The nature of an object-oriented program: How do practitioners understand the nature of what they are creating?

Errol Thompson[a]*, and Kinshuk[b]

Computer Science, Aston University of Birmingham, Birmingham, UK, [b]School of Computing and Information Systems, Athabasca University, Canada

Dr Errol Thompson,
Computer Science,
Aston University,
Aston Triangle,
Birmingham, B4 7ET,
United Kingdom

Email: kiwiet@acm.org

Errol Thompson completed a PhD in Education at Massey University while lecturing in the Department of Information Systems. He has an M.Sc. in Computer Science from the University of Canterbury, New Zealand. He has taught in a number of tertiary institutions in New Zealand and at The University of Birmingham. He is currently teaching software development topics in Computer Science at the Aston University, Birmingham, United Kingdom. He has practical experience of the software development industry having worked in a number of roles in the New Zealand computer industry.

Kinshuk is NSERC/iCORE/Xerox/Markin Industrial Research Chair for Adaptivity and Personalization in Informatics, Associate Dean of the Faculty of Science and Technology and Full Professor of School of Computing and Information Systems at Athabasca University, Canada. He has a PhD from De Montfort University, United Kingdom. Areas of his research interests include learning technologies, mobile and location aware learning systems, cognitive profiling and interactive technologies. In his on-going professional activities, he is Founding Chair of IEEE Technical Committee on Learning Technologies, and Founding Editor of the Educational Technology & Society Journal (SSCI indexed with Impact Factor of 1.067 according to Thomson Scientific 2009 Journal Citations Report).

The nature of an object-oriented program: How do practitioners understand the nature of what they are creating?

Object-oriented programming is seen as a difficult skill to master. There is considerable debate about the most appropriate way to introduce novice programmers to object-oriented concepts. Is it possible to uncover what the critical aspects or features are that enhance the learning of object-oriented programming? Practitioners have differing understandings of the nature of an object-oriented program. Uncovering these different ways of understanding leads to a greater understanding of the critical aspects and their relationship to the structure of the program produced. A phenomenographic study was conducted to uncover practitioner understandings of the nature of an object-oriented program. The study identified five levels of understanding and three dimensions of variation within these levels. These levels and dimensions of variation provide a framework for fostering conceptual change with respect to the nature of an object-oriented program.

Keywords: perception; understanding; program structure; critical aspects; phenomenography; object-oriented; programming;

Introduction

Starting from the perspective that programming is regarded as difficult to learn and that object-oriented programming is even more difficult (Lister et al., 2006), a study was conducted to identify the critical aspects of practitioners' understanding of object-oriented programming. The study was built on the notion that there is a connection between a learner's perception of what is to be learnt and how learning is achieved, and the learning outcome (Ramsden, 2003), and that there is also a connection between a practitioner's perception of the task and the outcome of that task.

A person's perception of a task is influenced by their understanding or awareness of related concepts (i.e. their conceptions) (Waddington, 2008). In a programming task, the programmer's perception of the requirements of the task is influenced by their understanding or awareness of the nature of a program. The research described in this paper seeks to uncover how practitioners understand the nature of an object-oriented program.

Theoretical background

The context of this research is how to help learners understand object-oriented programming. In order to understand the nature of learning, both a foundation in learning research and models for learning have been explored. Ramsden's (2003) "learner learning in context"

model was used as the basis for this research. This model discusses the perception that the learners have of the task and how it is an influence on their approach to learning and the nature of the learning achieved.

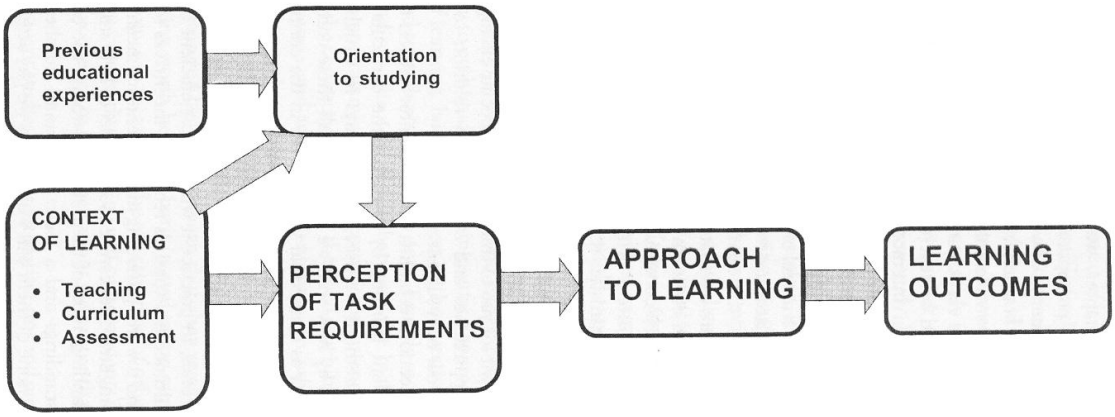

Figure 1: Ramsden's learner learning in context (Ramsden, 2003, p 82)

Ramsden says that "Approaches to learning are not something a student has; they represent what a learning task or set of tasks is for the learner" (2003, p 45). This perspective of learning paralleled the experience of the first author when he was teaching imperative programming. The students did not have any concept of the nature of a program and struggled to write correct programs. After the lecturer developed learning resources (Thompson 1992) that encouraged the development of Wirth's (1976) view of a program as algorithm plus data structures, the learners were able to program more easily.

