Teaching and Learning Object-oriented Programming Principles: Process and System

Sundaram Subramanian, PhD
Manukau Institute of Technology
Auckland, New Zealand
ssubrama@manukau.ac.nz

M Daud Ahmed, PhD
Manukau Institute of Technology
Auckland, New Zealand
daud.ahmed@manukau.ac.nz

Abstract

Object-oriented programming is one of the most influential programming paradigms and has been widely used for system development during the last decade. Many universities, polytechnics and IT training organisations teach this programming methodology and technique somewhere in their curriculum. Teaching the subject however, remains difficult. This paper reviews some of the best practices and identifies the fundamental difficulties of teaching and learning programming principles and techniques. It then proposes a process that illustrates a teaching and learning technique for object-oriented programming principles that is supported by a tool. Both the process and the tool have been applied, tested and validated in a tertiary educational institute that consistently improves students' learning outcomes and teaching experiences.

Keywords

OOAD, object orientation, analysis and design, OO principles, learning and teaching

This quality assured paper appeared at the 2nd annual conference of Computing and Information Technology Research and Education New Zealand (CITRENZ2011) incorporating the *24th Annual Conference of the National Advisory Committee on Computing Qualifications*, Rotorua, New Zealand, July 6-8. Samuel Mann and Michael Verhaart (Eds).

Introduction

This paper proposes a process that illustrates a teaching and learning technique for object-oriented (OO) programming principles and also designs and develops a unique teaching and learning tool to support the process. Both the process and the tool have been applied, tested and validated in a tertiary educational institute that consistently improved students' learning outcomes and teaching experiences. One of the most difficult problems in teaching object-oriented programming is the learners' unlearning of the global knowledge of control that is possible with procedural systems, and relying on the local knowledge of objects to accomplish tasks (Beck & Cunningham, 1989).This paper also identifies the benefits of teaching OO programming as the first programming course, and the challenges faced by the teachers struggled in explaining complex concepts to the beginners. Teaching OO principles can be difficult without right teaching tools. Rosenberg and Kölling (1997) and also Kölling (1999) suggest that OO programming does not have to be complex, given the appropriate tools. They also observe that visualisation tools for class structures allow interaction and experimentation with objects, which help grounding the learners to the first principles of OO.

The reasons attributing to the difficulties are: firstly, most textbooks maintain a procedural view of programming (Bennedsen & Caspersen, 2004); secondly, introductory programming texts have a syntax-driven organization of the material where the focus is primarily on the programming language (Bennedsen & Caspersen, 2004); and finally, teachers, due to limited choice of programming languages for an introductory programming course, use one of these languages to teach the principles of object orientation (Kölling, 1999).

Use of Bloom's taxonomy, where a gradual exposure to complexity and structure (*Knowledge* > *Comprehension* > *Application* > *Analysis* > *Synthesis* > *Evaluation*) was backed up by a model-driven approach to teaching object-oriented programming, helped reduce dropout rates in one university course, from 48% to 11% (Bennedsen & Caspersen, 2004).

Selection of language for introductory programming course is the key question for educators (Johansson, Ohlsson, & Molin, 1995; Kölling, 1999). A survey of languages commonly used for teaching object-orientation reveals various type of problems (Kölling, 1999). They differ in nature and scale for each of these languages. The factors that need to be considered are pedagogical, such as the kind of mental image the language encourages; and practical, such as the tools the students may use for the language on the installed hardware platform and within the budget allowed. Typically, these concerns were in conflict, and the best solution therefore would be a compromise. Johansson, Ohlsson and Molin (1995) recommend that basic concepts such as data-types, simple statements, and loop constructs are covered in just a few lectures before moving straight on to modelling sessions. For the practical sessions, a group dynamic approach is employed with the help of patterns to express guidelines, standard techniques and useful practical tips about how to achieve a good design. Patterns are found as a way to express not only the "what and how" of a design, but also why something is designed as it is. Students learn as much from discussions with each other as they do from material taught by the teachers. As a result, the level of creativity and motivation is found to be significantly above average. Such emphasis on experiential learning with simple patterns and basic concepts is highly successful. This is in contrast to the common belief that a large amount of practical work gives students a better understanding of theoretical aspects (Johansson, Ohlsson, & Molin, 1995).

