Identifying Novice Difficulties in Object Oriented Design

Benjy Thomasson, Mark Ratcliffe, Lynda Thomas
University of Wales, Aberystwyth
Penglais Hill
Aberystwyth, SY23 1BJ
+44 (1970) 622424
{mbr, ltt} @aber.ac.uk

ABSTRACT
We report on a study of novice programmers' object oriented class designs. These designs were analysed to discover what faults they displayed. The two most common faults related to non-referenced classes (inability to integrate them into the solution), and problems with attributes and class cohesion. The paper ends with some implication for teaching that may be indicated by the empirical results.

Categories and Subject Descriptors
D.1.5 [**Object-oriented Programming**]: Design
K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms
Design

Keywords
Software design, Introductory programming, Design faults

1. INTRODUCTION
For many years, educators have been concerned that beginning students of computer programming have not been performing to the levels expected. There is considerable evidence for this. For instance, in the much quoted international study by McCracken *et al* [7] of first year programming students who were attempting to solve an assigned programming problem, the average grade was just 21%.

1.1 Overview
The McCracken study looked at students' work on a complete software development task from design to coding. Subsequent work influenced by McCracken has focused on aspects of the whole task. A 2004 ITiCSE working group examined the ability of students to 'trace' code in multiple choice questions [6].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE'06, June 26–28, 2006, Bologna, Italy.
Copyright 2006 ACM 1-59593-055-8/06/0006...$5.00.

In this paper, we look at the 'other end' of the software development process and report on a study restricted to beginning computer science students' object-oriented *designs*. These designs have been examined in order to identify the most common faults. We hope that an understanding of students' errors in producing reasonable designs will help guide CS educators to focus effort in instruction.

This study reports on the initial stage of an ongoing project to develop a collaborative development environment (Vortex) that, through a case-based reasoning system, gives structured feedback to student-users and enables educators to capture information about their designs. Further information is available in [14].

2. BACKGROUND
Learning to program is not easy. Du Boulay [1] points out that there are issues of: orientation, learning to control the notional machine, understanding notation, acquiring standard structures and finally pragmatics. When educators teach beginning programming, they need to address all these issues at once and this is also not an easy task. We all have our own pet theories of how best to do this: objects first, breadth first, depth first, etc., so we may place more or less emphasis on Object-Oriented design in introductory classes, but almost all introductory texts seem to introduce the idea of a class with attributes and methods at some point in the first semester. In Aberystwyth we begin the introductory programming sequence (which is taught using Java) with a two pronged approach. We investigate programming language structures, but also spend quite a lot of effort in demonstrating and encouraging students to design classes. This is done through interactive lectures, closed lab sessions, and tutorials.

2.1 Design
As outlined in [11], most research in novice software development has tended to focus on learning and using a programming language; but some authors have examined design. Soloway *et al.* [12] discuss how to teach design in five phases with an emphasis on the idea of decomposing the problem and then selecting and composing plans to form a solution. More recently authors such as Muller [9] have discussed algorithmic patterns and how they may form a basis for students to recognise and solve variations on common problems.

2.2 Errors
Research into what kind of errors novice programming students make has also concentrated more on coding than on design. For

instance, in a recent paper, Hristova *et al.* [5] identified Java programming errors, but these were mainly at the level of syntax.

In addition to our personal experience there is considerable evidence that design is difficult for students, even those about to graduate. Spohrer and Soloway [13] noted nearly 20 years ago that students exhibit most difficulty in putting the 'pieces' of a program together rather than in programming language constructs *per se.* This could be generalised to the design arena. Only 9% of graduating seniors in a recent study [2] produced what the authors call a 'reasonable design' for a problem that required several classes and some interesting behaviour. Garner *et al.* [3] collected a count of student problems in labs and discovered that the second most frequent problem was 'stuck on program design' (behind 'basic mechanics' – small problems that were expected to diminish as time progressed.)

Or-Bach and Lavy investigated cognitive activities of abstraction in Object Orientation [10]. They found that third year students still struggled to sufficiently abstract entities when defining objects, thus missing necessary classes, placing irrelevant attributes within a class, and lowering cohesion.