Marton (2000) contends that "The problem doesn't exist as such, it is always understood in one way or another" (p 113). Any given task or problem is understood based on previous experience. The learner is either aware or unaware, or conscious or not conscious of the problem. The concept of a problem exists only in an awareness relationship.

Learners will experience the learning exercises that are set in different ways. Marton illustrates this with an exercise in division (p 110-111). The teacher has a particular idea of how the problem should be solved but the learners use a range of techniques based on their own understanding of the problem and how to solve it.

Working from this perspective, Bowden and Marton (2003) argue that the most important form of learning involves a change in the way that the learner sees something in the world. These changes involve changes in the way that the learner understands what is to be learnt and how learning is achieved (Marton & Booth, 1997).

If the learner's perception of the task is important for influencing their approach to learning then the teacher needs to know what task representations are appropriate for the required learning.

Understanding of Object-Oriented Concepts

Some work has been conducted that explored student understanding of the concepts of object and class (Eckerdal & Thuné, 2005). This work produced two parallel hierarchies (see Table 1); one for the comprehension of the concept *object* and the other for comprehension of the concept *class*. Eckerdal and Thuné acknowledge the similarity of these two hierarchies. The concept of a class as a description of an object is clearly visible in the two highest categories but not so clear in the initial category. The shift from the first category to the second is the realisation that the class and object are distinct; the object is not the code of the class but rather an entity that is described by a class.

Table 1: Comprehension of object and class (Eckerdal and Thuné 2005)

Conception of object	**Conception of class**
Object is experienced as a piece of code.	Class is experienced as an entity in the program, contributing to the structure of the code.
As above, and in addition, object is experienced as something that is active in the program.	As above, and in addition, class is experienced as a description of properties and behaviour of object.
As above, and in addition, object is experienced as a model of some real world phenomenon.	As above, and in addition, class is experienced as a description of properties and behaviour of the object, as a model of some real world phenomenon.

Lister et al. (2006) when examining the debate on whether to teach objects first summarised the key claims made in the debate. They concluded that there were four key claims. These are:

- Claim 1: OO programming is more difficult to learn than imperative.
- Claim 2: Students who learn objects-first do not learn algorithmic problem-solving.
- Claim 3: Students who learn imperative first find learning OO programming difficult.
- Claim 4: Successful student learning is dependent on a match between the language paradigm and the teaching paradigm.

Although Lister et al. identified some research that supported the first two claims; they found none that supported the latter two claims. The research referenced in relation to the first claim although identifying possible issues with the use of object-oriented programming, did not explicitly address the question of whether OO is more difficult than imperative. In relation to the second claim, the authors reference two studies and then say that "Replication and extensions of these types of studies would be helpful to the CSEd community." For the third claim, they identified two reports with anecdotal evidence to support the claim but found no reports based on systematic studies. They found no studies that supported or refuted the fourth claim.

In further investigation of the debate on teaching object-oriented programming, Berglund and Lister (2007) concluded that there are two fundamental understandings of what *objects first* meant. In their phenomenographic analysis, they identified three dimensions of variation that helped define the two understandings (see Table 2).

Table 2: The dimensions of variation of *objects first* (Berglund & Lister, 2007)

	Category 1. Objects first as an extension of imperative programming	**Category 2.** Objects first as something conceptually different from imperative programming
DoV1 Program execution	Objects are passive and are used when the program is run	Objects are active. Object interaction gives the algorithm
Dov2 Polymorphism	Polymorphism as different objects	Objects interact polymorphically
DoV3 Modifications	Modification as changing code	Modifications as adding to holes and hooks

In a continuation of the study, they added a third category which says that object-oriented programming transcends both Category 1 and Category 2 (Berglund & Lister, 2010). In this new category, they contend that the variations of the first two categories are not relevant and that "they are simply variations on a single theme".

These studies have focused on the educational content by either seeking to learn about the student perspectives or to explore the expressed understanding of the academics. None have sought to understand the way that practitioners conceive the nature of an object-oriented program or their approach to the task. There remains an open question with respect to whether these same understandings are reflected in practitioner perspectives.

