Improving the Teaching of Object-Oriented Design Knowledge

Javier Garzás
Kybele Consulting and Kybele research
Rey Juan Carlos University
Dpto. De Lenguajes II
Univeridad rey Juan Carlos
Madrid, Spain
Javier.Garzas@kybeleconsulting.com

Mario Piattini
Alarcos Research Group.
Escuela Superior de Informática - University of Castilla-La Mancha.
Ronda de Calatrava, s/n. 13071, Ciudad Real, Spain
Mario.Piattini@uclm.es

Abstract: In the general sphere of the teaching of software engineering concepts, it can be noted that there are very few pieces of work dealing with how to get across, through teaching, the practical experience that has been built up on the subject of object-oriented design. The few works that do exist focus on design patterns. Pattern catalogues, however, do not completely resolve the problem of imparting the experience about object-oriented design, an area where it is clear that the greatest benefit derived from the patterns is achieved when their designers are already-experienced. What is more, other elements associated with object-oriented knowledge, such as principles, heuristics, best practices, bad smells, etc., are components related to practical knowledge of design. These are barely taken into consideration, however. In an effort to solve these problems, we put forward an ontology which brings together and integrates object-oriented design which improves teaching, amongst other things. It makes the great quantity of knowledge that has been built up clearer and brings it together into a united whole. It is thus possible to create catalogues of integrated knowledge.

Categories and Subject Descriptors: D.1.5 [Programming Techniques]: Object-oriented programming; D.2.2 [Software Engineering] Design tools and techniques---Object-oriented design tools
General Terms: Design
Keywords: Teaching, Object-Oriented, Design, Knowledge

1. MOTIVATION

In recent times the software engineering community has paid a great deal of attention to the question of how to organize its knowledge, as well as on how to go about converting this into a syllabus plan. Addressing this issue, [1] comments on how a study plan dealing with concepts related to software engineering should include five complementary elements.

- Principles: Lasting concepts about a particular area.
- Practices: problem-solving techniques which those working in the field apply regularly and consciously.
- Applications: areas of specialization, where principles and practices are best expressed, in the most explicit way.
- Tools: An up-to-date appraisal of the products which aid in the application of principles and practices.
- Mathematics: The formal basis on which all that has been explained above is based.

In the specific field of object-oriented design (OOD), the elements we have mentioned above are just as important in defining a syllabus which will be propitious for teaching this subject. We are aware that OOD is one of the key elements for those being taught software engineering. Thus the relevance of all we say in this paper should be underlined. In fact the empirical study by [2] states that software engineering students themselves suggest that one way in which educational establishments could improve their services would be to offer courses on design. In this study the students see design as the greatest challenge in the quest for programmers to be efficient. In a similar vein, [3] comments that there are three main ideas which should be taken into account when teaching about OO:

- Show the overall context, instead of just the programming or structures of simple designs.
- Show how, starting from the requirements, one can get to the design and eventually to programming.
- Show the techniques which make it possible to detect when the models are good ones.

Nevertheless, despite the fact that it is clear that OOD teaching plays a highly important part and that the main ideas on how to teach it have been identified, a complete way of linking the great amount of practical knowledge on OOD ("principles and practices", to use the terminology of [1]) with in-class learning has not really been created. In the study carried out by [2] we can also note that in answering the question "How did you learn object technology?", some 89% of those surveyed replied that they were self-taught. What this suggests, according to [2],is that either OO is not seen as high priority or that the training received by students is not the most suitable.

In the field of OOD, patterns are the most popular element when using accumulated knowledge. In the last few years they have been built in as components of many plans of study in software engineering. Writers such as [4] remark that teaching design patterns is a challenging task, where it is not only necessary to give instruction about the structure of the solution offered by a design pattern. It is also vital that the student should understand when and where a pattern can be applied. We know that although patterns are of great importance, they do not fully resolve the problem of imparting practical experience in OOD. So [5] reminds us of how experience gained from the incorporation of patterns in education syllabuses has demonstrated two things. These conclusions are firstly that the isolated presentation of a pattern and its format is not effective in the case of students who are faced with patterns for the first time and secondly that these students do not indeed see the value of a pattern.

It is on similar lines that [6] make the point that when patterns are learnt only from focusing on them as pattern only tells us what to do, but not when to do it, or why.

There is yet another issue in relation to what we have just outlined. It is that, even if we pass over the problems which the teacher faces when trying to integrate patterns into the teaching of OOD, we hit up against another sizeable difficulty: knowledge about OOD consists of much more than just knowledge about patterns. It is made up of many other components such as principles, best practices, refactorings, etc. These are all elements which offer information based on experience, but which are at the same time diffuse and lacking in clear definition. The problems they present affect their application in practice and thus the teaching of them in OOD.

