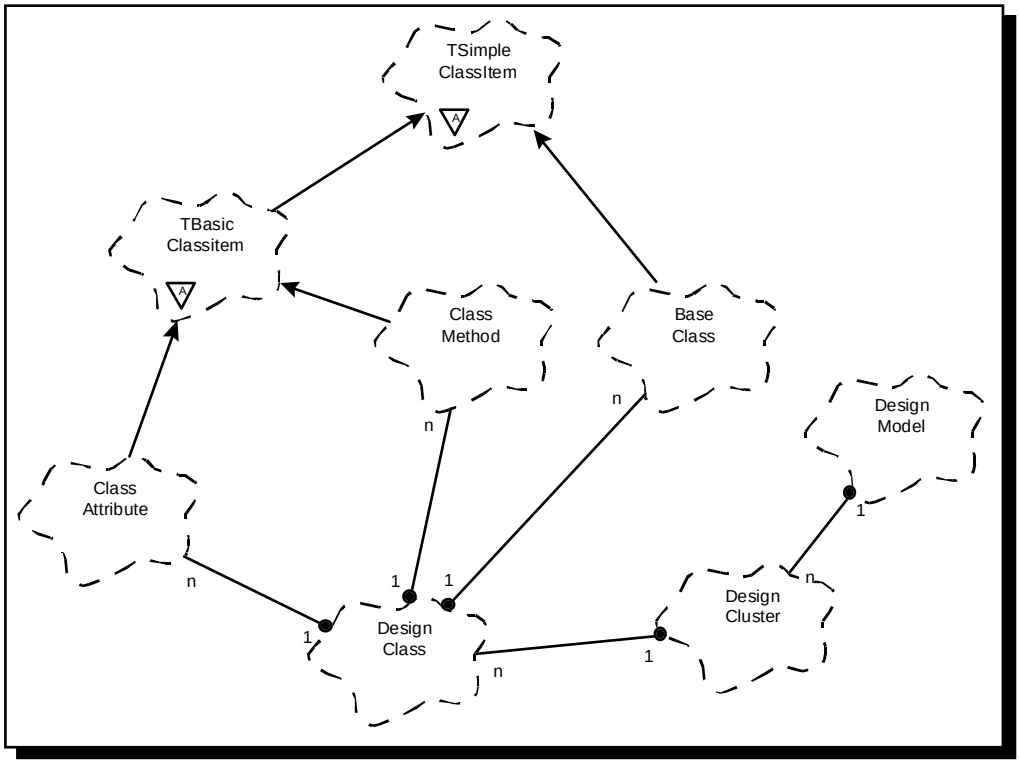


Figure A—1 : Classes for the abstract design model

The abstract class `Parser` in Figure A—2 exports a series of methods, or actions, that are called when the CDL grammar rules are reduced. This permits different types of `Parser` to be created by extending it. `C++DesignParser` and `DebugParser` are two of the more important derivatives of `Parser`; the former being used to instantiate C++ objects in the abstract design model and the latter employed to provide instructive error handling feedback for defective CDL files.

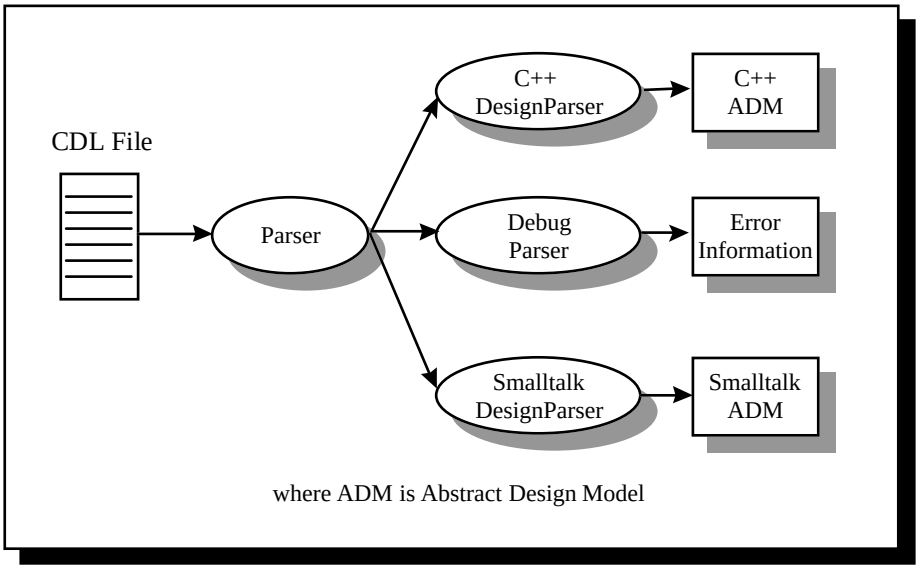


Figure A—2 : The `Parser` Abstract Interface

Parsers can also be implemented that instantiate objects of the abstract design model that belong to different programming languages such as Eiffel, CLOS and Smalltalk. For example, Figure A—2 depicts how a`SmalltalkParser` can be used to instantiate `Smalltalk` objects for a `Smalltalk` abstract design model by extending `Parser` and embedding `Smalltalk` code within its methods. To summarise, `Parser` encapsulates all knowledge of the CDL grammar and exports a well-defined, customisable action interface to its design elements.

CDL yacc-specified grammar

The following yacc-specified implementation of the CDL grammar invokes an embedded C API to initialise structures within the abstract design model. For clarity, the embedded C API calls have been removed to accentuate the rules governing the parsing of CDL files.

```
%token <intval> NUMBER
%token <string> STRING
%token CLASS CLASSNAME TEMPLATE METHOD FRIEND ABSTRACT
%token VIRTUAL METHODNAME SCOPE PUBLIC PROTECTED PRIVATE
%token PARAMETERS ATTRIBUTE CONTAINMENT BYVALUE DYNAMIC
%token BYREF BASENAME TYPENAME BASE IDNAME CLUSTER CLUSTERNAME

%start classFile
%%
classFile           :     /* empty rule */
                    |     classFileContents
                    |     error
                    ;

classFileContents   :     classFileContents classFileContent
                    |     classFileContent
                    ;

classFileContent    :     classCluster
                    |     class
                    ;


classCluster        :     CLUSTER '{'  clusterContents '}'
                    |     CLUSTER '{' '}'
                    ;

clusterContents     :     clusterContents clusterContent
                    |     clusterContent
                    ;
                    ;
clusterContent      :     CLUSTERNAME STRING
                    |     PARENTCLUSTER STRING
                    |     class
                    ;

class               :     CLASS '{' classContents '}'
                    |     CLASS '{' '}'
                    ;

classContents       :     classContents classContent
                    |     classContent
                    ;


classContent        :     attribute
                    |     base
                    |     method
                    |     TEMPLATE
                    |     FRIEND STRING
                    |     className
                    |     ABSTRACT
                    ;

className           :     CLASSNAME STRING
                    ;

scopeLine           :     SCOPE accessSpecifier
                    ;

accessSpecifier     :     PRIVATE
```

```
                    |     PROTECTED
                    |     PUBLIC
                    ;
containmentLine     :     CONTAINMENT containmentSpecifier
                    ;


containmentSpecifier:     BYVALUE
                    |     BYREF
                    ;

attribute           :     ATTRIBUTE '{' attributeContents '}'
                    |     ATTRIBUTE '{'  '}'
                    ;

attributeContents   :     attributeContents attributeContent
                    |     attributeContent
                    ;

attributeContent    :     idName
                    |     scopeLine
                    |     containmentLine
                    |     attrTypeName
                    ;

attrTypeName        :     TYPENAME STRING
                    ;
idName              :     IDNAME STRING
                    ;

attributeList       :     attributeList attribute
                    |     attribute
                    ;

base                :     BASE '{' baseContents '}'
                    |     BASE '{' '}'
                    ;

baseContents        :     baseContents baseContent
                    |     baseContent
                    ;

baseContent         :     scopeLine
                    |     VIRTUAL
                    |     baseName
                    ;

baseName            :     BASENAME STRING
                    ;

method              :     METHOD '{' methodContents '}'
                    |     METHOD '{' '}'
                    ;

methodContents      :     methodContents methodContent
                    |     methodContent
                    ;

methodContent       :     ABSTRACT | VIRTUAL | DYNAMIC | FRIEND
                    |     TEMPLATE
                    |     parameterList
                    |     scopeLine
                    |     methodName
                    ;

methodName          :     METHODNAME STRING
                    ;

parameterList       :     PARAMETERS '{' attributeList '}'
                    |     PARAMETERS '{' '}'
                    ;
%%
```

Appendix B

Applying TOAD

This appendix steps through the process of applying design heuristics to class models within TOAD. The student design in Figure B—1 was selected at random from the submitted 96/97 OBJ coursework and serves as the ongoing example. Although this appendix cannot comprehensively walk through the highly interactive evaluation process for the given design, it does provide a good insight into the way learners articulate their designs with TOAD.