Novice / Expert Distinction

Burkhardt, Détienne, & Wiedenbeck (2002) conducted research to evaluate expert and novice development of mental models with respect to specific tasks. To conduct their research they adapted a programming comprehension model that had been created by Pennington (1987a, 1987b). They argue that "novices do not spontaneously construct a strong situation model but are able to do so if the task demands it" (p 2). Both experts and novices develop similar program models.

However, Sajaniemi and Kuittinen (2007) argue that further research needs to be conducted to identify what mental representations are used by object-oriented programmers.

They argue that we need to know more about object-oriented experts' mental representations with respect to object-oriented programming, the cognitive development of novices with respect to the paradigm, methods to convey the object-oriented notational machine to novices, and novices' and experts' program comprehension processes (p 96-97).

Research conducted

If a learner's approach to learning is influenced by their understanding of the task then the teacher needs to know what representations are appropriate to stimulate the desired approach to learning. This led to the following research question: How does the awareness of an object-oriented program vary among 'practitioners'?

a) What are practitioners' ways of experiencing object-oriented program?
b) What are the "critical aspects" that distinguish the different variations in awareness expressed by 'practitioners'?

Thirty one practitioners who had some level of awareness of object-oriented programming and had written programs using this technique were interviewed. The practitioners interviewed had a wide range of experience and ranged from recent graduates with some industry experience to those regarded as leading practitioners in the field. The practitioners were chosen to ensure that a full range of experience and understanding was obtained. The selection of participants was both a convenience sample and a purposive sample.

The practitioners were initially asked "How do you assess whether a program has been written using object-oriented techniques and practices?" Early trials used sample code but this proved a distraction and was discarded. Further questions were then asked based on the interviewee's responses. Closer to the end of the interview, interviewees were also asked a question similar to "How would you describe what a 'program' is in an object-oriented context?" This varied depending on the direction that the interview had taken and whether the

interviewee asked for clarification. The objective of the questions was to uncover how the practitioners described the nature of an object-oriented program as this was believed to be the 'what' aspect influencing the perception of the task.

The interviews were analysed using phenomenographic techniques (Marton, 2000). Phenomenography is "an empirical research approach to the study of experience; the aim of phenomenographic research is to describe the essential and qualitative variation in the ways people experience phenomena or a particular phenomenon" (Booth, 2001, p 12). By conducting a study of practitioners with a wide range of expertise in object-oriented development, it may be possible to determine what representations may be effective for learning.

The analysis uncovered categories that described the different understandings of the nature of an object-oriented program and dimensions of variation (critical aspects) that distinguished the differences between the categories. The dimensions of variation help define the distinction and hierarchical relationship between the categories (Cope, 2002). A category should vary from another through a difference in one of more of the dimensions of variation or the relationship between the dimensions of variation.

Two different types of responses were received in the interviews. The response to the question "How do you assess" tended to focus on design issues of an object-oriented program (the 'how' aspect), and for the question on the nature or structure of an object-oriented program (the 'what' aspect). This paper reports on the results of analysing the data based on the nature or structure of an object-oriented program.

Results

Interviewees recognised the nature of a computer and the constraints that this placed on the nature of a program. Some were able to see beyond the technical constraints and visualise an object-oriented program as having a technical structure that was not constrained by the nature

of a computer or the operating system or environment in which the program would be executed. This led to five distinct categories.

The five categories are sequence of instructions (N1), written using an object-oriented framework (N2), based on data types (N3), based on interacting entities (N4), and artificial construct (N5). Three distinct dimensions of variation (aspects) helped define these categories. The two most visible dimensions are the flow of control and object usage. The flow of control influenced the meaning associated with two of the categories. The influence of how objects are used helped separate out the remaining three categories. Behind these two categories is the nature of the problem solution (see Table 3).

Category	Referential	Structural	Critical aspects		
			Flow of control	Object usage	Problem solution
N1	Sequence of instructions	Recognises that all programs have a flow of control that defines the order in which program instructions are executed. The order of execution is seen primarily as sequential in nature	Flow of control is seen as sequential	Objects are used from provided framework	Provides required functionality
N2	Written using an object-oriented framework	Sees a program as object-oriented if it uses an object-oriented framework or using an object-oriented environment	Flow of control is still seen as sequential but is not a major focus	Programmer objects are defined and used but primarily as a requirement of the language or framework	Provides required functionality
N3	Based on data types	Sees an object-oriented program as being based on defining new data types or on data analysis or the development of a data-driven model	Flow of control is not explicitly discussed or considered an issue	Objects are defined based on data requirements	A data model of the problem space (real world model)
N4	Based on interacting entities	Sees an object-oriented program as a set of behavioural entities that interact to achieve the required objective	Flow of control is defined by the interactions between objects	Objects are defined based on behavioural requirements	A behaviour-driven model reflecting objects in the problem space
N5	Artificial construct	Sees a program as an artificial construct imposed by operating systems and not applicable to the discussion of object-oriented entities	Flow of control is defined by the interactions between objects	Objects are liberated from the confines of program boundaries	A behaviour-driven model using useful abstractions

Table 3: Categories of the nature of an object-oriented program

The categories and the relationship with the dimensions of variation (aspects) are

discussed in the next sections.

