The 8th INternational Conference on Software Process Improvement - Research into Education and training, Quality in Teaching and Technology Based Learning, INSPIRE 2003, 23-25 April 2003, Glasgow, pp. 123-131

A Systematic Approach in Object-Oriented Design Heuristics Teaching

Ignatios Deligiannis[1], Ioannis Stamelos[2], Kerstin V. Siakas[1]

[1]Dept. of Information Technology, Technological Education Institution
P.O BOX 14561 GR-54101, Thessaloniki, Greece
Tel/Fax: (+30)2310 791295, e-mail: igndel@it.teithe.gr
Tel: (+30)2310 791296, e-mail: siaka@it.teithe.gr

[2]Department of Informatics, Aristotle University
54124 Thessaloniki, Greece
Tel:(+30)2310 998227, e-mail: stamelos@csd.auth.gr

Abstract

In the last few years, there has been growing enthusiasm, for object-oriented (O-O) approaches to information systems. There have been significant advancements in all areas of object-oriented information systems from programming to analysis, design and development. Considering the difficulty students face adopting the Object-Oriented (OO) concepts and techniques, this paper presents a systematic teaching approach aiming at applying efficiently and effectively these concepts. The model, based on OO design heuristics, and the most significant OO features, namely, abstraction, inheritance and composition, provides two teaching directions. First, is the *guidance role* of a number of appropriate design heuristics; Secondly, is the *assessment role* of a number of design heuristics in order to provide confirmation of their appropriate application on basic design structures. The proposed model is expected, first to guide the students on *how* to apply a specific OO feature at a given situation, and then to provide them with the ability to examine *whether* the chosen design alternative was the most appropriate.

Keywords
Object-Oriented, education, design, heuristic

The 8th INternational Conference on Software Process Improvement - Research into Education and training, Quality in Teaching and Technology Based Learning, INSPIRE 2003, 23-25 April 2003, Glasgow, pp. 123-131

1. Introduction

The OO paradigm has gained a broad acceptance during the last decade, mainly due to C++ and more recently to Java and UML. Based on abstraction, it intends to analyse and model real world problems, so that they can be implemented as software solutions, without losing the semantics of the original problem domain. Such a modeling approach raises the claims for higher productivity and increased quality in a number of significant quality factors, such as understandability, , maintainability and reusability.

Understandability expresses how well different components of the software is understood as requirements, design and dependencies between internal, external and shared components [1] . Maintainability is the ease with which a program can be corrected if an error is encountred, adapted if its environment changes or enhanced if the customer desires a change in requirements [1].Maintainability is usually measured indirectly by mean-time-to-change, which includes the time it takes to analyse a change request, design the modification, implement and test the change and finally distribute the change to all users. Reusability emphasizes creation and reuse of software building blocks [2]. Such building blocks usually called components should be catalogued for easy reference, standardized for easy application and validated for easy integration.

Nevertheless, some authors, providing empirical evidence, have expressed their doubts concerning claims accredited to OO technology about higher productivity and increased quality, in particular maintainability and reusability. Hatton [3], in a case study about corrective maintenance issues, indicated a number of concerns about OO technology having met its objectives. Jones [4], has identified a lack of empirical evidence to support defect removal efficiency and reusability.
Defect Removal Efficiency (DRE) is a quality metric that provides benefit at both project and process level.

However, others have shown positive effects concerning quality factors. Lewis [5], in his experimental study, has indicated that the OO paradigm is particularly suited to reuse, demonstrating a particular affinity to the reuse process. In a study by Briand *et al* [6] examining the quality factor maintainability, it was shown that the OO techniques are sensitive to bad and good design practices, and that OO design principles significantly affect the design quality. Also, in two empirical studies performed by the authors of this paper, examining the impact of a design heuristic on maintainability of OO designs, it was found that a specific design heuristic, dealing with the "god class problem" significantly affects maintainability as well as the design quality [7] [8].