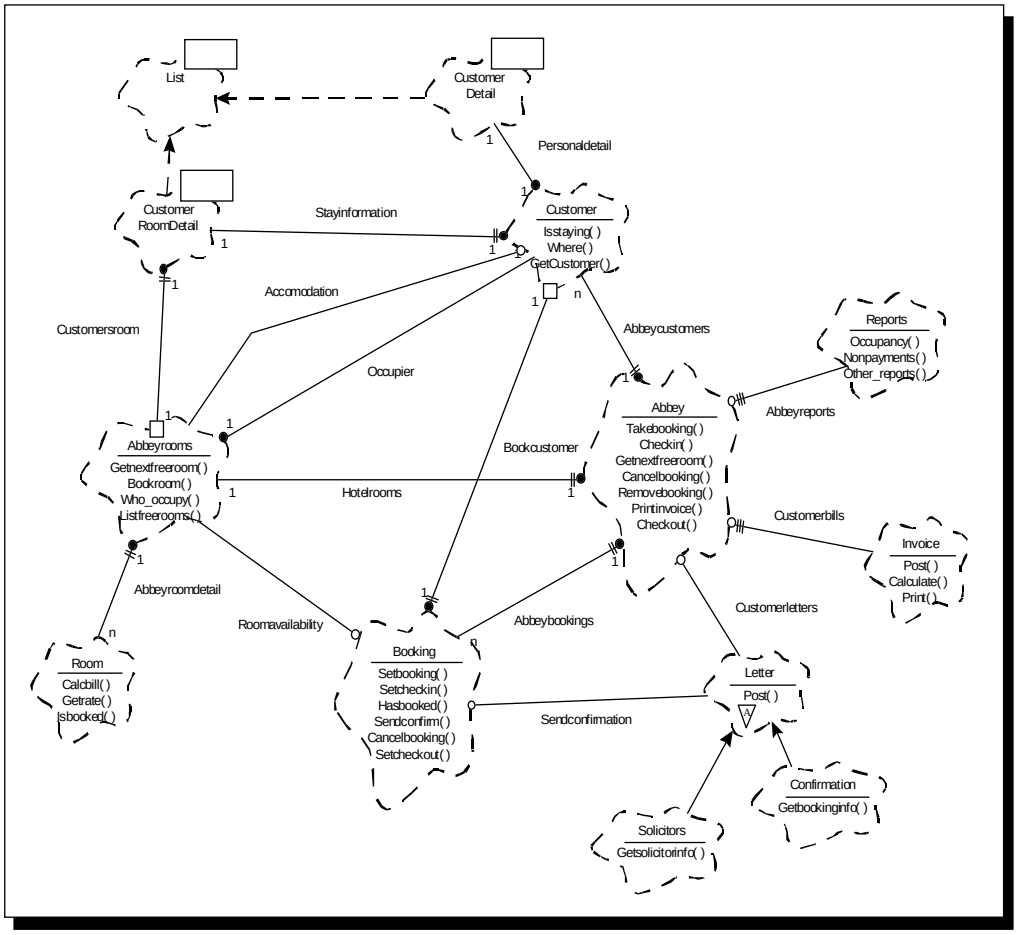


Figure B—1 : The main class diagram for the Abbey Hotel System

B.1 A student design

The student design implements the Abbey Hotel Management System (see Table 6—1). Figure B—2 displays the TOAD start-up screen which has loaded the Abbey design in preparation for its informal evaluation by heuristics. The Abbey design, represented by the `abbey.oodl` class description file, contains 22 classes and will be evaluated as a complete working system indicated by the `System` label in the bottom right hand corner of the TOAD screen.

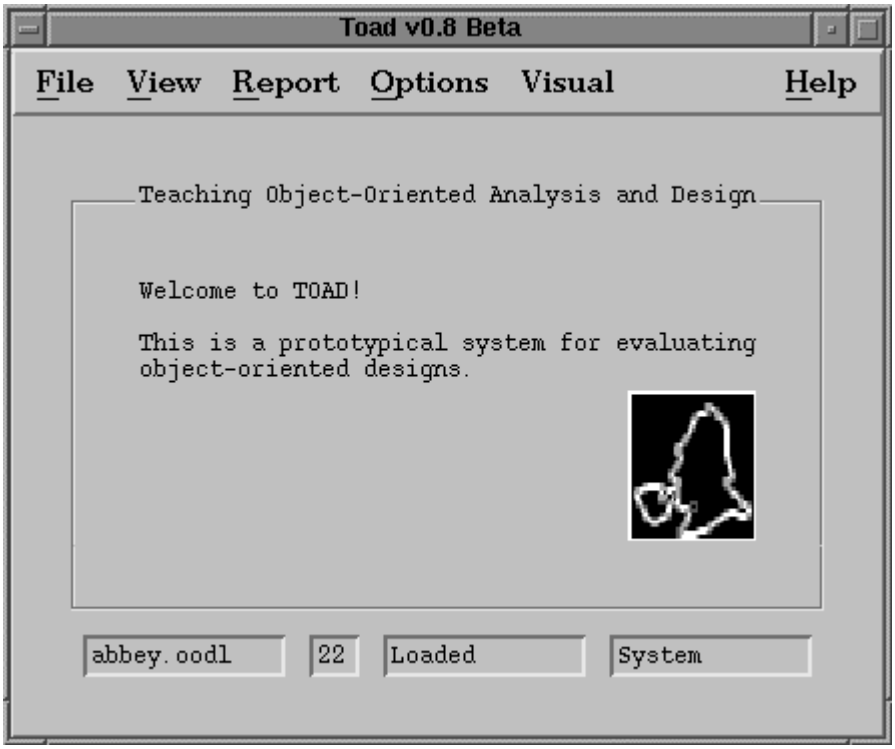


Figure B—2 : The TOAD start-up screen

B.2 Design reports

Figure B—3 displays the main evaluation area within TOAD. From here the user has access to an array of reporting tools and those heuristics that have been breached within the design. To determine which classes breach a heuristic `H`, TOAD applies the *problem detection* algorithms of `H` (see section 4.6.1) to every class within the Abbey design. The `Design Classes`

window in Figure B—3 lists all classes in the Abbey design for which AARDVARK is the current selection. The `Breached Heuristics` window lists those heuristics that were breached for the currently selected class.

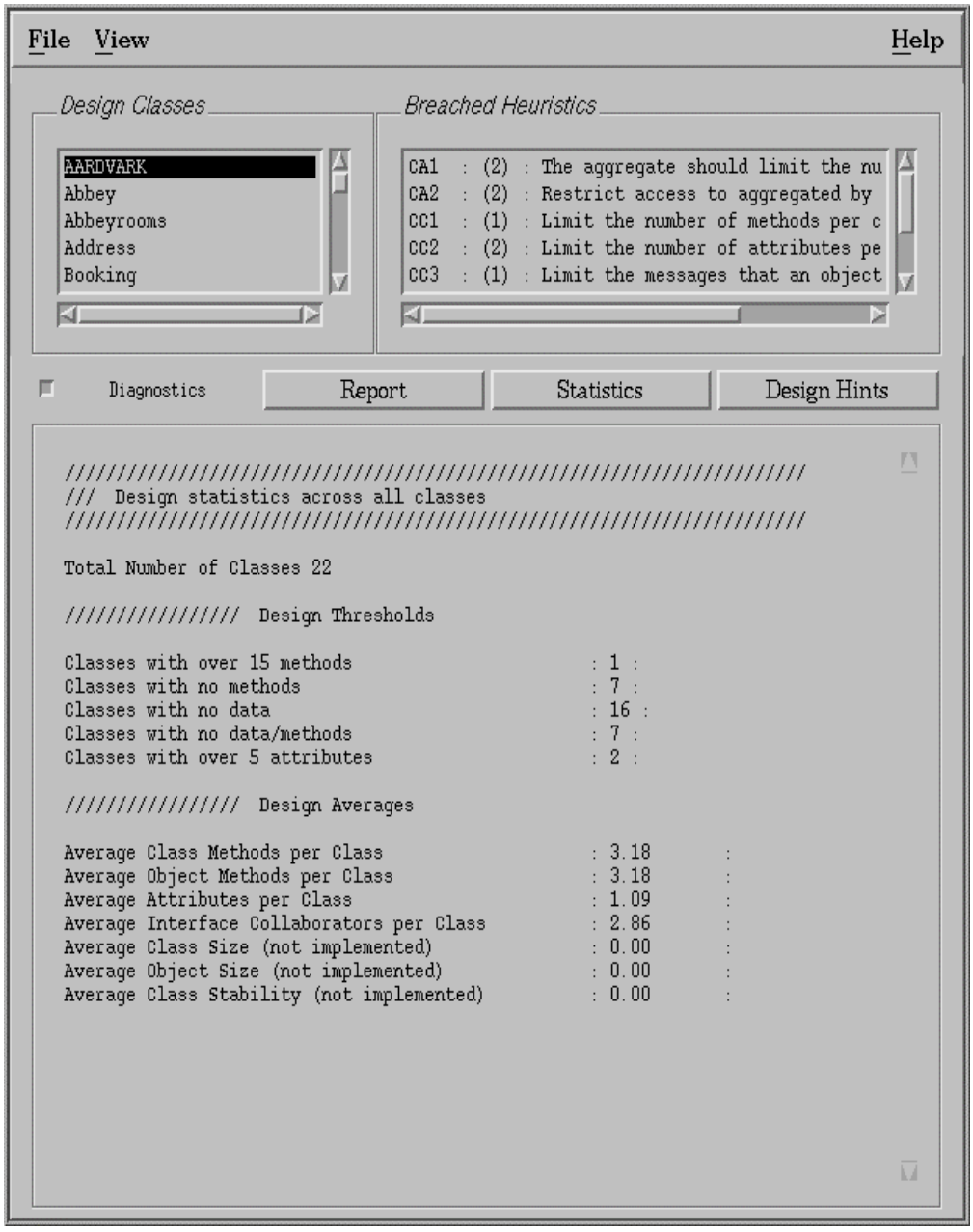


Figure B—3 : Class Browser within TOAD

AARDVARK is a reserved word in TOAD that is used to represent*all* classes in a design. AARDVARK is a special case that displays every heuristicH that was breached in the entire design where the figure in brackets records the

number of classes that `H` was breached in (see Figure B—3). In the main text area the statistical details suggests that the majority of the design's behaviour resides within a small number of classes — indicated by 7/22 classes containing no methods *and* no data. These empty classes are typically simple types provided for in the target programming language such as `String` and `Boolean`. The statistics also report that only one class has a large number of methods. This class was `Abbey`. Upon further inspection, `Abbey` breaches heuristics `CU1` and `CU2` suggesting that it is a good candidate for being a God class (see section 6.2.2.1).