There are very few pieces of work which touch on the problem of how to teach the student all the knowledge that has been gathered in OOD. Amongst the small number that exist we might highlight that of [7], who tell us how they use design heuristics to give support to teaching methods, with the following objectives:

- To give more thrust to the spreading and sharing of experience in design.
- To provide the student with a vision of what a good design is.
- To create awareness of software quality, in aspects such as maintainability, re-use and so on.
- To provide an educational framework
- To capture experience in an objective way.

Educators do, however, often find themselves discouraged in the job of teaching OOD concepts, especially when the students are not ready to learn them, in many instances [8]. There is, therefore, an evident need to bring to bear some mechanisms which allow students to apply patterns, along with the rest of the elements that are associated with OOD knowledge. We need a way of ensuring that students have mastered the basic concepts of OOD and that they have evolved with these as their starting point, to go on from there to be able to create real world designs that are complex, scalable and reusable.

With a view to solving the problems set out above, we propose an ontology on OOD, as well as a catalogue which brings together elements of knowledge in OOD such as principles, heuristics, best practices, etc. The use of an ontology improves the teacher's preparation and his or her ability to get across concepts which many teachers are very familiar with in the field of OOD but which they find difficult to convey to the student. [9].

2. ONTOLOGY FOR THE IMPROVING OF TEACHING KNOWLEDGE IN OOD

An ontology supposes a common vocabulary for those who need to share information in a given domain.[10]. The use of an ontology helps to (1) share a common understanding of information, (2) re-use knowledge, (3) make assumptions starting from a more explicit basis of knowledge, (4) separate domain knowledge from operational knowledge, and (5) analyse the domain knowledge.

When constructing an ontology that integrates and inter-relates OOD knowledge, we can begin by observing how the components of that knowledge may be organised in groups, thanks to similarities that exist between them. So, in spite of the fact that there are many associated OOD terms (patterns, principles heuristics, bad smells, best practices, etc), we have been able to observe that they may all be grouped together in two sets- declarative and operative.

The declarative elements are concepts that describe how to face a given problem: components such as heuristics, patterns, bad smells etc, are found in this group. Within this set of declarative elements, however, different sub-groups can be noted. So, for example, pattern catalogues such as those of [11] are applicable to all projects.

But in real projects we will need to use other elements of knowledge, and we will require, for example, technological knowledge (patterns for J2EE or .Net principles) or knowledge associated with a specific domain (real time, integration, etc.).Thus we see how declarative knowledge may be made up of three sub-categories: general, technological and domain knowledge.

On the other hand, despite the existence of a large amount of terminology and irrespective of patterns, it is obvious that elements such as heuristics, principles, bad smells, best practices and such like have a common structure which coincides with that of a rule. That is to say, they offer a recommendation depending on the fulfilling of a condition. We can then group all these elements of knowledge together, under the term ”rule”. It should be underlined that rules are different from patterns, since rules are based on natural language which is always more ambiguous, while patterns are more formalised and their description is always broader in scope.

To sum up on this point, elements which are declarative in character can be divided into three groups: general, technological and domain. These in turn are each divided into rules and patterns (see Fig. 1 at the end of this article where the ontology is seen in UML notation.)

Apart from the declarative knowledge we also find operative elements, which are elements that build up knowledge with respect to operations or processes so as to carry out changes in a design. Refactorings fit into this group directly. (These can be specified as parameterized program transformations which at the same time preserve the functionality of the program [12]).

There is another aspect to bear in mind, which is that there are relationships between knowledge elements. It is complicated to see these without a clear division and classification of the knowledge elements, but once they are classified it can be seen that:

- o Refactorings build up knowledge about how to introduce design elements systematically. The elements of declarative knowledge (rules and patterns) are incorporated into the design by operative knowledge (Refactorings). Thus we can affirm that declarative knowledge is introduced by declarative knowledge.
- o Between each rule-pattern there are two types of relationships: rules imply the use of patterns and patterns obey rules.
- o All pattern entities (general, domain or technological), have a reflexive relationship in which applying one pattern may imply the use of another.
- o Operative knowledge entities have reflexive composition relationships. In other words, one knowledge element is made up of others.

Lastly, we should take into account that there is, associated with each one of the relationships between knowledge entities, cardinality which defines the numerical limits of the relationship. (See Fig.1).