The objective of this paper is to propose a systematic teaching approach, based on OO design heuristics, of the most significant OO concept and mechanisms, such as abstraction, inheritance and composition. The paper is structured as follows. Section 2 describes background and related work. Section 3 and 4 describes respectively, a

systematic teaching approach and the education model, proposed by this paper. Finally, section 5, presents the conclusions and future research aims.

2. Background

The need to investigate claims about OO success is growing. However, in a review, examining the way experimentation was carried out, it was identified that in the majority of the experiments, students were used as subjects, posing doubts concerning their performance ability [9]. It is well known that the OO paradigm offers several mechanisms and tradeoffs where decisions on best alternatives are usually fuzzy and mostly based on expert judgment. In other words, cumulative knowledge is likely to play a very important role in the design phase. Thus, it is questionable whether novice designers performing, usually with cases, with limited time, are always the most appropriate subjects. The difficulty of learning OO concepts for programmers new to OO techniques and particularly for those having previously been exposed to traditional structured methods, is well documented [10-12].

In order to speed up *students'* familiarity with OO concepts, we believe emphasis should be placed on two aspects: First, OO design heuristics, concentrating cumulative and distilled knowledge based on experience, could improve students' ability both in understanding and successfully applying the OO mechanisms and concepts. Secondly, there is a plethora of OO design heuristics found in the literature. Students of OO software engineering courses could benefit from the systematic teaching of design heuristics. Furthermore, the importance of education in OO concepts has been identified by Sheetz [13] in his experimental investigation examining the difficulties OO practitioners of three levels of experience (novices, intermediates and experts) face in software development, when applying OO techniques and concepts. Here, "the core of meaning units" shared by participants at all levels of experience, and the relationships common to both novices and experts, provide the potential for the greatest return on education resources. The study concludes confirming that organizations adopting OO techniques must provide education programs across a range of topics.

On the other hand, heuristics are defined as 'rules of thumb', referring to the accumulated and distilled knowledge gained from the experience [14]. Since they form pieces of advice on detailed design aspects, claimed to be true in most cases, they can guide the designers in choosing between various alternatives. Related to a multitude of software aspects, they are aimed at enhancing software quality [14-17]. Furthermore, *design heuristics* provide an additional means for design assessment [18]. In addition, when violating heuristics, there is a risk of leading to complex and monolithic design structures.

In a few empirical studies, where the impact of heuristics was examined mainly on understandability and maintainability, it has been shown that considerable advantages could be gained when adhering to their guidance [6-8]. However, there is only one study

related to the education of OO design heuristics, where the authors collected a number of heuristics and embedded them into an educational tool in order to automate their use [19].

3. A systematic teaching approach

As mentioned earlier, the objective of this paper is to provide a teaching approach of a few but important OO design aspects, based on OO design heuristics. Hence, it is proposing two roles of heuristics' use: First, it is the *guidance role* they might provide in order to most effectively apply the OO mechanisms under consideration [20]. Secondly, it is the *assessment role* they might provide in order to confirm their appropriate application, based on objective assessment [18]. This is a clear improvement with respect to the most relevant approach we found [19]. According to Kirsopp and Shepperd [18] OO design heuristics, apart from the guidance role they provide to designers, they could also effectively serve as an assessment mechanism between different design strategies. They distinguish two types of assessment, *stand-alone*, in which a single version of a single entity is evaluated in isolation, and the *comparative* one. The first one is further distinguished in the *threshold-based* heuristics, and the *rule-based* heuristics. They are described as follows:

a) In the *threshold-based assessment,* each heuristic specifies a single value. An example of a heuristic type is suggested by Coad & Yourdon [21], *'Avoid having too many services per class. Each class typically has no more than six or seven public interfaces*. In this case the heuristic contains its own threshold value;

b) In the *rule-based assessment,* the heuristics embody rules. An example of this type of heuristic is suggested by Firesmith [16]: *'subclasses should not delete features of their superclasses'*;

c) In the *comparative assessment,* another entity or another version of the same entity is needed as the reference point for the assessment. An example of a heuristic of this type is: *'Avoid having a large, central object which controls most of the functionality'* [15], implying that relatively small and decentralized objects are preferable. This type of heuristic is more likely to be useful in discriminating between designs.