The learner can toggle between `Diagnostics` and `Solution` modes during design evaluation. Figure B—3 depicts TOAD in `Diagnostics` mode. In this mode TOAD will present *problem feedback* on any of the breached heuristics. Upon moving into `Solution` mode TOAD will supply the learner with a *generic solution* that details a well-documented approach to rectifying the problem identified by the breached heuristic.

Figure B—4 depicts `Abbey` as the current selection and lists those heuristics that it breaches. From this list of breached heuristics we note that `Abbey` has a large number of methods (`CC1`, `CC3`) and collaborates with a number of system classes (`CU1`, `CU2`). In `Diagnostics` mode the learner can select one of these heuristics to receive its problem feedback. This will present them with the heuristic's rationale, exactly where in the design the problem occurs and the reasoning behind *why* the heuristic was breached. Moving into `Solution` mode presents the student with a number of generic design alternatives. In this case, the generic solutions describe how to fragment tightly coupled classes (`CU1`), how to *demote* public collaborators (`CU2`) and suggestions for minimising the bandwidth of large objects (`CC1`, `CC3`).

A closer look at the interface-specific definition of `Abbey` in Figure B—4 provides additional clues as to why these heuristics have been breached. It demonstrates that the centralised nature of `Abbey` is a result of the student

using it to provide a complete interface to the *entire* hotel management system. More specifically, the student has augmented system functionality with the user interface and implemented it as a single class:`Abbey`. A means to separate the underlying system from the user interface will enhance the maintainability of the design.

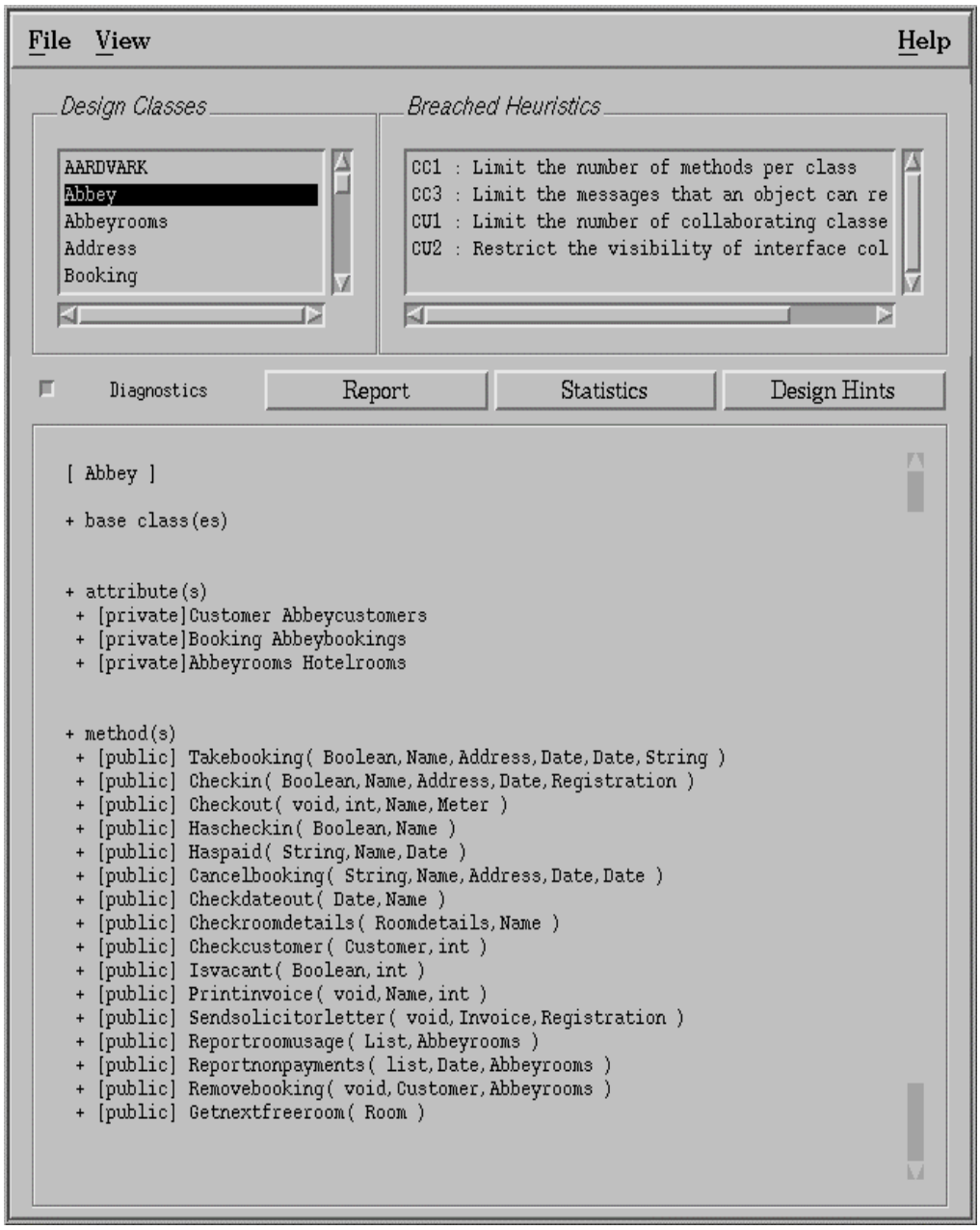


Figure B—4 : Pseudo design definition of Abbey

It is important to note the role TOAD played in the evaluation of `Abbey`. Heuristics objectively identified that there was a problem with `Abbey`, pinpointed which classes were involved and presented a number of possible solutions to incrementally enhance the design. However, the student was still required to interact with TOAD, supplying subjective input to fully assimilate the problem *before* applying its suggested solutions. Often, solutions are qualitative and fall outside the remit of TOAD. Nevertheless, TOAD provides the means to quickly assimilate a given design context and reason about the contributing problems. In this case, the suggested solutions still need to be applied but now the designer is more informed about of the reason(s) why the problem occurred.

The report for `AARDVARK` illustrated in Figure B—5 summarises a number of common design problems found within the Abbey design. In particular, 8 classes were found to have methods yet possesses no data; *dataless* classes. Objectively, TOAD cannot advise the user on how to improve dataless classes. However, TOAD can describe certain circumstances in which dataless classes are encouraged (e.g. disseminating notions of type through an inheritance hierarchy using abstract classes) and situations in which they are detrimental to the system (e.g. the implementation of most concrete classes). In fact, TOAD looks out for these exceptional cases and offers the appropriate feedback.

The design report also highlights those classes that have **high efferent** and/or **high afferent coupling** The high afferent coupling on `Boolean`, `Customer` and `Abbeyrooms` indicates that they export their services to a large number of classes in the system. Such classes need to be well-encapsulated to ensure that changes made to them do not propagate throughout the system. A class that collaborates with a large number of classes to fulfil its own responsibilities is subject to high efferent coupling. Typically, these (God) classes are centralising behaviour in the system. The `Abbey` class has high efferent coupling.

Design reports are purposefully brief presenting the designer with a concise overview of the problematic classes within a design. The next step for the designer would be to consult the individual reports for these classes to gain a deeper understanding of their strengths and/or weaknesses.