The Class-Responsibility-Collaboration (CRC) card approach for teaching OO principles is somewhat successful in teaching the concepts of objects to the

novice programmers, and in introducing existing complicated designs to experienced programmers (Beck & Cunningham, 1989; Coplien, 2003). The *class* name gives the vocabulary to discuss designs and the *responsibility* serves as a handle for discussing potential solutions. The more that can be expressed by short verb phrases, the more powerful and concise the designs are. However, Beck and Cunningham (1989) and Coplien (2003) deliberately blur the distinction between class and instance to drive designs toward completion with the aid of execution scenarios to help students retain their focus on abstraction.

Rosenberg and Kolling (1997) develop a simple OO language called the "Blue" to provide an integrated and easy-to-use programming environment for teaching. The tool explicitly supports manipulation of classes and objects, and allows user interaction with objects. Visualisation tools display classes and their relationships so that the design of an OO system is visible and can be discussed easily. The tool encourages students to consciously think about "design first" by forcing them to introduce classes and their relationships before any small scale code is written.

Learning to program in OO style is more difficult for an experienced procedural style programmer than a beginner who starts programming using OO techniques (Rosenberg & Kölling, 1997). It takes average programmers 6 to 18 months to switch their mind-set from a procedural to an OO view of the world. On the other hand, students do not have any difficulty in understanding OO principles if they encounter them first. Switching to a varied programming technique is more difficult than the concept of object-orientation (Kölling, 1999). OO in many occasions is viewed as just another language construct that can be taught after control structures, pointers and recursion. The programming languages used are too complex and the programming environments, if they exist, are confusing. Systems that are currently used for teaching OO principles are developed for professional software developers, making it difficult for the first-year students to cope; others grew out of historic coincidences. Kölling (1999) observes that the models, tools and resources employed in teaching OO should ensure clean concepts, pure object-orientation, safety, high level, simple object/execution model, readable syntax, no redundancy, easy transition to other languages, support for correctness assurance, and a suitable environment.

Even a sophisticated and widely used Interactive Software Engineering tool like "EiffelBench" lacks effectiveness (Kölling, 1999). Kolling observes from a student survey that 37% of the students choose not to use it and prefer a Unix editor and shell instead. Of those who used the graphical environment, 64.7% have not had a good experience, 35.3% remain neutral and none are positive about it.

Designs need to be presented as a two-pronged exercise: structural design, and functional design (Coplien, 2003). Structural design gives a system the long-term stability that can reduce discovery costs, localise maintenance work, and provide a foundation for reuse. The main component of structural view is the system of classes and the relationships between them. Functional design features more strongly in some of the earlier design approaches for computer programming (Kendall & Kendall, 2001) and has made a resurgence in OO environments in the guise of Use Cases. This has resulted in a premature conclusion that OO design notations such as Unified Modelling Language (UML) are about the class and the object diagrams, and the dynamics almost always remain as secondary to the class model.

The Use Case that captures the functional requirements often fall out somewhere along the way once students move on to class design, based on the belief that the class member functions would capture all the needed

functional capabilities (Coplien, 2003). The classes have a hierarchical nature to their structure, and hierarchy is an important abstraction tool. When used effectively, it brings organizational power, but when used to shed essential complexity, it becomes redundant.

The literature that investigated an "objects-first" approach (Ben-Ari, Ragonis, & Levy, 2002; Kölling, 1999) required that students study OO principles from the beginning, so as to avoid a shift of the programming paradigm that occurs in an "objects-late" approach. The objects-first approach has made extensive use of the popular and successful BlueJ educational integrated development environment for OO systems in Java. Although the dynamics of the execution of OO systems are complex, the details are found to be essential for a proper understanding of OO principles (Ben-Ari, Ragonis, & Levy, 2002).

Some of the misconceptions discovered by Ben-Ari, Ragonis and Levy (Ben-Ari, Ragonis, & Levy, 2002) and Kolling (1999) during the delivery of objects-first programs are attributed to the misunderstandings related to the following elements of OO principles:

1. the difference between a class and an object of the class,
2. the execution of a constructor as part of an object creation,
3. the operations can only be invoked on objects,
4. the state of an object and the fact that an operation changes the state,
5. the connection between the execution of an operation and its source code,
6. actual versus formal parameters,
7. the "uses" relation between classes, and
8. the values of fields in an object can themselves be referring to other objects.