Holland *et al.* [4] offer suggestions for how to avoid fostering misconceptions about Object-Oriented concepts in novices. These suggestions are based on their experience of designing and teaching introductory courses rather than on an empirical analysis of student designs. The misconceptions they highlight include:

- object/variable conflation – classes should have more than one instance variable,
- object/class conflation – there is usually more than one instance of a class,
- identity/attribute confusion – when a Module object, with identifier 'CS12320' has name as an instance variable with value 'CS12320' confusion ensues,
- Objects are not simple records – they may have different behaviour depending on their state (note that this is beyond the boundaries of this study).

All these misconceptions affect students' ability to design reasonable classes.

In this paper we concentrate on observable faults in students' designs rather than the misconceptions that may cause those faults, but clear inferences for improvement may be observed.

3. THE EXPERIMENT

In the first semester of all the Computing degrees at the University of Wales, Aberystwyth, students take an introduction to programming and object-orientation course that comprises between one third and one half of their time commitment. In this study we examined the designs that students produced about half way through the first semester.

The study was split into several phases. Students' designs, produced on paper in a lecture, were collected with the purpose of identifying the most common faults. There was considerable verification in the project as a whole, using different problems, different groups of students and, eventually, an early prototype of the Vortex tool that collected all the designs for later analysis. This paper concentrates on the studies that led to the categorisation of the errors that were exhibited in the student designs.

For all phases, students were instructed to produce class designs in the UML style that had been demonstrated by the lecturer, and introduced and used previously in practicals and tutorials. The problems were at the level of (in fact, very similar to) examples produced interactively in class and in tutorials, and were designed to examine at what level students had understood the process of decomposing a problem into its component parts. Our study focused specifically on the expression of class designs for a problem solution and required no representation of code or detailed behaviour. In fact, we only look here at class attributes as expressed by instance variables, not even method names.

3.1 Phase 1

In this phase we studied the designs of 180 students, 115 were novices and 65 had some programming experience. Results were collected on paper and students worked alone. The problem studied was the 'paper round':

> *Model classes for a paper round system identifying the classes required, their attributes and their methods. An individual paper round consists of a paperboy/papergirl delivering orders to various customers.*

3.2 Later Phases

Phase 2 occurred 3 weeks after Phase 1. The same students were given a different problem, in which they were asked to design classes for a car-hire company with multiple depots, cars and customers. The students were again asked only to produce class diagrams to support the final system. The class was divided so that some students (n=48) worked alone and others worked together (19 groups). Designs were again produced on paper and analysed manually. In Phase 3, conducted the following year, a different group of students (n=90) was given the car-hire problem. These students worked alone and produced designs on paper.

4. RESULTS

Obviously there may be variation in the solutions to these problems, but such variation would be minimal at the class level. A model solution was produced which identified classes. Student designs were then compared to the model for suitability and completeness. Firstly the designs were analysed for suitability, to ensure that the classes produced were relevant to the specification. Completeness was analysed by comparing the number of suitable classes successfully identified to those identified in the model answer.

Results from all phases were similar, in that almost all designs exhibited considerable problems. Designs were somewhat better when the students worked in a group (Phase 2) but the same kind of errors showed up (see Table 2). In this discussion, we will concentrate on the results of Phase 1 since they were typical of all phases.

4.1 Faults Identified in Phase 1

The model answer suggested 'Paper', 'Customer', 'Address', 'Order', 'Paperboy/girl', 'Round', and 'Person' as the necessary classes. The 'Person' class was not expected to be identified by many students as this illustrated inheritance, a topic not yet covered in the course.

4.1.1 Summary of Faults

Table 1 shows the extent to which students managed to identify individual *suitable* classes in a numeric sense. This shows that of 180 students, only 1 student managed to identify 7 suitable candidate classes. In fact this student's solution was identical to the model answer. We had expected that by simply reading the project specification a novice might reasonably identify 5 classes ('Paper', 'Round', 'Paperboy/girl', 'Order', 'Customer') and yet only 14% of first time programmers managed a design with 5 or more suitable classes. Most students were only able to identify 3 suitable classes.

Table 1. Results from Phase 1: Student design exercise

Experience	Number of classes identified						
	1	2	3	4	5	6	7
Complete novice n=115	1	14	45	39	13	2	1
Some Experience n=65	0	10	24	19	10	3	0

The designs were then analysed to see what design faults could be identified. The results are explained below with examples and discussion centring on the designs produced in Phase 1.