3. A CATALOGUE OF GENERAL RULES FOR OOD

This section follows the lines suggested by the ontology presented in this paper, setting out the idea of having integrated knowledge catalogues available to us, and highlighting just how important that is. Special attention is given to a catalogue of general rules whose aim is to aid in the application and teaching of OOD, since these rules form an important part of this subject.

In describing each of the rules, the catalogue takes as its starting point the sections which the pattern catalogue of [11] uses when describing a pattern and then generalises these sections, setting them out in detail. The relationship between rules and patterns has been put into the following categories:

- o Implies the use of [Patterns]: whenever it applies, patterns which are necessary in the design resulting from the application of a rule.
- o It is introduced by [Refactorings]: whenever it applies, refactorings or operations which introduce a rule into a design.

Another important characteristic of the catalogue is that it identifies each rule with a name to label it meaningfully. Care has been taken when choosing this name, the aim being to help the student to see clearly and to identify, as quickly as possible, where and when a rule may be applied. The name selected depends on the condition which triggers the rule. The present version of the catalogue is made up of 20 rules- this number is by no means a fixed quantity which will never experience any changes, of course. The current list of rules is as follows:

- o Rule for if there are dependencies of concrete classes.
- o Rule for if an object behaves differently according to its state.
- o Rule for if a class hierarchy has many levels.
- o Rule for if something is used very little or not used at all.
- o Rule for if a super class knows one of its sub-classes.
- o Rule for if a class collaborates with many.
- o Rule for if a change in an interface has an impact on many clients.
- o Rule for if there is no abstraction between an interface and its implementation.
- o Rule for if a super-class is concrete.
- o Rule for if a service has a lot of parameters.
- o Rule for if a class is large.
- o Rule for if elements of the user interface are within domain entities.
- o Rule for if a class more things from another class than from itself.
- o Rule for if a class rejects something it has inherited.
- o Rule for if attributes of a class are public or protected.

By way of example, in Table 1 the rule of “If between the Interface and its Implementation there is no Abstraction” is displayed.

Rule for IF there is no abstraction between an Interface and its Implementation

Purpose

We should have default implementations.

Also known as

No other names are known for this Rule.

Motivation

No references about this Rule have been found.

When designing, types or interfaces appear, in other words, special types which represent a sub-set of operations to which a subclass may respond without specifying how. In many cases, these interfaces may appear, for example, by the application of "Rule for IF there are dependencies of Concrete Classes". The difference between an interface and an abstract class is that the former does not have any service with implementation.

If an abstraction or implementation is not introduced between an interface and its implementation, it will often happen that, by default, each one of the classes which implement this interface will have to have this implementation. On many occasions, this will be the same for each of the sub-classes; the code will therefore be duplicated (one of the worst things that can appear in a system).

On other occasions service interfaces appear (the Subject interface in an Observer pattern [11], for instance). These may implement domain classes. In such a case it is not so clear that a domain class should have to implement the operations that the interface defines. This is usually resolved by putting in, between the interface and the class which implements it, an abstract class which joins defect implementation by default to the majority of interface operations.

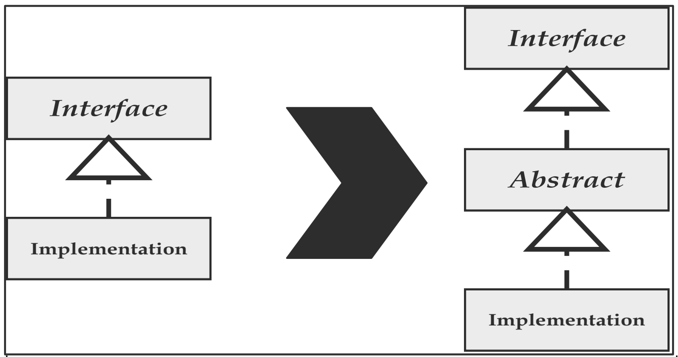
Taking into account all that has been explained above, we could think of eliminating the interface and of leaving only the abstract class. This might make future sub-classes of this abstract class not want this type of default behaviour, however. Overwriting it is not a good solution, moreover (see Rule for If a Class Rejects something of what it inherits).

Recommendation

IF between an Interface and its Implementation there is no abstraction THEN create an abstract class with a default implementation between the interface and the class that implements it.

Applicability

Use this rule when there is no Default Abstraction between the Interface and its Implementation.



Structure

Table 1. Rule for IF there is no abstraction between an Interface and its Implementation

Participants

Not Applicable.

