Proceedings of 25th Annual Eastern Conference of the Consortium of Computing Sciences in Colleges (CCSC-E), October, 2009.

The Essence of Object Orientation for CS0: Concepts without Code

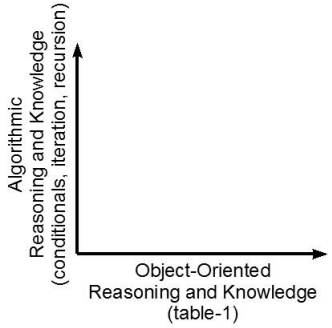
Raja Sooriamurthi
Information Systems Program
Carnegie Mellon University
Pittsburgh, PA 15213
raja@cmu.edu

ABSTRACT

Why is object-orientation so popular? Is it a fad or is there real value to developing software systems the object-oriented way? Given the emerging prevalence of computational thinking across the disciplines these are questions that a wide range of students are curious about. This paper describes our approach to providing a conceptual overview in a CS0 context of the essential ideas of and the value provided by object-orientation without resorting to code.

1 INTRODUCTION

For more than ten years the concepts of object-orientation have formed an integral component of the under-graduate curriculum. CS1 and CS2 courses introduce object-oriented programming with a variety of approaches[2, 5, 7, 10, 13]. While the pedagogical issue of teaching objects-early vs. objects-late is unresolved[11], object-orientation is well established as a predominant paradigm of software development. Why is object-orientation so popular? Is it a fad or is there real value to developing software systems the object-oriented way? If there is real value, what is it and how does object-orientation provide that value? Given the emerging prevalence of computational thinking across the disciplines[15] these are questions that a wide range of students—not just computer science majors but also engineering, information systems, science, business majors etc.—are curious about. This paper describes our approach to providing an overview of object-orientation in a CS0 context. Our approach is couched in an intuitive and qualitative discussion of the essential concepts of and value provided by object-orientation. These ideas are conceptually discussed sans code. Hence the discussion is accessible to students with a wide range of backgrounds and limited prior development experience. We have successfully used this approach to several audiences: college freshmen, high school participants in outreach programs, professional technology users groups, graduate students transitioning from other disciplines, birds-of-a-feather session at CS education conferences, participants of the Java Engagement for Teacher Training (JETT) workshops for high school computer science teachers [14] etc. Whereas tools such as Alice, BlueJ, Greenfoot, jGrasp, Dr.Java etc. help enormously in providing hands-on exposure to object-orientation, the focus of the approach we summarize in this paper is to provide a bird's-eye conceptual overview of the essential ideas of object-orientation to those new to the field.



Algorithmic and Object-Oriented Reasoning: While the focus of our discussion will be on object-orientation, algorithmic reasoning is the bedrock foundation upon which one builds software development skills. These two critical skills of algorithmic thinking and object-oriented thinking are interestingly in many ways related. Yet they are also orthogonal and complementary. Algorithmic reasoning focuses primarily on data and functional abstraction and how primitive computational elements may be combined

and composed by sequencing, alternation, iteration and recursion. Object orientation focuses more on identifying the nature of the problem, the main entities involved in the solution, their respective responsibilities and how they interact in a cooperative manner to obtain the computational solution [12, 16].

2 TWO FACETS OF SOFTWARE DEVELOPMENT

At the highest level, software development addresses two questions: (1) how to build the right system and (2) how to build the system right? Any paradigm of software development needs to support these two activities well. But what do these two activities entail?

Building the Right System: To build a *right system* means to build a system that meets the requirements of the client, i.e., the system does what a client would like the system to do and does it in the way the client would like it to be done. Perhaps the biggest challenge to meeting the requirements of a client is that rarely are the requirements predetermined and unchanging. At the onset of a project clients usually have a vague and ambiguous notion of what they want. As a result, during the construction of a software system requirements change—new requirements are added and existing requirements are modified. A good software development process needs to be able to effectively accommodate such changes.

Building the System Right: Apart from functional requirements (what a system should do) there are numerous non-functional requirements (also known as quality attributes) such as usability, reliability, performance, supportability etc[1]. Of these, over the life time of a system, the most expensive quality is maintainability. Estimates are that as much as 80% of the cost of a system of is in maintenance. But the word maintenance, as applied to software, is bit of a misnomer in that it is not like house or automobile maintenance—there is no wear and tear. Rather maintenance is about fixing bugs and more often it is about enhancing the functionality of a system to do things above and beyond what the system was expected to do when originally conceived. Any software development process needs to offer good support for the evolution of a deployed system.

In both of the above the common core is *managing change*. This can be done in two broad ways (i) reduce the need for change and (ii) when change is inevitable make it easier to effect. Reducing the need for change is partly a function of understanding the domain and modeling the system requirements better. An effective mechanism for mitigating the amount of effort that needs to be changed is to build systems in an incremental, iterative manner with continuous/regular feedback from the stakeholders[3]. But when change is inevitable it is desirable to reduce the impact of the change on the overall system. Change is initiated by the need to provide new functionality. There are two alternatives when we want enhanced functionality: (i) we need to modify pieces of the existing system or (ii) we don't modify existing pieces but just add new pieces to the system. A significant attraction of building systems the object-oriented way is that object-orientation offers good support for managing change in terms of building the right system and building the system right. The next several sections expand upon this in detail.