N1) Sequence of instructions

The focus in this category is on what the computer has to do with the program once it is created (flow of control) and on the tools used in its creation (objects from an object-oriented framework). This category shows an understanding of what the computer will do with the program and therefore of attributes of the internal structure of the program. The primary dimension of variation which distinguishes this category is the sequential nature of execution. The supporting dimension that makes it object-oriented is that objects are used from the supporting framework.

Put in the simplest of terms, "A program is some instructions to a computer to tell it what you want it to do" (I16).

In this view, the sequential nature of a program is linked with the way a program is loaded and executed within a computer system. This operating system requirement influences the understanding of what a program is. This is expressed as being independent of the programming paradigm.

> "It's a set of instructions that when it is compiled or executed it does something" (I27).

Participants who expressed this view suggest the sequential nature is not limited to how the program is executed. In designing a program, this sequential execution needs to be considered. How this is expressed in program code is dependent on the choice of programming language but does not remove the sequential nature of the program.

> "Getting across the idea that there are sequence or set of things you are wanting to happen and the way that you do that is write instructions and you are confined in how you write those instructions by the language you are going to use" (I12).

The emphasis is upon a program primarily being sequential in nature even though objects are being used. This view was expressed from the perspective of the sequential nature of any computer program. As such, it was seen as a characteristic that also applied and needed to be considered in the design of object-oriented programs.

With this category, there is some attempt to be able to identify a distinguishing characteristic of an object-oriented program. The base is seen as being sequential in nature but there is an emphasis on needing to use objects from an object-oriented framework. Using objects in this category is seen more from the perspective of being a requirement of the language or development environment and not as a means for structuring one's own code.

N2) Written using an object-oriented framework

This category focuses on the way classes and objects are used. This is emphasised in the context of using some form of pre-existing object-oriented framework or language in the creation of the program.

It places minimal emphasis on the way in which the program is executed although does not discard that emphasis. The focus is on the aspects of an object-oriented framework or the expected features of an object-oriented language, and places emphasis on the structural aspects provided by the framework. The object-oriented framework is seen to include those language features that are specific to class-based object-oriented languages. This category has a technology focus and could be split into subcategories depending on the technologies emphasised.

The simplest expression of this view is that if the program is written in a language that is referred to as being object-oriented then the end program must be object-oriented.

> "if something is written in Java or C#, I would think that was an object-oriented approach of doing programming" (I29).

Some wanted to ensure that specific language features were used in writing the program.

> "There is the special features like I said, inheritance and then there's encapsulation ... it doesn't matter using private which means they are not going to be public to other users." (I31)

Some argued that an object-oriented language is a strong indicator but not necessarily sufficient on its own. For these participants, they are expecting the use of some features or some characteristics that would make the program object-oriented. This is reflected in the following statement where there is a question about the size of a class.

> "First, I would have a look at the language that it was written in. That would certainly help. Some people can use object-oriented languages but not necessarily use the object-oriented features. We could use C++ as an example. They might actually code it like C or they might code Java as one big huge class which doesn't actually represent a whole bunch of objects" (I3).

This category sees an object-oriented program as using some sort of technological framework or techniques for programming. Using an object-oriented language makes this easier but isn't an essential requirement.

N3) Based on data types

The aspect that distinguishes this category from the other categories is the way in which objects are used within the program. This category highlights a view that sees object-oriented programming as an ability to extend data types provided by the programming language. It represents a data focussed or data model view of how to write programs. There is little concern about the flow of control in describing the characteristics of an object-oriented program.

One participant argued that this is the basis for handling complexity in large programs.

> "Object-oriented design says structure your program around the types of information you've got. For each type, build a module of your program that deals with that type of data." (I22)

The participant further clarified this based on language implementation when he said:

> "Common Lisp object system where objects are basically data types or instances of data types and you have generic methods that will be dispatched and inheritance and stuff. But […] the Smalltalk, C++, Java family, the idea is you can have classes which are types with attached methods" (I22).

In this perspective, the use of classes within the language is linked with the data type system. If the programmer writes a class then they are defining a new data type. This is based on the principle that abstractions within the code are based on data types.

> "How was this program broken up into modules? What are the abstractions? And if the abstractions are based on the data types, on the chunks of information that the thing processes and if that's where the boundaries were drawn around those then I'll say it's object-oriented" (I22).

Abstraction and modularisation are clearly linked with the defining of new data types.

With this category, there is a shift away from a focus on flow of control and more emphasis given to the nature and use of the objects. This emphasis sees the objects as

extending the types of the language and as a result, the program representing a data model of the real world or problem domain.