The special attraction of BlueJ for teaching OO is that the objects can be interactively created from a class icon, and the methods can be interactively invoked from an object icon. This enables incremental teaching of elementary concepts of OO principles without needing complex test suites. One of the most difficult OO principles concepts to teach is that the fields in the objects could be of a reference type and therefore the value of the field could potentially be pointing to other objects (Ben-Ari, Ragonis, & Levy, 2002).

Results based on empirical analysis of the two methodologies involving process-oriented and object-oriented tasks have been reported by Piattini (2002) comparing the two paradigms as teaching methods for information systems analysis and design. Piattini concludes that OO does not impose an artificial division of data and processes as occurred with functional methodologies, although he could not analytically verify the hypothesis that object-orientation is a more effective method for teaching system development concepts. However, object-orientation reduces the gap between the problem-space and the solution-space in the development process of information systems.

The foregoing analysis of literature indicates that there is a need for further research on the subject of teaching OO at year one. The research leads to the development of appropriate resources and tools.

In the next section, we outline the research methodology in brief and then present the proposed teaching and learning process. It then follows with design, development, implementation, testing validation and evaluation of the system. Finally the paper recommends a solution for teaching and learning object-oriented principles.

Research Methodology

This research adopts the multi-methodological research approach (Nunamaker, Chen, & Purdin, 1991) and the research framework (Hevner, March, & Park, 2004) for design science research. The adapted research framework is comprised of four interrelated phases:

observation; theory building; systems design and development; and realisation and evaluation. The research problem and objectives are derived from an educational strategic issue (Henderson & Venkatraman, 1993) that target to enhance the formulation and implementation of the teaching and learning strategy (Kalakota & Robinson, 2001) for a specialised curriculum item.

We first observe and study the real life problems and issues of learning and teaching object-oriented principles in the classroom environment. We also review theoretical and conceptual ideas in the literature, and currently available tools for teaching and learning OO principles in terms of their technical suitability, accessibility for the students, and support to the students' learning styles. This activity continues during the theory building, and system design and development phases. Extensive observations have been made during evaluation and testing of the system. Observation facilitates the formulation and refinement of the system throughout the research process.

We then extensively investigate the literature and develop a conceptual OOMIT model that addresses the problems and issues identified earlier. Initially, theory building process depends on the outcome of the observation phase, which in turn is used to develop and implement an OOMIT architecture and system in the system design and development phase. The OOMIT system is used as a proof-of-concept to demonstrate improvements of the OO principles learning and teaching. The system is realised, validated, tested, reviewed and evaluated as prescribed by Hevner et al., (2004) and the findings enable a better understanding of the problem which subsequently helped us to improve both the quality of the artefact and the design process. The iteration of observation, theory building, system design and development, and realisation and evaluation phases continue until an acceptable OOMIT system is developed.

The Learning and Teaching Process

We observe that the learners in their first year of an Information Systems degree have significant difficulties in correctly applying the OO principles that govern OO analysis and design, which later manifests as reduced motivation to further their studies and eventually result in poor attendance and performance. We therefore see a clear need to design new experiments, develop better learning and teaching models, and collect and analyse formal classroom and laboratory data in order to carry out an in-depth study of the processes that act as hindering agents to learning. The data collected under controlled conditions may include attendance, motivation, participation and marks obtained in assignments, tests and exams.

This paper proposes that the practice of using Java or Visual Basic languages to teach and demonstrate OO principles and their application be discontinued and the languages be replaced by a new learning tool in the curriculum. The students use the tool to define and describe objects, and their attributes and behaviour in terms of simple UML symbols. The tool needs to allow participants to create and interact with these objects. The students then test their objects under extreme conditions until the objects breakdown or exhibit abnormal behaviour.