Table 2 illustrates the number and kind of faults identified in all three phases. In the rest of this section we describe each of these faults, but first we must note that design faults rarely occur in isolation, so identification of one fault usually leads to the discovery of further difficulty within the design.

Table 2. Individual fault occurrences

Phase:	1	2	3
Total classes	647	295	254
Total designs	180	59	53
Total Non-Referenced Class faults	388	129	112
Designs exhibiting this fault	97%	86%	83%
Total References to Non-Existent Classes	65	34	17
Designs exhibiting this fault	28%	36%	30%
Total Single Attribute Misrepresentation faults	208	35	34
Designs exhibiting this fault	73%	41%	36%
Total Multiple Attribute Misrepresentation faults	18	12	12
Designs exhibiting this fault	9%	19%	17%
Total Multiple Object Misrepresentation faults	2	0	0

4.1.2 Non-Referenced Class faults

Analysis of the students' designs revealed that most students developed a class in isolation and failed to utilise it within their proposed system. Novices understood the necessity for a concept, but were unable to relate it to other classes through appropriate use of instance variables[1].

For example the design shown in Figure 1 shows 'Customer', 'PaperPerson' and 'Order' classes. The Order and Customer classes are related by an attribute in the Order class, but nowhere in the design is there any reference to the PaperPerson class.

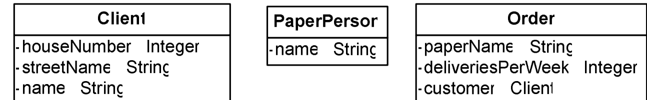

Figure 1. Non-referenced Class example

The results in Table 3 suggest that non-referenced classes are a serious problem for the novice programmers who took part in the study. Only 6 designs did not exhibit this problem and 42 exhibited only one such fault. The rest of the designs (132 of 180) manifested at least 2 instances of this design fault. One such fault *might* be an accident (although all students appeared to finish their designs in the time allotted). But we believe that two or more such faults are probably caused by inability to link the parts of the design into a coherent whole.

Table 3. Non-Referenced Class fault statistics

Total Non-Reference faults	388 (59.9%)
Designs with faults	174 (96.7%)
Max Non-Reference faults in one design	9
Avg Non-Reference faults per design	2.15

Figure 2 illustrates an example of a class which, though its author probably considered it 'complete', contains Non-Referenced Class faults because of a misconception about objects and classes.

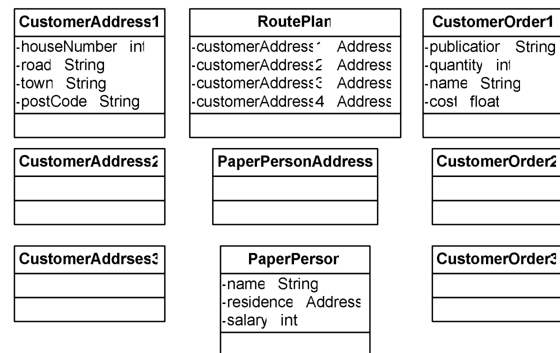

Figure 2. Worst Non-Referenced Class example

The design also reflects numerous others of the design faults that are discussed below, including Multiple Object Misrepresentations faults.

[1] Note that the designs were eventually entered into Vortex, where they could be automatically checked. In particular class names could be checked for spelling with WordNet [8] so missing class references due to simple misspellings were eliminated.

4.1.3 References to Non-Existent classes

In contrast to defining classes that are not referenced by the design, it was found also that novices made reference to classes which had not been defined.

If we rule out lack of time, as above, this fault may be caused either because the student considered the class 'too easy' to worry about at this stage e.g. Address, or had not worked through the design sufficiently to recognise its omission.

Table 4. Referenced Non-Existent Class Fault statistics

Total Referenced Non-Existent Class faults	65
Designs with faults	51 (28.3%)
Max Reference Non-Existent Class faults in one design	4

Table 4 shows that Referenced Non-Existent Class faults occurred in 28% of the designs, sometimes more than once. In total there were 65 separate references to non-existent classes.