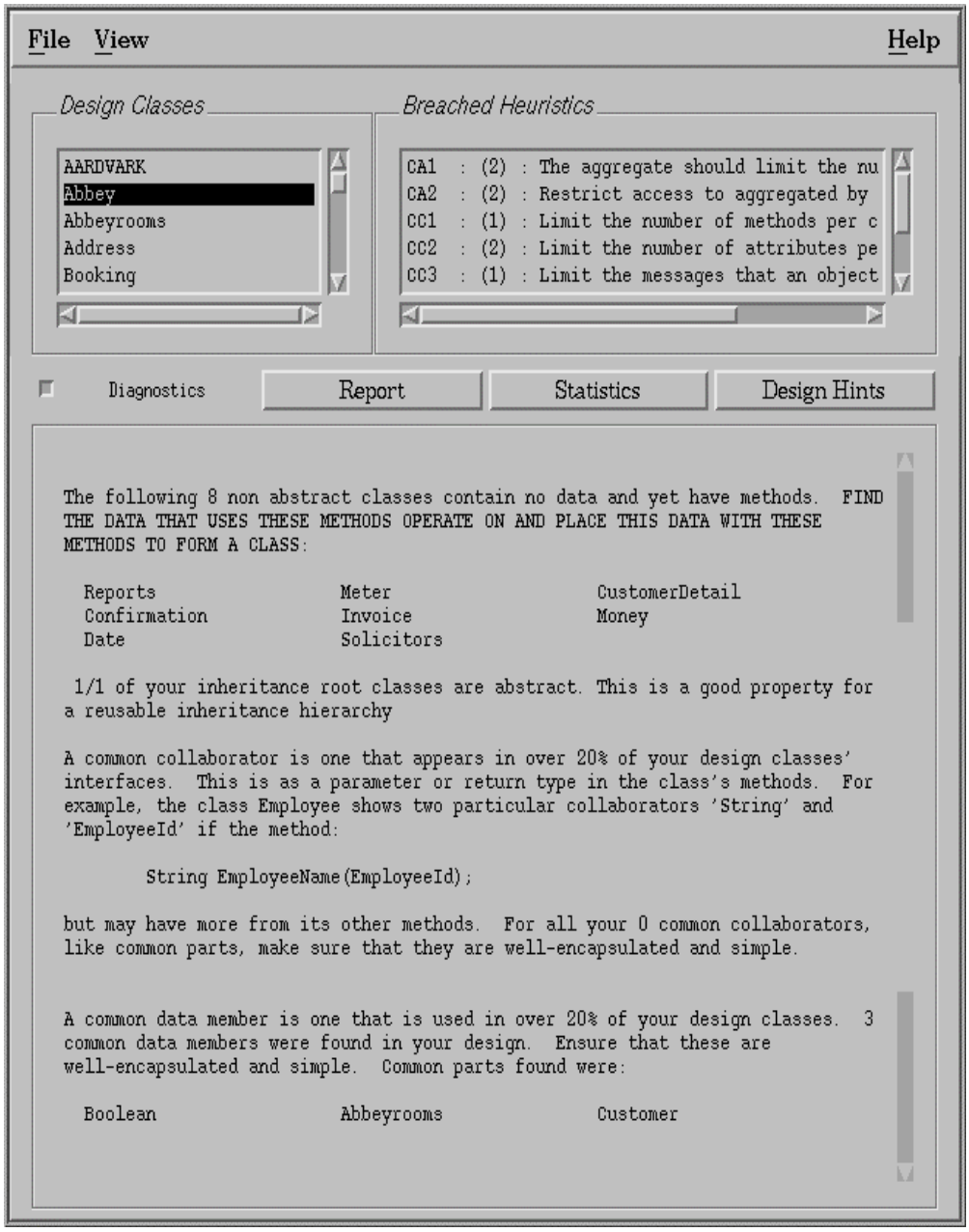


Figure B—5 : Design report for the Abbey Hotel System

As described in chapter 4, a class report is divided into a number of concept categories that were listed in Table 4—3. A concept category focuses upon

important design issues that are described in terms of heuristics. These categories are a direct link between paradigmatic concepts such as encapsulation and inheritance and the design heuristics that support them. From the encapsulation category in Figure B—6 the learner is referred to the breached heuristics `CC6` and `CA2`. The learner can then look up heuristics `CC6` and `CA2` to get diagnostic feedback regarding the identified design problem. On this occasion, the heuristics suggest that to better encapsulate `Abbeyrooms` they should make `Customer` private and provide a public accessor method to it.

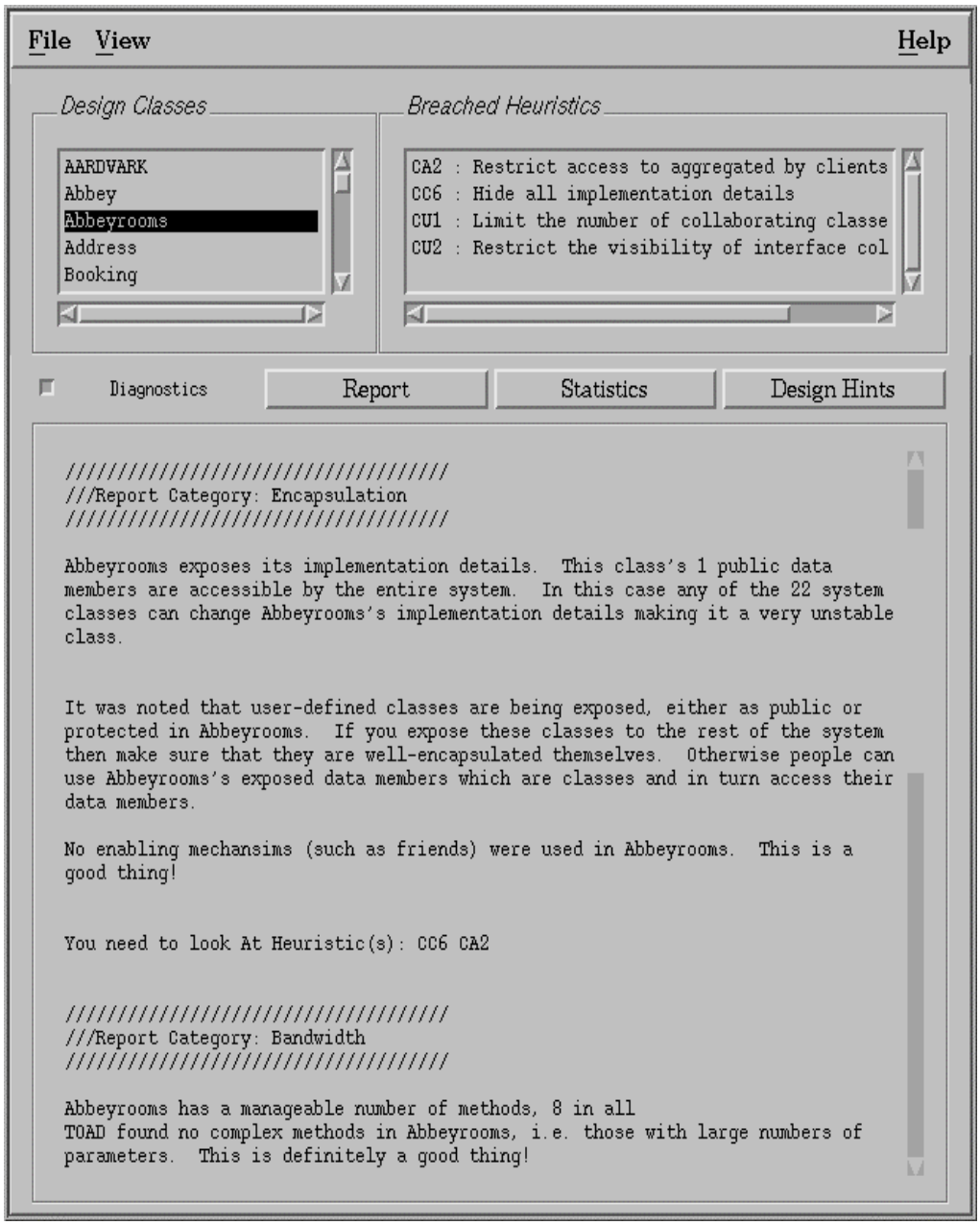


Figure B—6 : Class report for `Abbeyrooms`

TOAD reports provide quick access to common design problems. Design reports summarise the system under evaluation highlighting its strengths and weaknesses in terms of problematic classes. From here the designer uses class reports to gain a more in-depth understanding of a particular class's problems. Class reports organise their feedback around concept categories that ultimately direct the designer to the appropriate design heuristics.

Both design and class reports are general and tend to target the inexperienced. As designers become more proficient in their use of object technology, their use of TOAD reports will no longer be a source of learning but a more direct vehicle for getting feedback in terms of design heuristics. In the next section we report upon what the original designer thought the problems were with their design and whether they used heuristics to fix them.

B.3 What the student thought

TOAD provides numerous facilities for learners to comprehensively reason about the problems it locates in their software designs. This section outlines some of the difficulties that arose during design evaluation and how the student set about overcoming them. The example design used throughout this appendix was submitted by the student prior to evaluation by TOAD. `Design A` refers to this design and `Design B` to the design subsequent to evaluation *and* the completion of a software maintenance task. `Design B` has an additional 3 classes and supports the extra functionality that the maintenance task on `Design A` required.

This student and those on the OBJ course first applied the advice given by TOAD that resulted in the least amount of re-design. It was also noted that the students learnt from those design problems that they had encountered directly. For this student, the problems that arose in `Design A` were not *re-introduced* into `Design B`. For example, upon realising *why* classes with

large implementations posed problems in `Design A`, the student did not introduce new classes of this nature into `Design B`. In fact, they re-designed some of these problem classes to remove them completely from the system. Nevertheless, those problems that did not occur in `Design A` were often introduced in `Design B`. This re-enforced our original observation that learners learn best by example and the best examples tend to be those that are presented in the context of their own work.

On a number of occasions the student reported that they chose to ignore the advice given by TOAD because they found it inappropriate or too costly to enforce. For example, the heuristic for high object bandwidth was breached only once in `Design A` but 3 times in `Design B`. For the `Abbey` class, the student was aware that they were centralising system behaviour. They also knew how to set about resolving it but did not have enough time to do this *and* the maintenance task. As such, `Abbey` remained a God class. For the remaining two large classes in their design (`Customer`, `Booking`) the student argued that these classes were key to the design and should therefore be rich in behaviour. Unfortunately, the student did not question *why* these classes were rich in behaviour. A closer examination of `Customer` and `Booking` illustrated that a majority of their methods were getting and setting their instance variables. `Customer` and `Booking` were holding data instead of deriving information.

Finally, the student reported that TOAD did not provide a means for them to *tag* certain classes in their designs as simple types. This resulted in a number of design classes incorrectly breaching the `CU1` and `CU2` heuristics which led to a misleading indication of high coupling in a number of their design classes. Figure B—7 illustrates a new feature in TOAD that allows learners to explicit state which classes in their design should (not) be evaluated by TOAD as user-defined classes.