Thus, these two mutually complementary roles, may guarantee the appropriate application of the OO features: abstraction, inheritance, and composition, briefly described in the following.

Software design is considered to be the skeleton of a software system. Consequently, its quality significantly impacts the quality of the final products. An "ideal" OO design is one that does not distort the conceptual reality of the domain. Success in this, as it is argued, produces maintainable systems, because such models tend to be relatively easy to understand, and therefore relatively easy to modify [22].

However, teaching the conceptualization of real-world problems, as key abstractions, within the context of OO analysis and design remains an ongoing difficulty. A flawed

abstraction does not closely resemble the original problem at both a semantic and structural level, leading to tedious and error-prone software products [23]. On the other side, there are two basic mechanisms for extending a design, namely *class inheritance* and *object composition*. They are further the most common techniques for reusing functionality in OO systems. Hence, they must be carefully applied since they can be dangerous when used incorrectly [15].

In the next we describe an education model using these three major design aspects. Based on a number of the most significant design heuristics proposed over recent years, the model provides first a guidance module and secondly an assessment module.

Teaching with the model provided the opportunity to perform a formal experiment, using the students as informed subjects. The significance of a single heuristic, called "god class problem" (AA3 in Table 1) and dealing with poorly distributed system design intelligence, as well as its impact on the quality factor maintainability, was examined in a controlled experimental study [8]. It was found that violating the specific design heuristic, the participants were forced to misuse the inheritance mechanism in their provided solutions, thus additionally violating one of the three design heuristics presented in Table 1, namely IA1 to IA3.

Each design aspect, followed by a number of the appropriate design heuristics, is described in the following section.

4. The education model

Next, each of the investigated OO features is described, followed by some related empirical findings in order to further support our motivation for their inclusion to the proposed model.

Abstraction. According to Whitmire [25], an abstraction is an element in the domain model that represents all or part of a concrete or conceptual object in the domain model. The primary purpose of OO analysis is to discover the essential abstractions in the problem domain, while in the OO design is to implement these essential abstractions correctly and efficiently.

Cockburn [26] considers abstraction as a major factor in predicting the robustness of the design. A design property that most closely captures the abstraction feature is cohesion, thus providing an efficient way of assessing it objectively. Cohesion is an attribute of design rather than code that can be used to predict reusability, maintainability and changeability [27]. Its contribution to maintainability was empirically supported by the findings of an empirical study [7]. A number of proposed in the literature metrics are aimed at quantifying cohesion assessment [28].

Inheritance is a class-based relationship used to capture the 'kind-of' relationship between classes. Its main purposes are twofold: it acts as a mechanism for expressing commonality between two classes (generalisation), and it is used to specify that one class is a special type of another (specialisation). In practice, it is just a mechanism for extending an application's functionality by reusing functionality in parent classes [29]. It is argued that inheritance should be utilized to model commonality and specialization [30] [20] [16]. Inheritance can also be used for sub-typing, when substitutability is guaranteed [29].

The importance of the specific OO mechanism was the key motivator to its empirical investigation of a considerable number of controlled experiments [9]. Furthermore, the findings of a formal experiment, showed that it can effectively affect the quality factors maintainability and reusability [31].

Composition is a kind of whole-part association. It forms object-based relationships needed to model complex (part-of) hierarchies of objects. In composition the whole strongly owns its parts (e.g., an Engine is part of a Car), implying that the lifetime of the 'part' is controlled by the 'whole'. This control may be direct or transitive [32].

Based on the guidance module of the proposed model (Table 1), students may study the concept underlying a design heuristic and develop their ability to avoid wrong design decisions. Moreover, based on the assessment module they can easily evaluate their own design decisions. It was observed that such a mental process greatly assisted the students to develop analysis and design skills.