Although references to non-existent classes result in an incomplete design, they generally result in one that is still comprehensible. The novice has consciously identified the requirement for the class through the use of the class name but has left the design incomplete.

4.1.4 Misrepresentations

Another fault identified was novices' failure to use good Object Oriented Design principles. These faults are broken down here into:

- Single Attribute Misrepresentations,
- Multiple Attribute Misrepresentations, and
- Multiple Object Misrepresentations.

Although these faults lead to less than ideal designs, they are different from the previous two kind of faults in that designs with these faults could lead to code that compiles and 'works'.

4.1.4.1 Single Attribute Misrepresentation

Cohesion is a measure of how strongly related and focused the responsibilities of a single class are. A class should model the data attributes and behaviour of one concept. A Single Attribute Misrepresentation fault occurs in two situation. In the first, a class has an instance variable that should be part of another class. For example in Figure 3, price would be better within the Paper class rather than the Orders class. The second occurs when an attribute should be an instance of a user-defined class but the novice resorts to using a *String*. This is also demonstrated in Figure 3 where most of the attributes would be better represented as instances of user-defined classes.

Design 81 Sample

Orders
-paper String
-price int
-date String
-address String

Figure 3. Single Attribute Misrepresentation example

Single Attribute Misrepresentations were the second most frequent faults observed within designs in Phase 1. Table 5 shows that almost a third of the total classes produced exhibited this fault. As previously noted, Or-Bach [10] also found that students placed irrelevant attributes within a class and thus lowered cohesion. This suggests that novice designers struggle to employ the good Object Oriented skills we expect from them. This is understandable, considering that the participants of the study were novices, and decisions regarding which aspects of the problem space to model as classes are difficult. However, eventually we would hope that students obtain the necessary decomposition skills to create good Object Oriented designs.

Table 5. Single Attribute Misrepresentation statistics

Total Single Attribute Misrepresentation faults	275
Total classes with faults	208 (32.2%)
Designs with faults	131 (72.8%)
Max Single Attribute Representation faults in one design	7
Average number of faults per design	2.1

Of the 180 designs, 49 did not exhibit this problem, 51 exhibited it once, and the rest exhibited it more than once.

4.1.4.2 Multiple Attribute Misrepresentation

A Multiple Attribute Misrepresentation fault occurs when two or more attributes within a single class should be bundled into another class. This is similar to a Single Attribute Misrepresentation but affects multiple attributes. Figure 4 provides an example of this error within a Customer class, where 4 attributes provide the address details for the Customer element of the design. Separating these entities increases cohesion and decreases coupling and provides a better Object Oriented Design.

Customer from Design 16

Customer
-name : String
-houseNum : int
-street : String
-town : String
-postCode : String
-newspaper : Order

Figure 4. Multiple Attribute Misrepresentation example

Nearly 9% of the designs analysed exhibited a Multiple Attribute Misrepresentation, see Table 6.

Table 6. Multiple Attribute Misrepresentation statistics

Total Multiple Attribute Misrepresentation faults	20
Total classes with faults	18 (2.8%)
Designs with faults	16 (8.9%)
Max Multiple Attribute Representation faults in one design	2
Average number of faults per design	1.25

4.1.4.3 Multiple Object Misrepresentation

The previous two faults relate to novices failing to use appropriate cohesion and object references in their class

definitions. There are however other faults that relate to the object references that they *do* provide.

Multiple Object Misrepresentations refer to the use of several objects of the same type, such as the situation in which a collection should be used. Instead novices provide numerous instance variables. A typical example of this kind of Multiple Object Misrepresentation is shown in Figure 5.

Route from Design 136

Route	
-residence1	Address
-residence2	Address
-residence3	Address
-residence4	Address

Figure 5. Multiple Object Misrepresentation example

The 'Route' class in Figure 5 calls for 4 individual instance variables. The novice author probably intended to provide a collection of objects relating to the addresses for this class.

This fault was only identified a small number of times, which in some ways was surprising. This may relate to the examples that students had seen in lectures and tutorials.

Table 7. Multiple Object Misrepresentation statistics

Total Multiple Object Misrepresentation faults	2
Total classes with faults	2 (0.3%)
Designs with faults	2 (1.1%)

5. CONCLUSION AND IMPLICATIONS FOR TEACHING

In this paper we have focused on observable faults in novice programmers' class diagrams.