It is common practice to provide a lengthy summary of all OO-rules by topic and by principles. These rules and their applications to design situations are explained in detail during lecture sessions, which is followed by demonstrations. However, when the students are asked to test these rules in their tutorial classes and report their findings, along with discussions on costs and consequences of violating them, we observe that they almost always attempt to relate any new experience by either interpolating or extrapolating simpler rules that have been tested and documented earlier, which agrees well with earlier observation (Reid, 1801). Such rule based learning inhibits learners' abilities to apply a

combination of rules and principles without finding newer ones in between. Therefore, no significant improvement is observed by us in the students' abilities to use the knowledge and skills gained to the desired levels of accuracy, which also agrees with previously reported observations (Kölling, 1999). Therefore, this paper suggests a discontinuance of the practice of providing such rule books in order to test earlier observation made by Beck (Beck & Cunningham, 1989). Under the proposed new research, students are asked to maintain a rule book of their own, and are free to enter anything that seemed like a rule for their own future references.

Design and development

The learning tool called OOMIT uses standard UML descriptions of attributes and behaviour to describe objects. Any text editor may be used to describe objects and their relationships between each other. The tool then translates the UML data to Java language source code, and the Java language compiler compiles the source code to executables, as shown in figure 1 below.

Figure 1: The Operational Sequence

The OOMIT's translator has four main components, as shown in figure 2, such as; Worker, Database, Screen and UMLStore to achieve its objective in a two-step process. The first step parses the lines of text describing objects to ensure entry of correct UML syntax while recording the classes, attributes and behaviour in each of the classes, and their relationships as a collection called UMLStore. The second step validates the availability of resources used while translating the UML code to Java language.

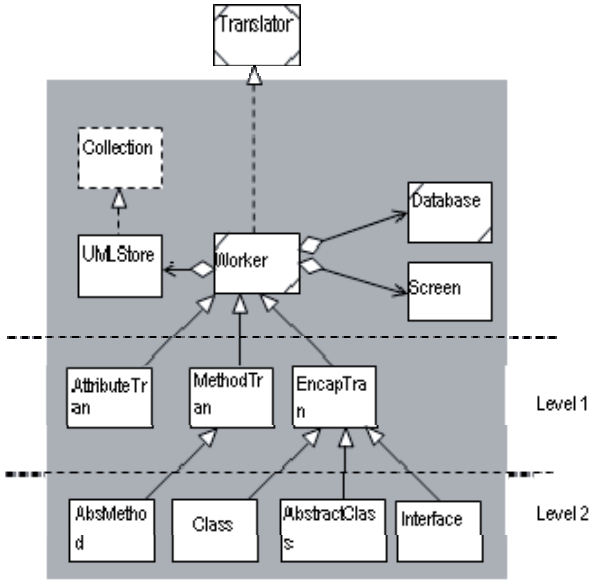

Figure 2: OOMit architecture

The interface **Translator** of OOMIT shown in figure 2 allows access to various features underneath that are responsible for the translation of various segments of lines that are read into the system. The Worker component provides the translation functions in two hierarchical levels as shown in figure 2. These lower level specialized components handle translations of attributes and their data types, object behavior and encapsulating devices, such as; Abstract Classes, Interfaces, and Entity Classes.

In formal computing languages, one needs to specify the end of every set (classes, methods, etc.), either with curly brackets or an "End" statement. In order to remove such formal expressions and look more like an UML expression, OOMIT allows learners to enter the following to describe classes such as; **Employee** and **Customer**:

```
Employee
  +firstname:text
  +lastname:text
  +work()
  print "Employee works"

Customer
  +name:text
  +buy()
  print "Customer buys"
```

An abstract class or an interface is entered as follows:

```
abstract class Vehicle   :      (a)Vehicle
interface Animal         :      (i)Animal
```

The UML code for abstract methods require a semicolon placed at the end of its signature, as follows:

+m1();	Accepts no parameters and returns no value
+m1(..);	Accepts parameters but returns no value
+m1():;	Accepts no parameters but returns a value
+m1(..):;	Accepts parameters and returns a value

As these non-implemented methods in an abstract class definition could pose problems to the translating processes in detecting changeovers between methods, the tool requires placing of the abstract methods before implements. Introduction of such a rule offers no special resistance, as it is common practice to place the abstract items at the beginning in order to be noticed by implementers. We also introduce additional symbols to extend the standard UML list of symbols to represent inheritance from a class or an abstract class, and implementation of an interface as follows:

Car > (a)Vehicle	Class Car inherits/extends an abstract class Vehicle
Car >> (i)Vehicle	Class Car implements interface Vehicle
Car > Roadster >> (i)Vehicle	Class Car inherits class Roadster and implements interface Vehicle
A > B >> (i)C1,(i)C2,(i)C3	Multiple implementation