Collaborations

Not Applicable.

Consequences

This rule has the following consequences: (1) It rules out the possibility of a duplicated code occurring, since all subclasses have a default implementation and there is therefore no need to create one for each class. (2) It avoids concrete classes implementing services that do not really fall within their responsibility.

Known Uses

This rule can be seen in many patterns, frameworks and software systems.

Implies the use of [Patrones]

Not Applicable.

Is introduced by refactorings

As the catalogue of [13] says, this Rule can be introduced mainly with the following refactorings: Extract Interface, Extract Subclass, Extract Superclass, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method.

4. CONCLUSIONS

Very few pieces of work deal with how to teach the accumulated practical experience in OOD. Our search in this area has revealed numerous articles on teaching object-orientation which generally introduce the students only to programming concepts. There is, however, an overall tendency to overlook OOD and it is in this sense that these works are most lacking. What is more, the few pieces of work that do touch on the teaching of experience in OOD focus on the pattern element. As many authors agree, however, the concept of pattern does not solve all the problems of design, nor is it the only element associated with knowledge.

Catalogues such as that of [11] are generally a major reference for designers who already have experience [5]. A designer who is already familiar with certain patterns will immediately be able to apply them in solving problems.[8].

Other elements must, then, be taken into account. These would be such elements as principles, heuristics, best practices, bad smells, etc. But at present these elements are is a state in which they are hard to apply.

To solve these problems and to systematise the teaching of practical concepts, we have produced ontology for OOD. The ontologies can be applied in a wide variety of contexts, for various contexts. In the field of teaching, these can improve communication between people. [14]. As a rule, information on design concepts is expressed using a vocabulary which is not very familiar and even in a format which is not too accessible. Where this happens, ontology provides a unified approach with a common terminology and integrated knowledge in a given domain. The ontology of OOD makes it possible to create integrated knowledge catalogues, at the same time providing a model for the teacher's own preparation. These catalogues are thus able to convert OOD knowledge into teaching units.