5. Conclusions

Conceptualizing real-world problems, within the context of OO analysis and design is not an easy task. This is particularly true for students and practitioners, novices to OO concepts and techniques, when challenged to choose the most appropriate among alternative design solutions. The OO design heuristics capturing accumulated and distilled knowledge gained from the experience provides valuable consultation on such decisions. The paper argued about the benefits of transferring this knowledge to a teaching environment and process. The paper suggested a teaching approach and a heuristic model, in order to easier understand and faster and more appropriately apply the significant OO concepts. This model, concentrated on the most important and often used OO concepts, is also aiming to strengthen students' confidence by providing an assessment means based on heuristics.

Further research planned on the basis of this approach includes an empirical investigation, examining the effects of the suggested education model on students' understanding and performance, within a formal experimentation environment.

Table 1. The education model (Guidance heuristics in light gray are codified as XGn. Assessment heuristics in deep gray are codified as XAn. "X" = OO concept)

Heuristic code	Abstraction
AG1	A class should capture one and only one key abstraction [13].
AG2	Keep related properties and behavior in one class [27]
AG3	Avoid 'fuzzy' class definitions. A class should be cohesive with a single, well defined and clearly bounded purpose [19]
AA1	All data (no more than six data members) should be hidden (private) within its class [27].
AA2	Avoid having too many services per class. Each class typically has no more than six or seven public interfaces [19]
AA3	Avoid having a large, central object which controls most of the functionality [13] [5].

	Inheritance
IG1	Inheritance should only be used for sub-typing (as opposed to implementation inheritance) [28]
IG2	Inheritance should be used to model generalization - specialization ("a-kind-of" taxonomies) relationships [29]
IA1	Only use inheritance when you want to inherit all the properties of a subclass, not just some of them [30]
IA2	"Subclasses should not delete features of their super classes. Subclasses that delete features probably are not specializations of their superclass(es) because they cannot do something that their parent(s) can ". [14]
IA3	It should be illegal for a derived class to override a base class method with a NOP method, i.e. a method which does nothing [27].

	Composition
CG1	Aggregation (composition) should not be used to represent non-compositional relationships, i.e., spatial/temporal inclusion, attribution, membership, attachment, ownership [26]
CA1	Containment classes should know what they contain, but contained classes should have no knowledge of who contains them [27].
CA2	Objects which share lexical scope, i.e. those contained in the same containing class, should not have uses relationships between them [13].

References

1. Pressman Roger S., *Software Engineering, A Practioner's Approach,*. European Adaption, ed. Fifth edition. 2000: Darrel Ince.
2. Hooper J.W.E. and C. R.O, *Software Reuse: Guidelines and Methods*. 1991: Plenum Press.
3. Hatton, L., *Does OO Sync with How We Think?* IEEE Software, 1998(May/June): p. 46-54.
4. Jones, G., *Gaps in the object-oriented paradigm*, in *IEEE Computer*. 1994. p. 90-91.
5. Lewis, J., et al., *An Empirical Study of the Object-Oriented Paradigm and Software Reuse.* OOPSLA '91, 1991: p. 184 - 196.
6. Briand, L., C. Bunse, and J. Daly, *A Controlled Experiment for Evaluating Quality Guidelines on The Maintainability of Object-Oriented Designs.* IEEE Trans. on Softw. Eng., 2001. **27**(6): p. 513-530.
7. Deligiannis, I., et al., *An empirical investigation of Object-Oriented Design Heuristics for Maintainability.* Journal of Systems and Software, 2002.
8. Deligiannis, I., et al. *A Controlled Experiment Investigation of An Object-Oriented Design Heuristic for Maintainability*. 2002.
9. Deligiannis, I., et al., *A Review of Experimental Investigations into Object-Oriented Technology.* Empirical Software Engineering Journal, 2002. **7**(3): p. 193-231.
10. Hillegersberg, J., K. Kuman, and R. Welke, *An empirical analysis of the performance and strategies of programmers new to object-oriented techniques*, in *Psychology of Programming Interest Group: 7th Workshop*. 1995.
11. Chandel, J. and S. Hand, *Object-Oriented Concepts: Where Do People Go Wrong?* Object Manager, 1994. **10**: p. 17-19.
12. Cohen, J., et al. *PANEL: Training Professionals in Object Technology*. in *OOPSLA'94*. 1994.
13. Sheetz, S.D., *Identifying the difficulties of object-oriented development.* J. of Systems and Software, 2002. **to appear**.
14. Booch, G., *Rules of Thumb.* ROAD, 1995. **2**(4): p. 2-3.
15. Riel, A., *Object-Oriented Design Heuristics*, ed. Addison-Wesley. 1996.
16. Firesmith, D., *Inheritance guidelines.* JOOP, 1995(May): p. 67-72.
17. Lorenz, M. and J. Kidd, *Object-Oriented Software Metrics*. Object-Oriented Series. 1994: Prentice Hall. 146.
18. Kirsopp, C. and M. Shepperd. *Using heuristics to assess object-oriented design quality*. in *5th Int. Conf. on Empirical Assessment & Evaluation in Softw. Eng.,*. 2001. Keele University, Staffordshire, UK.
19. Gibbon, C. and C. Higgins, *Teaching Object-Oriented Design with Heurustics.* ACM Sigplan Language Tips.