The symbol '**^**' is used for typecasting, as follows:

```
Car > Vehicle
v:Vehicle = new Car
c:Car = ^(v,Car)
```

We limited primitive data-types to the following four basic types with a '**+**' or '**–**' sign to indicate public or private as specified in the UML syntax (Version 1.0 and 2.0):

+age:number	stores integer values
-salary:decimal	stores decimal values
+name:text	stores text
+go:boolean	stores true/false
+nameList:text[]	Stores an array of text

We also provide some basic library classes to connect to a Microsoft Access Database to execute simple SQLs and additional facilities to create a simple graphic user interface to test designs. The GUI accepts input in a data dictionary format. The typical instruction sets for decision making and running loops are shown below:

loop 10 times	run the line that follows 10 times
loop changing i by 1 from 0 to 9	run the line that follows 10 times
loop as long as i<10	run the line that follows until the value of i reaches 9
if age<10	if true execute the line that follows
if age<10 … else	if true execute the line that follows the if statement or execute the line that follows the else statement

As only one line that follows can be executed in all the above instruction sets, learners are forced to form the habit of decomposing large tasks into smaller methods.

Implementation, Testing and validation

The graphical user interface of OOMIT is shown on in figure 3.

An example scenario that is shown in figure 3 is detailed in figure 4.

The **Translate** button automatically converts pseudo-code into Java, which significantly decreases the complexity of the tasks required by students. The **Compile** button compiles the Java source code and either reports error on the first window or displays success of compilation. The user then enters the class name that contains the main method and uses the **Run** button to execute. The results are displayed on the first window. For minor changes, the **Execute** button may be used to translate, compile and run in one go. The results could also be displayed as a text document.

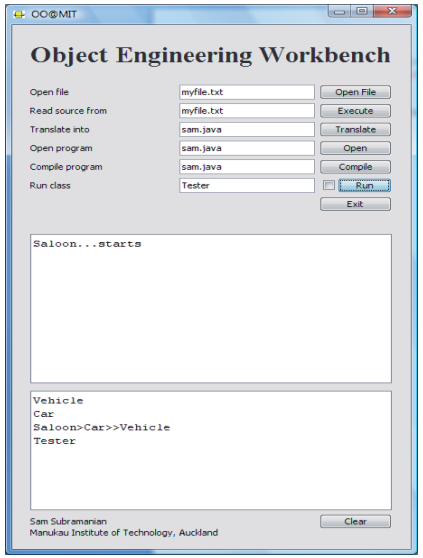

Figure 3: OOMIT User Interface

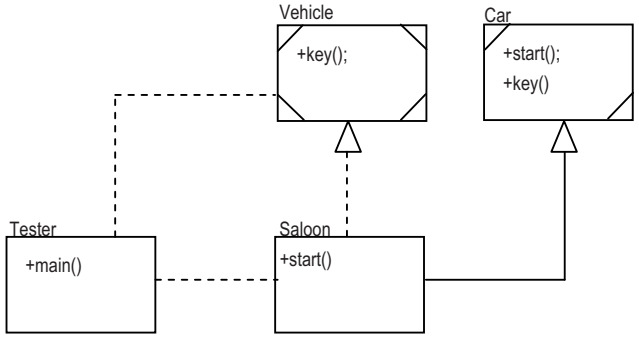

Figure 4: An example scenario

Evaluation

A number of approaches such as peer review, classroom evaluation, and critical analysis have been applied for the evaluation of the research process and artefacts. Peer review was conducted through a range of presentations to individual experts to seminars. The participants comprise of general and programming academics, and system designers and architects. These experts provided useful feedback, which were incorporated in refining the artefacts. Students also used the system for practicing and learning OO principles. During the process, we observed the *Functionality, Usability, Reliability, Performance and Supportability* features of the system and the findings were incorporated to refine the artefacts.

The design science perspective and seven guidelines proposed by Hevner et al. (2004) has been quite influential in generating a stream of research ideas and implementations that have generally been shown to be effective. These guidelines aid in the acquisition of knowledge and understanding of a design problem and its solution through building and applying an artifact. We consider each of these in turn and briefly discuss how our research process and OOMIT implementation conform to the guidelines.

