ACM SIGPLAN Language Tips

Check for updates

C++ Toolbox

Editor: *G. Bowden Wise, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180; wiseb@cs.rpi.edu*

Teaching Object-Oriented Design with Heuristics

C. Gibbon and C. Higgins

The transition from "programmer" to "class designer" is difficult. Pedagogic challenges involve imparting an in-depth knowledge of the object-oriented (OO) paradigm; gaining a firm understanding of the software engineering principles at play and knowing when and where to apply them effectively.

Curricula strucuture and mutating course requirements impose teaching and resource constraints that require better tools and techniques to meet their increasing demands. At the University of Nottingham, in the UK, our experience with design heuristics to transfer OO design (OOD) expertise to the inexperienced designer has proven to be a resourceful teaching aid.

This article discusses the implications and use of design heuristics at Nottingham in overcoming the resource constraints faced when teaching OOD. A recent survey of UK OOD curricla is given which highlights the problems attributed to teaching OOD in academia and how design heuristics can assist in OO education. A prototypical tool that semi-automates the heuristical analysis of C++ design documents is also discussed as an on-line teaching aid for learner designers.

1 The Teaching Task

1.1 Nottingham's OOD Curriculum

At the University of Nottingham, first year computer science undergraduate (CS1) students are taught how to program with abstract data types (ADT) using C++ as part of two contiguous compulsory modules. This object-based foundation is a prerequisite to the second year (CS2), semester 1 Object-Oriented Methods course (OBJ). The OBJ course introduces the fundamental principles underpinning object technology (OT) using well-documented OOD methods. Advanced C++ programming features such as inheritance, polymorphism and templates are taught to show how C++ implements object-oriented solutions.

Nottingham's CS department is relatively small compared to many other UK CS departments with only 62 students taking the OBJ course in 95/96. However, an expanding department has seen the OBJ course increase by 19% on last year, and 30% on the year before that. Next year brings further student increases but with few additional teaching resources. This poses manageable problems in lectures where the primary objective is information dissemination and knowledge impartation. Meeting this goal merely requires the location of a lecture theatre that can accomodate the surplus students. Unfortunately, increasing student numbers have more far reaching consequences in the laboratory where OO theory is put into practice.

[4] discusses the difficulties encountered at Nottingham and Staffordshire University in the UK. These *teaching constraints* can be summarised as follows:

- increasing staff/student ratios lead to undermanned lab sessions less capable of providing the necessary pragmatic object technology teaching support;
- the expensive tools required to provide seamless OOA/OOD/OOP limit the size of labs to the number of affordable software licences;
- the common ADT-based CS1 course approach place additional demands on CS2 courses to *undo* procedural thinking and *relearn* OO concepts.

It is unlikely that the staff/student ratio will improve or that CASE tools will become more accessible to academics. Furthermore, when faced with the dual teaching task of converting procedual programmers to OO programmers and then to class designers something is needed to optimise these transitional phases if the course is to deliver competent graduates.

1.2 Surveying UK OOD Curricula

Recently, we queried 27 UK universities on their current OOD course content and its future directions. Fur-

thermore, we inquired as to what OO programming language(OOPL) they used to implement OO solutions.

Not surprisingly, 90% of the survey respondents quoted C++ as the primary OOPL. Reasons included student demands and job prospects; availability of cheap (free) C++ resources pertaining to compilers, environments and public domain libraries; an abundance of C++ literature; and no shortage of staff with C++ knowledge. However, the majority acknowledged its unsuitability as a teaching language for object technology often preferring purer OOPLs such as Eiffel and Smalltalk. Only 5 universities plan to change from C++ to another OOPL, but of these, 3 were looking to move to C++'s sister language - Java!

Surprisingly, only one UK survey respondent used Smalltalk as the primary OOPL and another cited Modula-3. Smalltalk, Eiffel, Object Pascal and Modula-3 featured as secondary OOPLs in 8 of the universities, all of which employed C++ as their primary language.