For example, `String` in Figure B—7 is shown to be an empty class that is provided for in the target programming language. Figure B—7 illustrates how `Boolean`, `Duration`, `String`, `Date`, `Name` and `Address` can be

excluded from the analysis of Abbey design. Typically, the student would evaluate the design first and based upon their first assessment decide which classes did not contribute to the review process. These classes become prime candidates for being excluded if the student wishes to re-evaluate their design in a more focused manner.

Figure B—7 : Selecting simple types in the design

In light of the increased functionality inherent in `Design B`, it breached a similar number of heuristics as `Design A`. Furthermore, the student was aware of how to improve `Design A` further having applied TOAD but was unable to commit these changes in the allotted time-frame. `Design B`,

illustrated in Figure B—8, was better encapsulated and more decentralised than `Design A`.

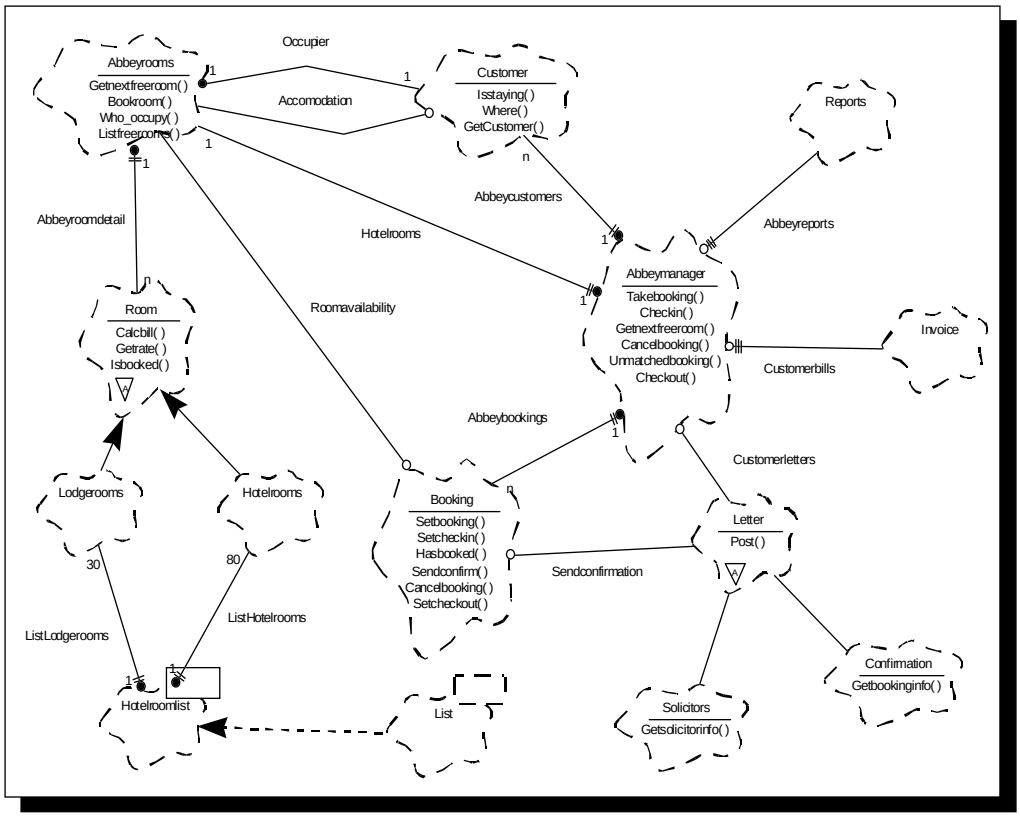


Figure B—8 : Design B for the Abbey system

B.4 Summary

We have described in this appendix the means by which designs are evaluated within TOAD. It is important to realise that the majority of the decision-making process is carried out by the *practitioner*. TOAD provides a means for identifying the problems, reasoning about them and presenting a number of solutions that are applied at the user's discretion.

Appendix C

Further acknowledgements

We wish to thank all those institutions that took time to complete our surveys and those companies/individuals that supported our research efforts—*thankyou*.

Survey respondents

STAFFORDSHIRE UNIVERSITY
UNIVERSITY OF NORTHUMBRIA
UNIVERSITY OF SUNDERLAND
UNIVERSITY OF HUDDERSFIELD
UNIVERSITY OF CAMBRIDGE COMPUTER LABS
UNIVERSITY OF PORTSMOUTH
UNIVERSITY OF EAST ANGLIA
UNIVERSITY OF DUNDEE
UNIVERSITY OF CENTRAL ENGLAND
PAISLEY UNIVERSITY
UNIVERSITY OF ULSTER
UNIVERSITY OF NEWCASTLE UPON TYNE
NAPIER UNIVERSITY
UNIVERSITY OF ESSEX
UNIVERSITY OF TEXAS
OHIO STATE UNIVERSITY
UNIVERSITY OF WISCONSIN
NORTH CAROLINA STATE UNIVERSITY
SWINBURNE UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF ILLINOIS
BUSINESS SCHOOL OF COPENHAGEN
SOUTHERN ADVENTIST UNIVERSITY
THE OPEN UNIVERSITY
BOURNEMOUTH UNIVERSITY
QUEEN'S UNIVERSITY OF BELFAST
UNIVERSITY OF HULL
UNIVERSITY OF WALES
UNIVERSITY COLLEGE LONDON
ANGLIA POLYTECHNIC UNIVERSITY
UNIVERSITY OF KENT
HERIOTT-WATT UNIVERSITY
LEEDS METROPOLITAN
UNIVERSITY OF GLASGOW
NOTTINGHAM TRENT UNIVERSITY
UNIVERSITY OF EXETER
UNIVERSITY OF NOTTINGHAM
UNIVERSITY OF ALBERTA, CANADA
SOUTHERN UTAH UNIVERSITY
EAST TENNESSEE STATE UNIVERSITY
IBM WORLD-WIDE
KYUSHU INSTITUTE OF TECHNOLOGY, JAPAN
CLEMSON UNIVERSITY
BERKLEY UNIVERSITY
UNIVERSITY OF NEW ORLEANS
AARHUS UNIVERSITY
UNIVERSIDAD DE MOITERREY

The implementation gurus

Thank you to John Lilley for his advice on writing robust grammars and whose efforts indirectly aided in the implementation of the document translator for converting C++ source code into the class description language.

Thank you to Premukar Devanbu for providing the gen++ tool for parsing C++ source code and to Linda Rising for introducing me to numerous resources regarding software metrics.

And last but not least to William Armitage, our departmental system administrator. Several years later under his administration and after reading numerous tomes on operating systems and networks, I still feel no closer to establishing the bounds to this man's UNIX knowledge. Thanks for that Will!

And the rest

A big thank you to the students that took the 95/96 and 96/97 OBJ courses at the University of Nottingham. Their invaluable feedback on the use of design heuristics and their altruistic approach to the course made it an even more enjoyable experience teaching them.

Finally, thanks to a number of companies that have showered us with tools, beta releases, advice and publications throughout our research. Of these, a special thanks goes out to the ACM sponsored conference OOPSLA and their students volunteer program. I cannot begin to gauge the experience gained from attending their conferences, only made possible by the generous financial support they extend to students— ***THANK YOU OOPSLA!***

Appendix D

Further reading, resources and research

A large body of research regarding design heuristics and software metrics currently resides at the TOAD web site:

`http://www.cs.nott.ac.uk/~cag/toad.html`.

The `heuristics@cs.nott.ac.uk` mailing group was set up to discuss design heuristics and has contributed greatly to the ongoing efforts to comprehensively document them. The latest version of the `Object-Oriented Design Heuristic` catalogue v1.0a can also be found at the TOAD web site.

The current list of TOAD related publications are thus:

Gibbon, C.A. and Higgins, C.A., "The case for design Heuristics in OO curricula", Educators' Symposium, OOPSLA'97, October, 1997.

Gibbon, C.A. And Higgins, C.A., "An informal approach to software design evaluation", INSPIRE'97, BCS Quality, Gothenburg, 18-19 August, 1997.

Gibbon, C.A. and Higgins, C.A., "Towards a learner-centred approach to testing object-oriented designs", Proceedings of the Asian-Pacific Software Engineering Conference, Seoul, South Korea, December 4-7, 1996.

Gibbon, C.A, Lovegrove, G., Higgins, C.A., "Tools and techniques to assist OO education", Educators' Symposium, OOPSLA'96, San Jose, California, 1996.

Gibbon, C.A., "Evaluating and visualising object-oriented designs", Doctoral Symposium, OOPSLA'96, San Jose, California, 1996.

Gibbon, C.A. and Higgins, C.A., "Teaching object-oriented design with heuristics", SIGPlan Notices, 31(7), July 1996, pages 12-16.

Gibbon, C.A., "Modelling system dynamics with classes", Tech. Report, NOTTCS-TR-97-2, Department of Computer Science, University of Nottingham, ENGLAND.

Gibbon, C.A., "Object-oriented design heuristics: a working document", Internal Report, Department of Computer Science, University of Nottingham, ENGLAND.