1. Design science requires the creation of an innovative and purposeful artefact.	This research proposes an innovative learning tool.
2. The artefact thus created must be relevant.	Proposed OOMIT system is useful and relevant for learning and teaching OO principles.
3. Thorough evaluation of the artefact must reveal its utility.	Both the academics and students evaluated OOMIT system through its application in the classroom environment.
4. Research contributions.	It contributed to the enhanced learning of OO principles.
5. The artefact must be rigorously defined, represented, coherent, and consistent.	OOMIT supports the principles of object orientation, and also facilitates creation and testing of the objects.
6. The artefact or its creation process is the best in a given problem space.	Iterative and expert peer review processes were adopted for the creation of the proposed system.
7. The results from the creation and use of an artefact must be communicated to both researchers and practitioners.	Students' experience of using this system and its capability to meet OO principles learning needs have been presented in faculty seminars.

Conclusion

Learning and teaching object oriented principles at year one has been challenging to both students and academics. The major factors that contribute to the difficulties are: 1) the procedure orientation of learners, 2) use of formal computing languages in understanding objects, and 3) application of the complex relationship between objects for their interaction to demonstrate overall system behaviour. The learning sessions need to avoid use of formal computing languages, avoid introduction of procedures and methods (behaviour) until the students have grasped the idea of objects, and support learning, with proper tools.

This research follows the design science research framework to propose a methodology supported by a tool to overcome identified problems and issues in learning and teaching object oriented principles. OOMIT encourages learning by constructing familiar objects and conducting experiments on them. OOMIT has been successfully used in the classroom environment. The proposed methodology and tool are suitable to be

adopted to the teaching of OO principles at the tertiary level.

References

Astington, J. W., Harris, P. L., & Olson, D. R. (Eds.). (1988). Developing Theories of Mind. Cambridge: Cambridge University Press.

Beck, K., & Cunningham, W. (1989). A Laboratory For Teaching Object-Oriented Thinking Paper presented at the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), New Orleans, Louisiana.

Ben-Ari, M., Ragonis, N., & Levy, R. B.-B. (2002). A Vision of Visualization in Teaching Object-Oriented Programming. Retrieved 15 August, 2009, from http://stwww.weizmann.ac.il/G-CS/BENARI/articles/noa-pvw.pdf

Bennedsen, J., & Caspersen, M. E. (2004). Teaching Object-Oriented Programming – Towards Teaching a Systematic Programming Process. Retrieved 10 August, 2009, from http://www.cs.umu.se/~jubo/Meetings/ECOOP04/Submissions/BennedsenCaspersen.pdf

Coplien, J. O. (2003). Software: The Next Generation, Teaching OO: Putting the Object back into OOD. Retrieved 10 July, 2009, from http://www.artima.com/weblogs/viewpost.jsp?thread=6771

Henderson, J., & Venkatraman, N. (1993). Strategic alignment: Leveraging information technology for transforming organisations. IBM Systems Journal, 32(1), 472-484.

Hevner, A. R., March, S. T., & Park, J. (2004). Design science in information systems research. MIS Quarterly, 28(1), 75-105.

Johansson, C., Ohlsson, L., & Molin, P. (1995). Teaching Object-orientation: From Semi-colons to Frameworks.

Kalakota, R., & Robinson, M. (2001). E-business 2.0: Road map for success. Boston, MA: Addison-Wesley, Pearson Education.

Kendall, K. E., & Kendall, J. E. (2001). Systems Analysis and Design (5th ed.). NJ: Prentice Hall.

Kölling, M. (1999). The Problem of Teaching Object-Oriented Programming, Part 1: Languages. Journal of Object-Oriented Programming, 11(8), 8-15.

Nunamaker, J. F. J., Chen, M., & Purdin, T. D. M. (1991). Systems development in information systems research. Journal of Management Information Systems, 7(3), 89-106.

Piattini, M. (2002). Teaching Object-oriented and Functional paradigms in software engineering/development. Journal of Informatics Education and Research 3(2).

Reid, T. (1801). An Inquiry into the Human Mind on the principles of common sense. London: Ad Nfill and Co.

Rosenberg, J., & Kölling, M. (1997). Blue - A System For Teaching Object-Oriented Programming. Paper presented at the OOPSLA97 Workshop: Resources for Early Object Design Education, Georgia.