Half of the survey respondents used a single OOD method as their teaching vehicle, a third utilised a method mix and the remainder did not employ OOD methods at all. Of the single OOD method institutions, OMT and Booch jointly shared the majority with Coad/Yourdon taking the minority. For mixed OOD methods, OMT and Booch again featured highly but with aspects of Wirfs-Brock inter-weaved. The general consensus among those that taught Booch and OMT as their main method was a move to the forthcoming Unified Method.

Of the survey subjects, only 6 introduced full OO from the outset, 15 preferred to use the object-based approach as a stepping stone to full OO status. In the latter case, it was felt that a solid foundation in ADT-based programming and data structures proved invaluable and gave the students a sound footing in things OO or otherwise.

Unfortunately, the survey did not attract institutions not using or planning to use object technology. This information is currently being sought by means of a second survey into UK OOD curricula.

1.3 Problems and Caveats

In light of the survey results and the teaching constraints identified in section 1.1, we believe teaching methods must encompass much more than any single OOD method affords. We concur with observations made in [5] that teaching methods should be highly pragmatic in their approach and that learning is best achieved by example. In [3] we outline the requirements for a teaching method by looking beyond OOD methods and by viewing them simply as process suggestions. In doing so, the teaching method takes responsibility for providing the teaching framework and does not leave it to the OOD method. It is important to separate these concerns in tailoring the teaching method to the needs and capabilities of the target teaching environment.

In the UK, it is unlikely that the primary OOPL (C++) will change or that a full OO approach will be adopted in the near future; the teaching method should accommodate these realities. In the next section we present design heuristics to aid in the teaching of OOD. In section 3 a prototypical tool that implements the design heuristics and automates their application is given. The results gleaned to date are illustrated in section 4 and call for assistance proposed in section 5.

2 Design Heuristics

2.1 What is and what is not a design heuristic?

A heuristic offers insightful information based upon experience that is known to work in practice. Heuristics can be used to guide the inexperienced on a path already taken by their experienced counterparts. To achieve this goal, heuristics must identify and encapsulate experience. Their subsequent application culminates in the reuse of this experience.

In the context of OOA/D, a design heuristic embodies design experience gleaned from erstwhile OO developers. At Nottingham, design heuristics that capture quality criterion such as maintainability and reusability can be (re)introduced to students when designing OO systems to enhance the quality of their software products. More importantly, design heuristics serve to educate and accentuate learners in OO techniques that address these important quality criteria.

A heuristic is not a metric. A heuristic is a rule of thumb that places its users in the region of what is correct [1]. It is not meant to be exact. In fact, heuristics benefit from their impreciseness by providing an informal guide to good(and bad) practices. Contrariwise, metrics are formal, rigid and precise. Metrics derive their advantages from formality and undergo stringent validation processes to achieve the required level of correctness. For pragmatic teaching purposes, this is both unnecessary and undesirable. We need informal techniques to drive the informal and highly pragmatic design processes employed

by teaching methods. Heuristics exhibit all of these properties.

Design heuristics are not new, however their existence has been subsumed by research in the software metrics field. [8] and [6] have explicity acknowledged and categorised design heuristics and numerous programming texts [7], [2] have touched upon design heuristics at the language level.

2.2 Objectives

The teaching constraints mentioned in section 1.1 presented some of the shortcomings befalling academic institutions teaching OOA/D. At Nottingaham we use design heuristics to supplement teaching methods with the following objectives:

- to supplement design expertise that cannot be disseminated during labs due to increasing staff/student ratios;
- to provide much needed student reassurance as to what constitutes good design practice and high quality products often limited in mainstream OOD methods;
- to raise awareness of software qualities such as maintainability and reusability and how they can be achieved using key OO concepts;
- to provide an intricate OO education framework that embodies not only diverse definitions of OO concepts and examples of usage but strucutured mechanisms for accessing them in the context of the learner's own design. For example, when a design heuristic is *triggered* teaching efforts are initially focused upon information relevant to that design heuristic and the concepts it uses;
- to capture design experience in a relatively objective manner so that heuristic detection may be automated.