N4) Based on interacting entities

Where the previous category focused on the data aspects as the way to use objects, this category returns to a focus on flow of control and focuses on the use of objects or components in interactions to implement the behaviour of the program. It sees a program being based on the interactions between objects or components.

The emphasis is on the interactions between objects rather than what the objects represent. This could be described as a focus on the behaviour of the objects, or the implementing of algorithms through the interactions between objects. Often this category was stated bluntly, as in:

> "An OO program as a whole is a collection of communicating objects" (I23).

A participant expanded on this idea based on his objective in writing software.

> "One of my goals when I build a piece of code is to write simple classes with simple methods and have the complexity of the software in the interactions between the objects rather than implemented explicitly in the methods of the objects" (I14).

This can be contrasted with the previous category (based on the use of data types) where a participant talked of dealing with complexity through the use of new data types. Here the focus on dealing with the complexity of the software is on the interactions between the objects (the way that they work together to achieve the desired result, the behaviour) rather than the data.

Another compared objects and their interactions with those of molecules.

> "objects are a lot of fine grained things interacting with each other generally ... and objects are kind of like molecules" (I18).

Molecules interact to form new substances and build complex systems. This is reinforced by another participant who saw the interactions as critical in achieving anything with object-oriented programming.

> "It is the interaction between objects […]. If this system consists of many different objects then this message passing is the way that all these different objects communicate with each other to achieve the goal of the entire system" (I29).

The interaction isn't simply to retrieve data. The interaction is to accomplish something. It delivers a service or some functionality, some desired behaviour.

This is reflected in an approach to teaching based on metaphors. In this case, an object-oriented program is seen as similar to the interactions in a restaurant.

> "So in practice and teaching that means often to talk about objects as if they were people. I use a lot of metaphors where people have interacted. Where you're a customer in a restaurant, you talk to the waiter, the waiter talks to the chef" (I20).

Each of the workers has a task to perform and together they provide the services for the customer. This participant saw this as reflecting what objects are doing in an object-oriented program. Each object has a service or behaviour that it provides to the system. This participant went further and argued that the interactions are based on a behavioural-driven model of the problem domain.

> "An object is entirely defined by its behaviour. Data is secondary. Behaviour is what makes an object. Behaviour means which messages it understands or in Java terms which methods it has and how they behave, what they do […] Fields are entirely driven by behaviour. You introduce fields to support the intended behaviour" (I20).

This concept was also linked to the ideas of recursive structures and nested virtual machines.

> "One of the things you have in OO is the sort of recursive idea of an object so your system is recursively composed of objects and the relationship between your system and then the actors […] in the outside world is the same between a particular object in the system and its real object communicate. This is Alan Kaye's idea of an object as a recursion of the computer itself. Object-orientation builds on and amplifies Dykstra's model of nested VMs because you have got that recursion there" (I13).

This view is present in the interpretation of an object as a set of machines.

> "They had well compartmentalised all the behaviour and associated state into objects. [...] Object-oriented programming is making all these objects and then just sort of setting them going and off they go and do their thing. […] Making the object is like making other little machines. All of the classes are like how to make a machine and then in the program, you just make a bunch of all these different type of little machines and you start them going by calling a method on them and that calls another method on another one" (I5).

Object-oriented programming was seen as a way for decomposing the problem space into smaller problems and recognising the relationships and interactions involved.

> "Object software is software that uses a particular decomposition strategy for breaking up this large problem into sub-problems, and sub-sub-problems and sub-sub-sub-problems based around both the division of data and division of operations or services and the relationships between them and the communication between them, one the relationship defines" (I28).

The description of the interactions and relationships between objects was seen as being core to the development process.

> "The objects have to communicate with each other to achieve a task. […] It's not just a fact they are identifying objects but they're identifying the relationships between those objects" (I28).

As well as talking about an object-oriented program as interacting entities, there is an emphasis on a program being a model of the real world.

This is reflected by a participant who argued for the use of metaphors to illustrate the interacting objects understanding. This participant recognised the limitations of using metaphors but

> "mapping it onto people in our real world, the metaphor carries very far actually […]. And so starting to think about programming in terms of interacting objects is the most important step and everything else can follow from this. So in practice and teaching that means to talk about objects as if they were people. I use a lot of metaphors where people have interacted" (I20).

This is reinforced in terms of modelling when the participant argued, that for object-oriented programming,

> "Programming first and foremost is creating a model of the world and object-orientation is the tool that you have to model the world. And almost as a side effect, almost by accident that model is executable" (I20).

A program is broken up into objects that represent real world entities and that interact to achieve the desired objective of the program. As one participant put it, the algorithm is represented by the interactions rather than the logic contained in the methods. This is a major difference from the "sequence of instruction" category and also from the "data type" category. None of the participants who expressed this view saw interactions being related to the use of multi-threaded code or environments that fostered interaction between components. They argued that interactions between objects were fundamental to object-oriented programming.