Appendix E

Applying class role migration

This appendix presents the results of applying class role migration to OO software. Section E.1 first discusses the motivation behind class role migration before describing its models and their implementation in section E.2. We demonstrate that class role migration provides a good mechanism for highlighting those classes within software that are most likely to change and at what cost to the rest of the system. Section E.3 subjectively orders the classes within three OO systems according to their maintainability properties. These classes are then ranked objectively using class role migration. In section E.4 we draw a number of conclusions based upon a comparative analysis of the presented results.

E.1 Motivation

Class role migration asserts that by designing for change, at design time, system modifications are less costly to fix than during implementation. To this end, class role migration assesses every class in a design model to determine the likelihood of them undergoing future changes and the cost that these changes would have on rest of them system. Class role migration provides a means to *prioritise* classes in large software architectures. This enables developers to review issues of maintainability within their OOD documents in a *structured* and *comprehensive* manner.

However it is classes, and not objects, that exist during the design phase. This results in models of design maintainability that are restricted to evaluating the available static properties in an OOD. The model of maintainability supported by class role migration articulates the static properties of an OOD in a dynamic manner by taking advantage of the inherent client/server relationships that exists between classes. In this model

a class is assigned a behavioural role for every class relationship that it participates in. Table E—1 lists the server, agent and actor roles that a class may assume for the aggregation, using and inheritance relationship. Class role migration highlights those classes that are most likely to change *role* based upon the well-documented interactions between servers, actors and agents *and* the design context in which these classes are set. An estimate of the cost of a role migration on the rest of the system is a function of the number of classes that the proposed role change will affect.

Relationship	Role	Description
Inheritance	server	has only descendent classes
	actor	has only ascendent classes
	agent	has both descendent and ascendent classes
Aggregation	server	only contained by other aggregates
	actor	only contains other aggregates
	agent	both contains and contained by aggregates
Using	server	only provides class services
	actor	only uses class services
	agent	both provides and uses class services

Table E—1 : All roles for all class relationships

E.2 Class role migration within TOAD

E.2.1 Ordering role migrations

Given the class roles server, actor and agent, Table E—2 lists all possible role migrations. Those belonging to set#1 are improbable owing to the inverse relationship that exists between actors and servers; it is unlikely that a server will become an actor or an actor a server. Of those remaining, set#3 migrations are preferred over those in set#2; the former representing software changes that are additive where the latter may incur the removal of system functionality [Gi97a].

A role migration documents a design change for a particular relationship. For example, the actor/agent role migration on a class `C` for the inheritance relationship refers to the subclassing of the target class `C`. A *migration form* is used to document the causes, effects and consequences of all role migrations

together with their likelihood and cost to the system. Gibbon [Gi97a] used these migration forms to document the role migrations listed within Table E—3. Class role migration within TOAD objectifies the information contained on these migration forms to automate their deployment during design evaluation.

Migration Set	Migration Type
set#1	server/actor actor/server
set#2	agent/server agent/actor
set#3	actor/agent server/agent

Table E—2 : All possible role migrations

Migration Type	Cost
agent/server	[most costly]
agent/actor	
server/agent	
actor/agent	[least costly]

Table E—3 : Prioritising sets #2 and #3 role migrations

E.2.2 Likelihood and cost

Class role migrations within a design are ranked according to their likelihood and cost attributes, where:

- *likelihood* identifies those classes most susceptible to change;
- *cost* determines the impact on the system resulting from a role migration.

A *ranking function* is used to prioritise role migrations within an OOD. It is applied to all role migrations delivering a priority value between 0 and 1, where 0 is the lowest priority and 1 the highest. It is these priority values that are used to order role migrations; classes with a high priority value are

considered a maintenance risk. There are essentially three types of ranking function:

```
priorityValue        =  likelihood()
priorityValue        =  cost()
priorityValue        =  weightingFunction()

where

weightingFunction()  =  likelihood()*W1 + cost()*W2
                        W1,W2 are weighting factors summing 1
```

Using these ranking functions, role migrations can be ordered by the likelihood of change, the cost of that change or a combination of the two by applying the *weighting function*. Figure E—1 illustrates how the weighting function can be adjusted using weighting factors to make it likelihood-biased or cost-biased.

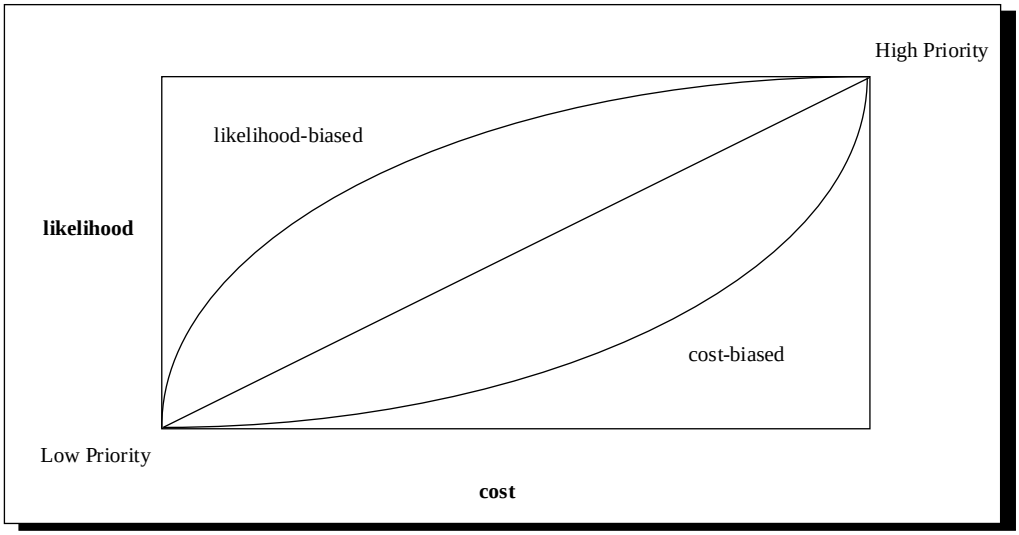


Figure E—1 : Ranking classes with weighting functions

Two factors contribute to the likelihood and cost calculations of a class role migration; the type of role migration (see Table E—3) and the design context. The migration type provides insight into the typical behaviour of classes in this role. However, it is the context in which a class is set that ultimately determines the likelihood and cost of a role migration. The *class context* holds

information on the position of a class in a hierarchy and how it interacts with its neighbouring classes. From the context, class role migration will reason about likelihood and cost according to the type of class relationship and the paradigmatic concepts that influence the role migration.

Consider the agent/actor class role migration for a class `C` within an inheritance hierarchy that represents the pruning away all descendent classes of `C`. The likelihood of this happening depends upon a number of factors within the inheritance hierarchy described by its class context:

- breadth and depth of inheritance hierarchy;
- position of `C` in the hierarchy;
- abstract or concrete nature of `C`;
- number of descendent classes to `C`;
- number, type and abstract nature of ascendent classes to `C`.

and class features outside the inheritance hierarchy:

- number of (polymorphic) clients to descendent classes of `C`;
- number of (polymorphic) clients to `C`.

Hence, when determining the likelihood of a potential class role migration, common applications of object technology contribute important contextual information. Also from the class context, role migration can determine the visibility of its clients to estimate the impact of potentially costly propagating changes. For example, descendent class pruning of `C` is less costly if the visibility between `C` and its immediate child classes is *private*. This would confine class changes to the inheritance hierarchy so those instantiating clients of `C`'s descendent classes would not be affected.

Gibbon [Gi97a] details examples of the causes, effects and consequences of class role migrations within inheritance hierarchies and using graphs to

better illustrate the benefits of prioritising classes according to this model of maintainability. The remainder of this section reifies the presented concepts by analysing the results of applying inheritance-based class role migration to OO software.

E.3 Analysing the Results

Class role migration for the inheritance relationship was applied to three OO systems listed in Table E—4. We aim to demonstrate that role migration can accurately order the classes within these systems by the *likelihood* of them undergoing maintenance. Based upon these preliminary results, future research will seek to validate class role migration for ordering classes according to their cost and for the aggregation and using relationships.

OO Software	Classes	Inheritance classes	Root classes
AISEARCH	28	28	5
TOAD	120	105	11
Gnu C++ v2.7.2	192	70	16

Table E—4 : Class library statistics

The tables at the rear of this appendix document the results of applying class role migration. Every system listed in Table E—4 has a subjective table and an objective table. The former depicts the subjective ordering of change prone classes and the latter uses class role migration to rank them. The following sections evaluates each system in turn by comparing and contrasting their subjective ranking against their objective one. A discussion on the common problems and the practical benefits of role migration is given in section E.3.4.

E.3.1 AISEARCH

AISEARCH is a set of reusable search algorithms that the designer must inherit from before they can apply them. The majority of the inheritance hierarchies within AISEARCH were broad, shallow and had a small number of classes. This resulted in a small number of classes in the system assuming the agent role and a large number of actors that were equally likely to undergo a role migration.

Only one discrepancy between the subjective and objective ordering of classes in AISEARCH occurred. In the VOBJECT_ hierarchy, Table E—6 ranks the AONODE_ agent/server role migration higher than that of UNINODE_. However upon closer examination this was an error on the part of the subjective ordering. Given that both AONODE_ and UNINODE_ have the same ancestry and that AONODE_ has a greater number of concrete descendent classes, class role migration was correct in assuming that UNINODE_ was more likely to become a root class than AONODE_.