It is not the intention of design heuristics to supplant the OO educator but to supplement their knowledge and displace their all-encompassing role in teaching OOD. At Nottingham, research effort into discovering and inventing design heuristics has been expended to produce a heuristic catalogue [4], [3], a collective term describing a set of design heuristics. But before we go any further, let use examine a design heuristic.

2.3 An Example

Consider the following design heuristic:

> An inheritance hierarchy should be no deeper than six.

How many times has this assertion been forwarded in literature? On each occasion it has been criticised for its vagueness and lack of substance, ultimately resulting in it being labelled inappropriate. We argue that this design heuristic and others like it are useful, if and only if, placed in the right context, substantiated with the required rationale and direct references to the OO principles they (ab)use is given. Promoters of similar design heuristics that fail to do this afford the credence to the aforementioned criticism.

Let us re-examine the inheritance design heuristic. A depth of six is arbituary and configurable. In the context of student designs, a depth of six serves only to trigger this design heuristic. Experience at Nottingham has shown that inheritance hierarchies that surpass this threshold are malformed and/or semantically incorrect. Upon trapping the offender it is the heuristic's job to explain why they are potentially at fault; illustrate exactly where in the design the anomaly occurred; and suggest, if possible, design alternatives. At this point the role of the heuristic is over. The heuristic does not have the intelligence nor the right to police or punish its offenders, merely to educate them in the design practices of the majority. Design heuristics that adhere to this code of practice make for useful OO teaching aids.

Although a simple design heuristic, depth of inheritance proves to be one of the more interesting ones. It raises issues encircling specialisation and reuse. For example, should inheritance be read as isA or is it valid to permit isASortOf; is inheritance specialisation; can we make use of delegation as a design alternative; and just how deep is too deep? The last of these proves the most controversial issue.

Contrary to OO literature, we do not believe that inheritance hierarchies should not strive to become deep. It is apparent when a keen, inexperienced student class designer has read around the subject by the *depth* of their first inheritance hierarchy. Blind promotion of deep and narrow hierarchies without warning has seen this heuristic trap a number of misguided students. Although the design heuristic cannot objectively determine the optimal depth, it can highlight the need for subjective verification to determine when deep is too deep. Incidently, as observed in

[9], this is when the depth of the inheritance hierarchy impedes reuse.

3 The TOAD System

The OBJ OOD course expects students to implement the software system outlined in the supplied requirement specification. The course gives a mandatory set of lectures and labs in which students use CASE tools to design and implement their software. Currently, design heuristics are presented during lectures to educate students on creating high quality products using key OO concepts. Owing to the success met with design heuristics in lectures, the next logical step was to apply the design heuristics directly to the learner's OOA/D products, at design time, giving immediate feedback on its areas of weakness as well as highlighting its strengths. To achieve this, a prototypical tool, TOAD (Teaching Object-oriented Analysis and Design), was implemented that could automate the detection and application of design heuristics and coordinate its feedback.

3.1 Applying Heuristics in TOAD

Design heuristics act upon design documents (i.e. C++ headers) to examine their static class properties. Using Booch's terminology, heuristic analysis views class (inter)relationships *aggregation*, *using* and *inheritance* both in isolation and then collectively. The latter involves the structured use of the design heuristics to provide design alternatives.

For example, if the design heuristic *limit multiple inheritance* is triggered for class A and it also triggers *avoid breaks in the class/type hierarchy* then the design alternative of delegation would be suggested. The heuristic analyser would have observed in the case of C++ that multiple inheritance involved private or protected inheritance that can be resolved by delegation. Although design heuristics are self-contained it is rarely the case that they proceed in isolation. In this example information of inheritance and delegation would have been dissemination using the students work as the design exemplar.

TOAD is scheduled to go on-line in October 1996 as part of the Nottingham's OBJ course described in section 1.1. The results of its usage will be presented in a subsequent paper.