N5) Artificial construct

The research interview placed emphasis on the use of object technology in the creation of programs. For some participants, this was a too restrictive view.

In a very direct response, one participant saw a program as something clearly originating from the requirements of the operating system and not really of interest to the software developer.

> "I don't know what a program is. I mean a program is very much a completely uninteresting idea that is enforced by certain languages and operating systems, like Unix. The Unix operating system, for instance, a program is something that can be executed by exec. Other operating systems are just collections of functions and classes and sort of live inside the execution of a program. So the concept of a program in that case doesn't make sense. So I think the concept of program is essentially a ridiculous invention that's necessary by very restrictive languages and operating systems" (I24).

A program is something expected by the operating system. The operating system starts and terminates the program, but a program is not seen here as being integral to object-oriented. The operating system requirement is seen as a restriction that has been removed by some environments.

The contention here is that the notion of an object-oriented program as a container for a set of objects is restrictive. An object or set of objects (i.e. component) should not be constrained by artificial program boundaries. Environments like Common Lisp remove this distinction. The same objects and components could be used to solve a number of different problems and be linked together in different ways. As a consequence the notion of an object-oriented program becomes a restriction on the way of thinking about how components and objects can be used.

This focus on reuse is also reflected in the view that true components should be like integrated circuits and hardware components. The unit can be replaced or upgraded without throwing away all of the code. This is reflected in

> "the whole goal of component software is that […] you want it to be like hardware components, I can plug in an IC and can pull that IC out and I can plug boards into a computer and different components. I can have different CD-ROM players or DVD players and all these components have clean interfaces and I can substitute one for another in the hardware" (I30).

This view promotes the idea that development isn't about programs but rather it is about reusable components. This participant took this further to argue for a new way of defining the interface to a component or object. He called this a collar interface based on his hardware understanding. He said:

> "So the goal is that you should be able to draw a line around the component and understand all the things that are coming in and going out of it. So all the inputs that come in and all the outputs that go out; that's standard hardware. Every pin is either an input or an output or sometimes both and you can exactly define what the inputs and the outputs are. And the curious thing about object-oriented programming is that while they claim that they're really interested in interfaces, they've really only handled half of it. They really only define the interface into an object and while they

> do have the interface of the result of the object, what they are missing is that any object can arbitrarily call some other object internally and there is never any definition of that interface" (I30).

The objective for this participant was to be able to define reusable units that could be easily extracted and replaced. A program is simply composed of reusable components.

Along with the notion that a program is an artificial construct, there is increased emphasis on the model being based on useful abstractions. This shift was seen in discussion in the previous category in relation to models where some real world entities might be excluded from the model and other objects added that enhanced the ability to implement the model. This perspective is also present in this category.

> "You could get that structure but I would expect some of my domain phenomena to be reflected in classes but there would be other classes that are not necessarily representing that. But certainly, I would expect my domain objects to be represented as collections of classes, it might be a hierarchy of classes and even distinct hierarchies of classes if that's justified from an implementation point of view." (I24)

This category sees the notion of a program as having historical roots. It relates to what a computer system has done historically, and not necessarily to what is now desired of software systems. The notion of a program places an artificial barrier to the desired interactions and reuse of the functionality. In essence, there is a desire to have objects or components that can interact across boundaries and be used flexibly to construct systems.

Discussion

The analysis technique used does not identify learning dependencies nor does it indicate who tends to hold particular levels of understanding. Participants may express different categories of understanding at different times during the interview. What the categories describe are variations in understandings or awareness of the phenomenon and the dimensions of variation that help distinguish the categories.

The context of the interviews was how object-oriented programming is understood. The dimensions of variation and relationships between those dimensions are more complex at higher levels. For example, participants expressed that to understand flow of control as interactions (N4 and N5), it was still necessary to understand sequential flow of control (N1

and N2) for the writing of methods. They did not express the reverse that knowledge of interactions was required to understand sequential flow of control.

Defining objects based on behavioural requirements (N4) requires an understanding of data requirements (N3). Liberated objects (N5) assume an understanding of both behavioural (N4) and data based objects (N3). All depend on knowing how to use objects from a framework (N1) or to define objects using the language constructs (N2).

Being able to identify the level of understanding based on these categories provides a framework for evaluating student learning.

An alternative view is to see the categories as possible targets for learning and to explore how to plan teaching strategies to achieve the desired outcomes (Thompson, 2010). Variation theory has been used to identify the space of learning by identifying the variations used in a teaching strategy (Marton & Tsui, 2003). The corollary is that the teacher should be able to plan the variation around the object of learning to ensure that the appropriate space of learning is opened up to the learner. These categories provide some of the required variations.