E.3.2 TOAD

The subjective orderings within TOAD were identical to that of the objective except for the server/agent role migrations. This role migration looks at the probability of generalising on the root class. From Table E—9, the objective ordering informs the designer that TIterator is more likely to undergo this role migration than TAnalyser. However, from having an intimate knowledge of the TOAD system the subjective ordering places TAnalyser above that of TIterator. This is because TAnalyser exports more concrete behaviour than TIterator and also possesses a greater number of methods. Although these role properties could be tested for objectively, the current implementation of class role migration does not support these features.

E.3.3 Gnu C++ library

Class role migration disagreed on two counts with the subjective ordering of classes that comprised the `Gnu C++ library` version 2.7.2 for UNIX. The first case involved determining the likelihood of servers (root classes) being generalised. A number of 2-class inheritance hierarchies were shown to be equally likely of undergoing this server/agent role migration. However, class role migration failed to realise that if the root class was concrete, it was more likely to be generalised than a root class that was already abstract. In this case it meant placing the abstract root classes `BaseDLList` and `BaseSLList` below these 2-class inheritance hierarchies that had concrete roots. The need to consider the abstract nature of root classes has now been incorporated into class role migration.

The second discrepancy involved the agent/server role migration within the `_ios_fields` hierarchy that documents the likelihood of a class becoming a root class; that is to prune away its ascendent classes. Class role migration reported that `fstream` and `iostream` were equally likely to become root classes based upon their position in the hierarchy, the number and type of ascendent classes and the amount of descendent classes that depended upon them. However, class role migration did not take into account the dependencies between the `fstream` and `iostream`. From Figure E—2 we note that `fstream` inherits from `iostream`. Therefore, `iostream` will always be more likely to become a root class than `fstream`. A means to model the dependencies between (equally likely) classes needs to engineered into class role migration.

E.3.4 Discussion

The results of applying class role migration to inheritance hierarchies has raised a number issues. As anticipated, the true benefits of class role migration come from applying it to large inheritance hierarchies, in particular, those that are more than two levels deep. For broad and shallow hierarchies class role migration will need to implement more discerning

mechanisms for differentiating between equally likely classes. Possible directions for these more discriminatory techniques were reported in sections E.3.2 and E.3.3 which included the abstract nature of the root classes and modelling the dependencies between classes.

A practical issue that needs to be considered is the time taken to subjectively order classes in a system. Table E—5 clearly illustrates the advantages of automating the ordering of classes with class role migration. The high correlation between the subjective and objective orderings illustrates that class role migration is a viable means for ranking change prone classes for the systems evaluated in this appendix. However, alternative mechanisms of experimentation reported in section E.5 rely upon future research to fully validate class role migration before it can be confidently applied to OO software.

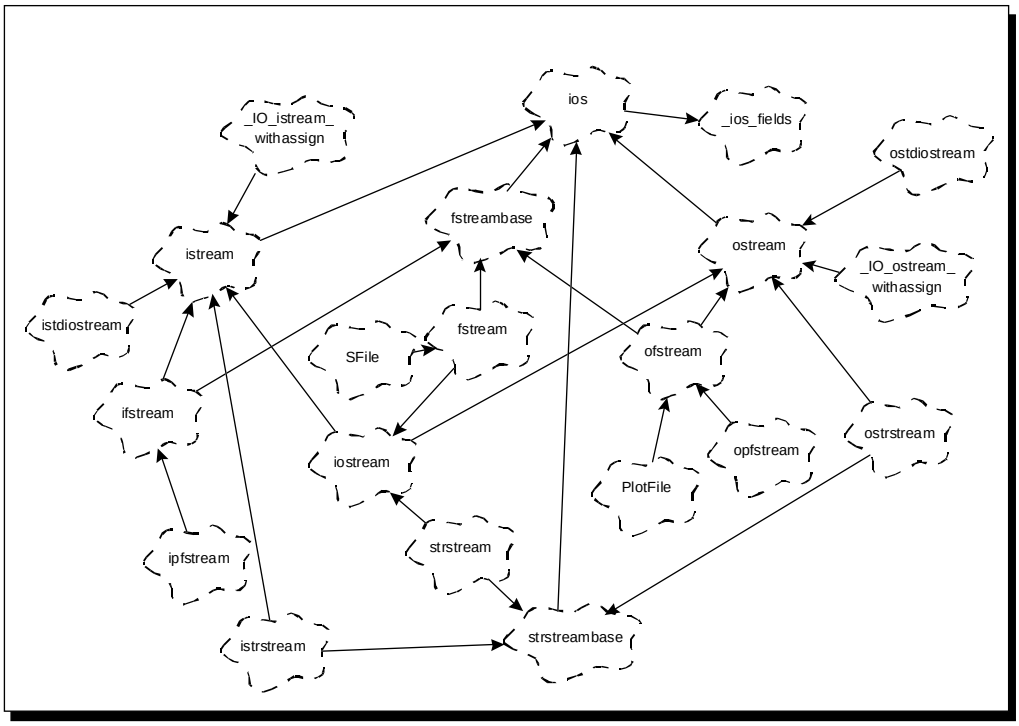


Figure E—2 : The Gnu C++ `_ios_fields` inheritance hierarchy

To summarise, class role migration is a novel approach for identifying classes likely to undergo future changes and at what cost to the system. It draws directly upon the inherent client/server relations that exists between classes

to model the static properties of an OOD in dynamic manner. Preliminary results have illustrated that the inheritance-based role migrations have a high correlation with subjective opinion on deciding which classes are most likely to change in a class hierarchy. Experiments to verify the ordering of classes by their cost and role migrations for the aggregation and using relationships is left for future research. However, the results have highlighted a number of areas that class role migration needs to address if it is to become a *pragmatic* means for evaluating future changes in OO software. This concerns are reported in the next section.

OO software	Time in minutes
AISEARCH	35
TOAD	55
Gnu C++ library	195

Table E—5 : Time taken to subjectively order classes in the systems

E.4 Future Work

Our migration model documents maintenance tasks as a series of role changes. However, at a finer level of granularity, it would be useful to model system changes that do not result in a role migration. For example, an agent class in an inheritance hierarchy might have two parent classes and a large number of descendent classes. Removing one of these parent classes is a marked shift *towards* an agent/server role migration but would not be signalled as such unless *both* parent classes were removed. Modelling the *degree* of class role migration is left for future research.

Another issue concerning the maintainability model is defining how class role migrations inter-relate. Consider an agent/actor role migration on a class `C` for the aggregation relationship. This entails the removal of `C` from the implementation of all classes `S` that contain it. Let `P` be those classes within `S` that provide accessor methods to `C`. The clients of `P` that depend upon these accessor methods now form an integral part of the change equation for the

original agent/actor role migration on `C`. However, the emphasis has now shifted away from the aggregation relationship to that of the using. Research into the inter-dependencies between role migrations belonging to different class relationships needs to be modelled to gain more insight and finer control over system wide changes. Preliminary research into the inter-dependencies between role migrations has given the ordering on class relationships illustrated in Figure E—3 according to perceived costs to the system [Gi97a].

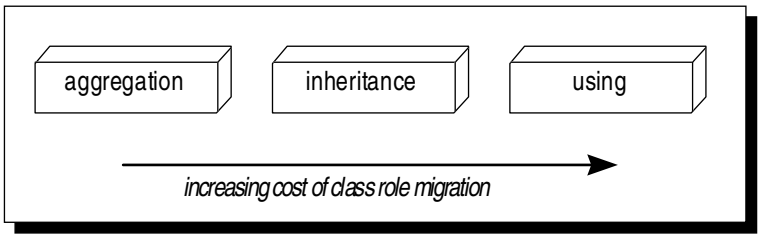


Figure E—3 : Ordering of relationship role migrations

Finally, given this model of maintainability and its preliminary results received to date, a series of pragmatic questions regarding its application need to be addressed:

- What happens when a developer is presented with a potentially costly and/or highly likely class role migration?
- What feedback and/or advice is available to the developer, at design time, to correct the problem?
- Can we partially mechanise the process of fixing the identified maintenance problem?

Considering that each role migration embodies a well-bounded maintenance task, it seems likely that developers could be referred to solutions to the observed design problems. Documenting role migrations so that they are more effective design aids is left for future research.

E.5 Conclusions

Class role migration uses a behaviour-driven model of maintainability to evaluate the likelihood of future software changes and their cost to the rest of the system. Class role migration was applied to three OO systems and subsequently evaluated. The promising results gained from these initial experiments warrants further investigation into class role migration by applying it to larger and yet more diverse design architectures. Unfortunately, the lack of available historical project data on the *actual* maintenance activities performed on the evaluated software meant that the presented results require further validation. Access to such data would provide a better indication of which classes changed frequently and at what cost to the system. Nevertheless, in the absence of historical project data, correlating subjective expertise with actual migration results provided valuable insights into the advantages and limitations of class role migration. We feel that with further research into this model of maintainability an effective and practical mechanism for the timely identification of change prone classes within large OO software systems would result.

E.6 Class role migration tables

Within every table, role migrations are ranked, by likelihood, from high to low. For example in Table E—6, `SVOBJECT_` is more likely to undergo an Agent/Server role migration than `NODE_`, which is more likely than `AONODE_` and so on.

Within a particular table cell, a list of classes prefixed with an * denotes classes that are equally likely to undergo maintenance. Starred lists of classes that are separated by an — denote a different group of equally likely classes with the upper group being more likely to change than the lower. In Table E—6, `UNICOST_GRAPH_`, `UNICOST_TREE_`, `BEST_` are equally likely to undergo role migration, which are more likely than `BREADTH_GRAPH_`, `BREADTH_TREE_`, `DEPTH_GRAPH_` and `DEPTH_TREE_`. The shaded boxes indicate that there were no role migrations of that type in the hierarchy. The

bold italicised classes in the objective tables highlights discrepancies with the subjective orderings. Finally, the ALL hierarchy is a special hierarchy used for ordering Server/Agent role migrations.