If we examine the faults, we see that most common is the Non-Referenced Class fault. In other words students know that they need a class to model a concept but cannot figure out how to integrate the class into their designs. This is consistent with other researchers' findings. As Du Boulay [1] noted, the ability to see the program as a whole and understand its parts, and the relationship between them, is a skill which grows over time. When educators are introducing classes, it is reasonable at first to restrict the view to one class, and then introduce multiple instances, but the fact that classes interact *must* be demonstrated early enough so that students do not operate with an incorrect model.

The next most common faults were Misrepresentations. These boiled down to a failure to achieve high cohesion in class design. Again, experience helps here, but this fault might be somewhat mitigated if educators stress cohesion and coupling at an early stage. This concept is often postponed until a discussion of metrics in advanced level courses.

Interestingly, misconceptions discussed by Holland *et al.* [4] such as object/variable and object/class conflation and identity/attribute confusion were not much evidenced. It may be that the style of instruction that we have adopted has helped. Failure to use a collection was also quite low - perhaps our style of instruction has addressed that also.

The research reported on here was followed up by the creation of an interactive Case-Based tutoring tool, Vortex, which seeks to warn students when their designs exhibit these faults, and then suggests directions to find solutions. This work will be reported elsewhere, but is currently available in [14].

6. REFERENCES

[1] Du Boulay, B., Some Difficulties in Learning to Program, in *Studying the Novice Programmer*, Soloway, E. and Spohrer, J. C. (eds), Lawrence Eribaum, 1988, 283-299.

[2] Eckerdal, A., McCartney, R, Mostrom, J.E., Ratcliffe, M., and Zander, C., Can graduating students design software systems? in *Proceedings SIGCSE '06*, 2006.

[3] Garner, S., Haden, P. & Robins, A., My Program is correct but it doesn't run: A preliminary investigation of novice programmers' problems, in *Proceedings of Australasian Computing Education Conference*, 2005, 173-180.

[4] Holland, S., Griffiths, R. & Woodman, M., Avoiding Object Misconceptions, in *Proceedings SIGCSE '97,* 1997, 131-134.

[5] Hristova, M., Misra, A., Rutter, M., and Mercuri, R., Identifying and Correcting Java Programming Errors for Introductory Computer Science Students, in *Proceedings SIGCSE '03*, 2003.

[6] Lister, R., Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L., A Multi-National Study of Reading and Tracing Skills in Novice Programmers, *ACM SIGCSE Bulletin*, 36, 4 (Dec. 2004), 119-150.

[7] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., Laxer, C., Thomas, L. A., Utting, I., and Wilusz, T., A Multi National Study of Assessment of Programming Skills of First year CS students, *SIGCSE Bulletin*, 33, 4 (Dec. 2001), 125-140.

[8] Miller, G. A., WordNet a lexical database for the English language, http://wordnet.princeton.edu/, 2005.

[9] Muller, O., Pattern Oriented Instruction and Enhancement of Analogical Reasoning, *Proceedings ICER '05,* 2005.

[10] Or-Bach, R. and Lavy, I., Cognitive Activities of Abstraction in Object Orientation: An Empirical Study, *SIGCSE Bulletin* 36, 2 (2004), 82-86.

[11] Robins, A., Rountree, A.J., and Rountree, N., Learning and teaching programming: A review and discussion, *Computer Science Education*, 13, 2 (2003), 137 – 172.

[12] Soloway, E., Spohrer, J.C., and Littman, D., E unum pluribus: Generating Alternative Designs, in Mayer, R.E. (ed), *Teaching and Learning Computer Programming,* Lawrence Eribaum, 1988.

[13] Spohrer, J. C. & Soloway, E., Novice mistakes: Are the folk wisdoms correct? in *Studying the Novice Programmer*, Soloway, E. and Spohrer, J. C. (eds), Lawrence Eribaum, 1988, 401-416.

[14] Thomasson, B., *Identifying Faults and Misconceptions of Novice Programmers Learning Object Oriented Design*, PhD Thesis, University of Wales, Aberystwyth, 2005.