With the emphasis on interviewing practitioners rather than students or academics, this work expands the categories reported by Eckerdal and Thuné (2005) by providing variations not covered by their work. This work also identifies three dimensions of variation that help distinguish the categories.

With respect to the second claim in Lister et al. (2006), if flow of control is seen as sequential in nature and this is seen as a core definition of an algorithm (Lewis & Loftus, 2006) then this study would suggest that a learner who has developed an understanding of object-oriented programming at level N3 through N5 may indeed not learn this sequential oriented algorithmic thinking.

Exploring the understanding of algorithms further it is possible that a learner having learnt object-oriented programming has indeed developed algorithmic thinking if algorithms

are seen in terms of interactions represented in design patterns as proposed by Nguyen, Ricken, and Wong (2005).

Exploring patterns as an alternative to algorithms reveals that there are other proposals that have made this suggestion. Clancy and Linn (1999) argued that "textbook writers and course designers need to create design patterns that students find accessible and can connect to new problems" (p 40). They argue this on the basis of using them to scaffold learning as proposed by Rosson and Carroll (1996). The patterns that Clancy and Linn seek to have used are more like Beck's elementary patterns (1997, 2007) rather than the more complex design patterns documented by Gamma, Helm, Johnson, and Vlissides (1995).

Brookshear (2007) says that "an algorithm is abstract and distinct from its representation. A single algorithm can be represented in many ways" (p 214). Beck (2007) says that "patterns are based on commonality" (p 5). They are things that programmers repeatedly do. In the case of implementation patterns, they are applied "every few seconds" (p 2). Design patterns may be used a few times a day (p 2). Beck's implementation patterns are more low level than algorithms, and algorithms are not seen in design elements. There is a commonality in that they express common ways of solving programming problems that are independent of the programming language although not necessarily of the programming paradigm.

The nature of a program, both in terms of the language constructs used and the algorithms / patterns to be implemented, is a key aspect in learning to program. The algorithms and patterns have to be refined and shaped for the particular task in hand and possibly to the programming paradigm and language.

Berglund & Lister (2007, 2010) also identify the variation of flow in control in their analysis of what *objects first* meant. They emphasise that "object interactions gives the algorithm." Object usage is described initially as passive and then as active. If passive usage

is the use of objects as in category N1 and N2, and active is seen as behavioural objects (N4 and N5) then there is a parallel between Berglund and Lister's "program execution" dimension and the results of this study. The results of this study divide Berglund & Lister's dimension "program execution" into "flow of control" and "object usage."

All of these studies indicate that there are significant differences between programming in the imperative paradigm and programming in an object-oriented paradigm. To help students transition between paradigms, it is necessary to address the dimensions of variation; that is, the learner needs to address the change in flow of control and the issues surrounding object usage.

Conclusion

The results of this study, through identifying the variations in the understanding of practitioners, have supported the view that there is a significant difference between imperative programming and object-oriented programming. The study has provided additional data that identifies critical aspects in these differences and provides a clarification of the hierarchical relationships. The hierarchy is defined based on the number of dimensions of variation and increasing relationships between the dimensions.

The study does not identify a sequence of teaching but it does raise questions in relation to the focus of teaching especially if the objective is to achieve a particular level of understanding within the hierarchy. Ehlert and Schulte (2010) have shown that introducing OOP first or later makes little difference to the learning outcome. Their study did not seek to foster conceptual change. If conceptual change is seen as the foundation of learning then this study recommends that the teaching strategy should focus on conceptual change with emphasis on flow of control as interactions between entities and objects as behavioural entities. There is also a corresponding need to revise the way that reusable logic or design ideas are introduced to match the conceptual change.

This study does not establish a relationship between a particular category of understanding and the nature and quality of the work produced. Other factors such as the practitioner's understanding of design characteristics (Thompson, 2008) may provide additional influences on the nature and quality of the work produced. It is recommended that further research is done to explore the relationship between practitioner understandings and their approach to using design patterns and frameworks.

From the data collected for this research, it is not possible to draw precise conclusions about the relationships between categories and novice or expert understandings. Indicators would suggest that technology experience and expertise does play a part in the likely categories that a participant may express but the data also shows exceptions to these relationships. A greater quantity of data targeted at identifying this relationship is required before any conclusions can be drawn.

Further research is recommended to verify the impact of teaching programming with an emphasis on conceptual change is required. It is also recommend that the study of practitioner understanding of the nature of a program be extended to include other paradigms and the conceptual change that would enhance the use of the paradigms.

Acknowledgements

The primary planning, data gathering, and phenomenographic analysis was performed with guidance from Associate Professor Janet Davies while the first author was working on his PhD at Massey University, New Zealand. This research is partially supported by NSERC, iCORE, Xerox, and the research related funding by Mr. A. Markin.