Hierarchy	Actor/Agent	Server/Agent	Agent/Actor	Agent/Server
AOSEARCH_	*AODEPTH_TREE_ *AOBREACH_TREE_			
LIST_	SORTEDLIST_			
BISEARCH_	*BIBREADTH_GRAPH_ *BIBREADTH_TREE_ *BIDEPTH_GRAPH_ *BIDEPTH_TREE_			
SEARCH_	*UNICOST_GRAPH_ *UNICOST_TREE_ *BEST_ — *BREADTH_GRAPH_ *BREADTH_TREE_ *DEPTH_GRAPH_ *DEPTH_TREE_			
VOBJECT	*ANDNODE_ *ORNODE_ *BEST_NODE_		UNI_NODE AO_NODE NODE_ SVOBJECT_	SVOBJECT_ NODE_ AONODE_ UNINODE_
ALL		LIST_ AOSEARCH_ BISEARCH_ VOJBECT_ SEARCH_		

Table E—6 : Subjective ordering of classes within AISEARCH

Hierarchy	Actor/Agent	Server/Agent	Agent/Actor	Agent/Server
AOSEARCH_	*AODEPTH_TREE_ *AOBREACH_TREE_			
LIST_	SORTEDLIST_	?		
BISEARCH_	*BIBREADTH_GRAPH_ *BIBREADTH_TREE_ *BIDEPTH_GRAPH_ *BIDEPTH_TREE_			
SEARCH_	*UNICOST_GRAPH_ *UNICOST_TREE_ *BEST_ — *BREADTH_GRAPH_ *BREADTH_TREE_ *DEPTH_GRAPH_ *DEPTH_TREE_			
VOBJECT	*ANDNODE_ *ORNODE_ *BEST_NODE_		UNI_NODE AO_NODE NODE_ SVOBJECT_	SVOBJECT_ NODE_ ***UNINODE***_ ***AONODE***_
ALL		LIST_ AOSEARCH_ BISEARCH_		

		VOJBECT_ SEARCH_		

Table E—7 : Objective ordering of classes within AISEARCH

Hierarchies	Actor/Agent	Server/ Agent	Agent/Actor	Agent/ Server
TClassMetric	*ClassSize *ClassStability			
TAnalyser	*AAnalyser *IAnalyser *UAnalyser		TRelAnalyser	TRelAnalyser
TSimpleClassItem	BaseClass *ClassAttribute *ClassMethod		TBasicClassItem	TBasicClassItem
TQuality	*AggrClassQuality *ClassQuality *InheritClassQuality *UsingClassQuality *AggrRelQuality *InheritRelQuality *UsingRelQuality		TRelationQuality TClassQuality	TClassQuality TRelationQuality
TResponseClass	*AggrRC *InheritRC *UsingRC			
TTOADClass	*ATOADClass *ITOADClass *UTOADClass			
TReport	DesignReport *CBandwidth *CCollaborators *CEncapsulation *CImplementation *CInheritance *ClassReport — *AggregationReport *InheritReport		TRQualityReport TQualityReport TCategoryReport	TQualityReport TRQualityReport TCategoryReport
TIterators	*BagPtrIterator *SetPtrIterator		*BagIterator *SetIterator	*BagIterator *SetIterator
TIterableContainer	*BagPtr *SetPtr		*UnBoundedBag *UnBoundedSet — *Bag *Set — MemberContainer	MembeContainer — *Bag *Set — *UnBoundedBag *UnBoundedSet
TRoleMigration	*IActorAgent *IServerAgent *IAgentServer *IAgentActor		TInheritMigration	TInheritMigration
TParser	*DesignParser *VerifyParser			
BasicComponent	*ToadReport ... 15 equally likely classes *ControlBox — *SortedScrolledList *SortedList *FloatThresholdBox *IntegerThresholdBox		*TScrolledList *Thresholdbox UIComponent	UIComponent *TScrolledList *ThresholdBox

ALL		TGenerateMop *TClassMetric *TParser — *TTOADClass *TResponseClass TSimpleClassItem TAnalyser TIterator TRoleMigration TQuality TReport BasicComponent TIterableContainer		

Table E—8 : Subjective ordering of classes within TOAD

Hierarchies	Actor/Agent	Server/ Agent	Agent/Actor	Agent/ Server
TClassMetric	*ClassSize *ClassStability			
TAnalyser	*AAnalyser *IAnalyser *UAnalyser		TRelAnalyser	TRelAnalyser
TSimpleClassItem	BaseClass *ClassAttribute *ClassMethod		TBasicClassItem	TBasicClassItem
TQuality	*AggrClassQuality *ClassQuality *InheritClassQuality *UsingClassQuality *AggrRelQuality *InheritRelQuality *UsingRelQuality		TRelationQuality TClassQuality	TClassQuality TRelationQuality
TResponseClass	*AggrRC *InheritRC *UsingRC			
TTOADClass	*ATOADClass *ITOADClass *UTOADClass			
TReport	DesignReport *CBandwidth *CCollaborators *CEncapsulation *CImplementation *CInheritance *ClassReport — *AggregationReport *InheritReport		TRQualityReport TQualityReport TCategoryReport	TQualityReport TRQualityReport TCategoryReport
TIterators	*BagPtrIterator *SetPtrIterator		*BagIterator *SetIterator	*BagIterator *SetIterator
TIterableContainer	*BagPtr *SetPtr		*UnBoundedBag *UnBoundedSet — *Bag *Set — MemberContainer	MembeContainer — *Bag *Set — *UnBoundedBag *UnBoundedSet
TRoleMigration	*IActorAgent *IServerAgent *IAgentServer *IAgentActor		TInheritMigration	TInheritMigration
TParser	*DesignParser *VerifyParser			
BasicComponent	*ToadReport ... 15 equally likely classes *ControlBox — *SortedScrolledList *SortedList		*TScrolledList *Thresholdbox UIComponent	UIComponent *ThresholdBox *TScrolledList

	*FloatThresholdBox *IntegerThresholdBox			
ALL		TGenerateMop *TClassMetric *TParser — *TTOADClass *TResponseClass TSimpleClassItem ***TIterator*** ***TAnalyser*** TRoleMigration TQuality TReport BasicComponent TIterableContainer		

Table E—9: Objective ordering of classes within TOAD

Hierarchies	Actor/Agent	Server/ Agent	Agent/Actor	Agent/ Server
BaseDList	DLList			
BaseDLNode	DLNode			
BaseSLList	SList			
RNG	*ACG *MLCG			
input_iterator	istream_iterator			
_IO_marker	streammarker			
Random	*Weibull *Uniform *Poisson *NegativeExpntl *Geometric *Erlang *DiscreteUnform *Binomial LogNormal		Normal	Normal
_IO_FILE_	*indirect_buf *strstream_buf *edit_streambuf — *func_parsebuf *general_parsebuf *string_parsebuf *procbuf *stdiobuf		filebuf parsebuf streambuf	streambuf filebuf parsebuf
SampleStatistic	SampleHistogram			
_ios_fields	*_IO_istream_withassign *istdiostream *_IO_ostream_withassig n *ostdiostream *istrstream *ostrstream *ipfstream *opfstream *PlotFile strstream Sfile		ifstream fstream ofstream *iostream *strstreambase fstreambase istream ostream ios	ios strstreambase fstreambase istream ostream ifstream ofstream *iostream *fstream
randomAccess	reverse_iterator			
ALL		*SampleStatistic *IO_marker *bidirectional_iterator *random_access_iterator *input_iterator *BaseDLNode		

		*BaseSLNode — *BaseDLList *BaseSLList RNG output_iterator unary_iterator Random binary_function _IO_FILE _ios_fields		

Table E—10 : Subjective ordering of classes within Gnu C++

Hierarchies	Actor/Agent	Server/ Agent	Agent/Actor	Agent/ Server
BaseDList	DLList			
BaseDLNode	DLNode			
BaseSLList	SList			
RNG	*ACG *MLCG			
input_iterator	istream_iterator			
_IO_marker	streammarker			
Random	*Weibull *Uniform *Poisson *NegativeExpntl *Geometric *Erlang *DiscreteUnform *Binomial LogNormal		Normal	Normal
_IO_FILE_	*indirect_buf *strstream_buf *edit_streambuf — *func_parsebuf *general_parsebuf *string_parsebuf *procbuf *stdiobuf		filebuf parsebuf streambuf	streambuf filebuf parsebuf
SampleStatistic	SampleHistogram			
_ios_fields	*_IO_istream_withassign *istdiostream *_IO_ostream_withassign *ostdiostream *istrstream *ostrstream *ipfstream *opfstream *PlotFile strstream Sfile		ifstream fstream ofstream *iostream *strstreambase fstreambase istream ostream ios	ios strstreambase fstreambase istream ostream ifstream ofstream ***iostream*** ***fstream***
randomAccess	reverse_iterator			
ALL		*SampleStatistic *IO_marker *bidirectional_iterator *random_access_iterator		

		*input_iterator *BaseDLNode *BaseSLNode ***BaseDLList*** ***BaseSLList*** RNG output_iterator unary_iterator Random binary_function _IO_FILE _ios_fields		

Table E—11 : Objective ordering of classes within Gnu C++