20. Capretz, L. and P. Lee, *Object-oriented design: guidelines and techniques.* Information and Software Technology, 1993. **35**(April): p. 195-206.
21. Coad, P. and E. Yourdon, *Object-Oriented Design*. first ed. ed. 1991: Prentice-Hall.
22. Pooley, R. and P. Stevens, *Using UML: Software Engineering with Objects and Components*, ed. O.T. Series. 1999: Addison-Wesley.
23. Rumbaugh, M., et al., *Object-Oriented Modeling and Design*. 1991: Prentice-Hall.
24. Bennet, S., S. McRobb, and R. Farmer, *Object-Oriented Systems Analysis and Design using UML*. 1999: McGraw-Hill.
25. Whitmire, S., *Object Oriented Design Measurement.* 1997: John Wiley & sons.
26. Cockburn, A., *The Coffee Machine Design Problem: Part 1 & 2.* C/C++ User's Journal, 1998(May/June).
27. Yourdon, E. and L. Constantine, *Structured Design*. 1979, Englewood Cliffs, N.J.: Prentice-Hall.
28. Briand, L., J. Daly, and J. Wust, *A Unified Framework for Cohesion Measurement in Object-Oriented Systems (1997).* Empirical Software Engineering: An International Journal, 1997.
29. Amstrong, J. and R. Mitchell, *Uses and abuses of inheritance.* Softw. Eng. J., 1994(Jan.): p. 19-26.
30. Lieberherr, K. and A. Riel. *Contributions to Teaching Object-Oriented Design and Programming*. in *OOPSLA*. 1989.
31. Deligiannis, I., et al., *A Controlled Experiment Investigation of an Object-Oriented Design Heuristic for Maintainability.* J. of Systems and Software, 2003. **accepted**.
32. Civelo, F. *Roles for composite objects in object-oriented analysis and design*. in *OOPSLA '93*. 1993.
33. Riel, A. *Introducing to Object-Oriented Design Heuristics*. in *OOPSLA'94*. 1994. Portland, Oregon, USA.
34. Bar-David, T., *Practical consequences of formal definitions of inheritance.* Journal of Object Oriented Programming, 1992(July/August): p. 43-49.
35. McGrecor, J.D. and T. Korson, *Supporting dimensions of classification in object-oriented design.* JOOP, 1993(Feb): p. 25-30.
36. Rumbaugh, J., *Disinherited! Examples of misuse of inheritance.* JOOP, 1993(Jan): p. 19-24.

The 8th INternational Conference on Software Process Improvement - Research into Education and training, Quality in Teaching and Technology Based Learning, INSPIRE 2003, 23-25 April 2003, Glasgow, pp. 123-131