References

Beck, K. (1997). *Smalltalk best practice patterns* (1st ed.). Upper Saddle River, NJ: Prentice Hall.

Beck, K. (2007). *Implementation patterns*. Upper Saddle River, NJ: Addison Wesley.

Berglund, A., & Lister, R. (2007). *Debating the OO debate: Where is the problem?* Paper presented at the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007).

Berglund, A., & Lister, R. (2010). *Introductory Programming and the Didactic Triangle*. Paper presented at the Twelfth Eighth Australasian Computing Education Conference (ACE2010).

Booth, S. (2001). *Learning to Program as Entering the Datalogical Culture: A Phenomenographic Exploration*. Paper presented at the In 9th European Conference for Research on Learning and Instruction (EARLI), Fribourg, Switzerland.

Botvinick, M., & Plaut, D. C. (2002). Representing task context: proposals based on a connectionist model of action. *Psychological Research, 66*, 298-311.

Bowden, J. A., & Marton, F. (2003). *University of learning*. London: Routledge Falmer.

Brookshear, J. G. (2007). *Computer science: An overview* (9th ed.). Boston: Pearson Education Limited.

Burkhardt, J.-M., Détienne, F., & Wiedenbeck, S. (2002). Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *Empirical Software Engineering*, 7(2), 115-156. doi: http://dx.doi.org/10.1023/A:1015297914742

Clancy, M. J., & Linn, M. C. (1999). *Patterns and pedagogy.* Paper presented at the Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education.

Cope, C. (2002). Educationally critical aspects of the concept of an Information System. *Informing Science, 5*(2), 67-79.

Eckerdal, A., & Thuné, M. (2005, 27-29 June). *Novice Java programmers' conceptions of "object" and "class", and variation theory.* Paper presented at the ITiCSE'05, Monte de Caparica, Portugal.

Ehlert, A., & Schulte, C. (2010). *Comparison of OOP first and OOP later: first results regarding the role of comfort level*. Paper presented at the Proceedings of the fifteenth annual conference on Innovation and technology in computer science education.

Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1995). *Design patterns : Elements of reusable object-oriented software*. Reading, Massachusetts: Addison Wesley Longman.

Lewis, J., & Loftus, W. (2006). *Java software solutions: Foundations of program design* (5th ed.). Boston: Addison Wesley.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., et al. (2006). Research perspectives on the object-first debate. *Inroads - The SIGCSE Bulletin, 38*(4), 146-165.

Marton, F. (2000). The structure of awareness. In J. A. Bowden & E. Walsh (Eds.), *Phenomenography* (pp. 102-116). Melbourne, Australia: RMIT University Press.

Marton, F., & Booth, S. A. (1997). *Learning and awareness*. Mahwah, NJ: Lawrence Erlbaum Associates.

Marton, F., & Tsui, A. B. M. (Eds.). (2003). *Classroom discourse and the space of learning*. Mahwah, NJ; London: Lawrence Erlbaum Associates, Publishers.

Nguyen, D. Z., Ricken, M., & Wong, S. (2005). Design patterns for parsing. *Inroads - The SIGCSE Bulletin, 37*(1), 477-481.

Pennington, N. (1987a). Comprehension strategies in programming. In G. M. Olson, S. Sheppard & E. Soloway (Eds.), *Empirical studies of programmers: Second Workshop* (pp. 100-113). Norwood, NJ: Ablex Publishing.

Pennington, N. (1987b). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology, 19*, 295-341.

Ramsden, P. (2003). *Learning to teach in higher education* (2nd ed.). London: Routledge Falmer.

Rosson, M. B., & Carroll, J. M. (1996). Scaffolded examples for learning object-oriented design. *Communications of the ACM, 39*(4), 46-47.

Sajaniemi, J., & Kuittinen, M. (2007, 2-6 July). *From procedures to objects: What have we (not) done?* Paper presented at the 19th Workshop of the Psychology of programming Interest Group, University of Joensuu, Finland.

Sedlmeier, P. (2000). How to improve statistical thinking: Choose the task representation wisely and learn by doing. *Instructional Science, 28*, 227-262.

Thompson, E. (1992). *CBC-PP100 programming principles workbook*. Auckland: Carrington Polytechnic.

Thompson, E. (2008). *How do they understand? Practitioner perceptions of an object-oriented program.* Dissertation, Massey University, Palmerston North.

Thompson, E. (2010). *From phenomenography study to planning teaching*. Paper presented at the Proceedings of the fifteenth annual conference on Innovation and technology in computer science education.

Waddington, T. (2008, 23 November). Conception Leads Perception: On enhancing your powers or perception. http://www.psychologytoday.com/blog/smarts/200811/conception-leads-perception psychology/Waddington2008PT.pdf

Wirth, N. (1976). *Algorithms + data structures = programs*. Englewood Cliffs NJ: Prentice Hall.