4 Results

The benefits realised by employing design heuristics have been made apparent by the learners that have used them. Learners have been both undergraduate students taking the OBJ OOD course and commercial developers migrating to OT. In both cases the design heuristics were applied towards the back end of an OOD course when all the relevant concepts had been introduced. Introducing the design heuristics in this manner tied together all aspects of the course as well as re-visiting and re-analysing them in different contexts. It was obvious that the design heuristics touched upon several complementary and competing OO concepts in achieving their objectives. These needed to be explained, resulting in a more in-depth examination of the concepts involved.

The standard of OODs resulting from the application of TOAD will be available at the end of the first semester 96/97 OBJ course. These will be comp ared and constrasted with previous years to gain some insight into their usefulness. Student questionnaires and design experiments will also serve to affirm or refute our expectations.

5 Call for Assistance

We believe the discovery of design heuristics has been ongoing for years but without focus. It is imperative that design heuristics are actively sought and catalogued regularly so that they are promulgated back to the OO community at large. The discovery of new solutions, the rejection of old ones and the revision of existing design heuristics is an important factor in maintaining an up-to-date heuristic catalogue. To achieve this requires coordination and active participation by those people in a position to help, namely experienced designers, design methodologists and software metricians. To these people we seek feedback in the moulding of an informal heuristic catalogue.

We are in the process of constructing a web page (http://www.cs.nott.ac.uk/~cag/phd/toad.html) that will disseminate information on design heuristics and their application. Please visit our web site and contribute to the cataloging of design experience in the form of heuristics. It is hoped that an exchange of experience will ensue enabling the inexperienced to learn directly from the experienced.

A cursory but important note to CASE tool vendors. In creating TOAD we were faced with the problem of either

implementing our own CASE tool or parsing resultant design documents. This is because CASE tool vendors do not provide the *much needed* APIs for accessing design attributes illustrated in their OOD diagrams. This has had a notably effect on our development efforts. We opted for the latter by converting design documents (C++ headers) into flat files conforming to TOAD's object-oriented description language (OODL) for subseqent analysis. This is both an undesirable and costly exercise.

6 Acknowledgements

A special thanks goes to Dr. Gillian Lovegrove for helping to compile the OOD UK survey and clarifying a number of ambivalent research directions.

References

[1] BOOCH, GRADY. Rules of thumb. *Report on Object Analysis and Design (ROAD)* 2, 4 (Nov-Dec 1995), 2–3.

[2] COPLIEN, J. O. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, Reading, Mass., 1992.

[3] GIBBON, C. AND HIGGINS, C. Towards a learner-centred approach to teaching object-oriented design. In *Submitted Paper to APSEC'96* (December 1996).

[4] GIBBON, C., LOVEGROVE, G.L. AND HIGGINS, C. Tools, heuristics and techniques to assist oo education. In *Forthcoming OOPSLA'96* (October 1996), ACM Press.

[5] HENDERSON-SELLERS, BRIAN AND EDWARDS, JULIAN M. Identifying three levels of o–o methodologies. *Report on Object Analysis and Design (ROAD) 1*, 2 (July-August 1995), 25–28.

[6] JOHNSON, P AND FOOTE, B. Designing reusable classes. *Journal of Object-Oriented Programming 1*, 2 (1988), 22–35.

[7] MEYERS, SCOTT. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley Professional Computing Series, 1995.

[8] RIEL, ARTHUR J. Introduction to Object-Oriented Design Heuristics. *OOPSLA '94 Tutorial 38* (October 1994). Portland, Oregon.

[9] WANG, BARBARA LIN AND WANG, JIN. Is a deep class hierarchy considered harmful? *Object Magazine 4*, 7 (November-December 1994), 35–36.

Cleveland Gibbon is a Ph.D. candidate in the Computer Science Department, University of Nottingham, England. His primary research area is in object-oriented design heuristics and software product metrics. Other research interests include user interface design and visualising object-oriented software in virtual environments. He is a member of the ACM. He can be reached on cag@cs.nott.ac.uk or read-about at http://www.cs.nott.ac.uk/~cag.

Dr. Colin Higgins is a Senior Lecturer in the Computer Science Department, Univeristy of Nottingham, England. His primary research area is in AI and pattern recognition, particularly the recognition of cursive script. Other interests include object-oriented design and software product metrics using Prolog.