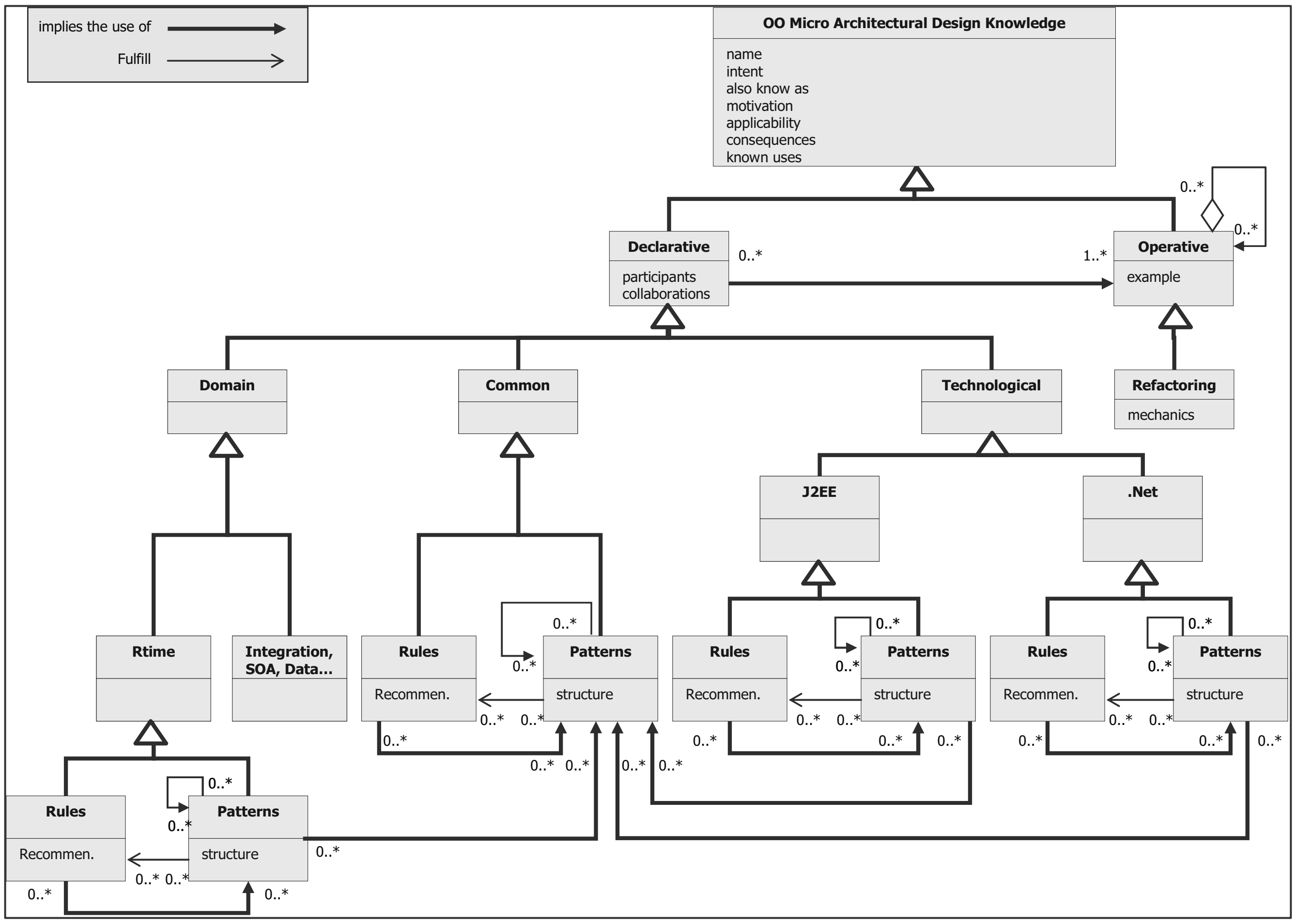
5. ACKNOWLEDGEMENTS

This research is partially supported by the MAS project of the General Research Council (Dirección General de Investigación) of the Spanish Ministry of Education and Science (TIC 2003-02737-C02-02) and ENIGMAS (Entorno Inteligente para la Gestión del Mantenimiento Avanzado del Software), supported by the Department of Education and Science of the Junta de Comunidades de Castilla-La Mancha (Regional Government of Castile-La Mancha) (PBI-05-058).

REFERENCES

[1] Meyer, B., Software engineering in the academy. Computer, 2001: p. 28-35.
[2] Mosley, P., A Taxonomy for learning object technology 2004.
[3] Smialek, M., Teaching OOAD with active lectures and brainstorms, in OOPSLA 2000 2000.
[4] Sendall, S. Gauging the Quality of Examples for Teaching Design Patterns. in "Killer Examples" for Design Patterns and Objects First Workshop, OOPSLA (www.cse.buffalo.edu/alphonce/OOPSLA2002/KillerExamples). 2002. Seattle.
[5] Warren, I. Teaching Patterns and Software Design. in Proc. Seventh Australasian Computing Education Conference (ACE2005). 2005. Newcastle, Australia: ACS.
[6] Shalloway, A. and J. Trott, Design Patterns Explained: A New Perspective on Object-Oriented Design. 1st ed. 2001: Addison-Wesley Professional.
[7] Gibbon, C. and C. Higgins, Teaching object-oriented design with heuristics. ACM SIGPLAN, 1996. 31(7): p. 12 - 16.
[8] Lewis, T.L. and M.B. Rosson. A measure of design readiness: using patterns to facilitate teaching introductory object-oriented design. in 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA). 2002. Seattle, Washington
[9] Veal, W. and J. MaKinster, Pedagogical Content Knowledge Taxonomies. Electronic Journal of Science Education, 1999. 3(4).
[10] Noy, N.F. and D.L. McGuinness, Ontology Development 101: A Guide to Creating Your First Ontology, S.K.S.L.T.R.K.-.-a.S.M. Informatics, M.A.A.o.a. Technical Report SMI-2001-0880, and http://www.ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html, Editors. 2000.
[11] Gamma, E., et al., Design Patterns. 1995: Addison-Wesley Professional.
[12] Opdyke, W., Refactoring Object Oriented Frameworks, in Computer Science. 1992, Illinois: Urbana-Champain.
[13] Fowler, M., et al., Refactoring: Improving the Design of Existing Code. 1st edition ed. 2000: Addison-Wesley Professional.
[14] Jasper, R. and M. Uschold. A Framework for Understanding and Classifying Ontology Applications. in Twelfth Workshop on Knowledge Acquisition Modeling and Management KAW'99. 1999. Canada.

Fig 1. Ontology of Knowledge in OOD.


implies the use of
Fulfill
OO Micro Architectural Design Knowledge
name
intent
also know as
motivation
applicability
consequences
known uses
Declarative
participants
collaborations
Operative
example
Domain
Common
Technological
Refactoring
mechanics
J2EE
.Net
Rtime
Integration, SOA, Data...
Rules
Recommen.
Patterns
structure
0..*
1..*