3 PROGRAMMING AS A PROCESS OF MODELING A WORLD

Object-orientation (OO) can help to reduce the need for change by helping us to better model the domain. Programming is a process of modeling a world. All paradigms of programming assist us with different aspects of this modeling process which is also the essential idea behind the process of abstraction[6, 8]. As a case in point, we discuss the very first killer application for PCs, the electronic spreadsheet. Starting with Visicalc, spreadsheets have modeled the information processed by accountants and the accounting processes they use.

More generally, we humans are good problem solvers. A significant attraction of object-orientation is that it tends to model the way we humans solve problems. Consider the way we get things done in this world: Either we have the ability to do a task or we ask someone who

instance	class	message	method
inheritance	delegation	polymorphism	interface
abstraction	encapsulation	overloading	overriding

Table 1: Some of the foundational concepts of object-orientation. Each of these concepts are introduced to students in an intuitive manner using examples from outside the realm of programming.

Symbolic Math	$\sum_{i=m}^{n} i$	Textual Math	`n`
			`sigma(i)`
			`i=m`

Functional	`sigma(m,n)`	Message sending	`m.sigma(n)`

Table 2: Four alternative notations for the same concept of summing an interval of integers.

has the ability to do it for us. For example, let us consider the following task: a professor wants to reserve a room to conduct an exam. Professors typically do not have the authority to make room reservations. Hence the professor would typically make a request to the department secretary. The secretary in turn forwards the request to the building manager who makes the room reservation. The reservation details are then sent back to the secretary and from there back to professor. We draw a parallel between that which we term as making a request in the real world and sending a message in object oriented parlance and emphasize that the way we do anything in a pure OO way is by sending a message to an object whose responsibility it is to perform that action. The response to a request (message) is a method. The set of messages a person responds to is termed the interface (e.g., sit, stand are typically part of our interface but fly isn't).

An anthropomorphic view of reasoning wherein inanimate objects are also deemed to have the ability to respond to requests helps to develop the OO thought process. The animation in Disney movie classics such as *Bedknobs and Broomsticks* and *Beauty and the Beast* helps to illustrate this notion of anthropomorphisation. In this manner we give our students an intuitive introduction to various concepts of object orientation some of which are listed in Table 1.

4 NOTATIONAL VARIATIONS AND THEIR SEMANTIC EQUIVALENCE

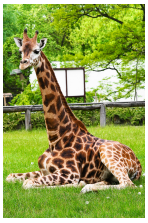
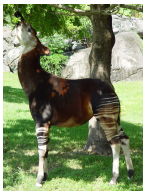
Barring a very few exceptions (CLOS being a prominent exception), most OO languages use the "dot notation" for denoting message sending. We have found this to be a useful context to discuss the power of notations in general and notational variations. Consider Table 2 which illustrates four ways of denoting the sum of all integers from `m` to `n` inclusive. Conventionally this summation is denoted in mathematics with the Greek letter Σ. But suppose we did not have the facility to graphically denote the Greek letter sigma (e.g., we were typing notes in plain text etc), then the textual math notation would meet our needs. Now suppose we had a further restriction on our notation system: we could only write linearly on a line. Then the functional notation would suffice. It is important to recognize that all three notations (symbolic math, textual math, and functional) are semantically equivalent—they denote the same value. Finally if we wanted to express the summation from `m` to `n` using the message sending notation of object oriented programming we get the last notation. In pure object oriented languages such as Ruby or Smalltalk one could actually express summation with this notation as in these languages integers are full fledged objects. We have found that appreciating the fact that the same concept could be denoted in multiple ways is a key step towards getting comfortable with the novel message sending notation of object-orientation.

As an interesting aside, it is useful to look at the linguistic typology and notations used in human languages. English is termed an S-V-O language as we write sentences of the form "Jack ate an apple" with the subject followed by the verb followed by the object. Many East Indian

languages (e.g., Tamil), Korean, Japanese etc., are S-O-V languages. It is quite interesting to note that there are human languages that fall into one of the six possible permutations of linguistic typology! The rarest order seems to be O-V-S with only a handful of human languages. Students find it amusing that the artificial language Klingon (from Star Trek) was specifically designed to be O-V-S so as to be very different from English.

5 OBJECT-ORIENTATION: METAPHORS AND ANALOGIES

In this section we give a sample of a few more qualitative overviews of some essential concepts of object-orientation.

The Power of Inheritance: A question we ask in class is "do you know what an *okapi* is?" Most students do not and wonder whether it is a place, a fruit, an animal etc? Once we mention that an okapi is like a zebra and a giraffe students immediately get a mental model. If we were to ask how many legs an okapi[1] has, a reasonable answer would be 4 (correct). If we were to ask what color is the tongue of an okapi another reasonable answer would be pink, which is wrong as it turns out an okapi has a blue tongue! A simple example of over-riding a default.

As an another example of the utility of organizing knowledge in an inheritance hierarchy and using default reasoning we discuss the psychological experiments of Collins and Quillian[4]. Participants of the experiment were asked a series of questions and their reaction times were noted. A sample set of questions were: can a canary sing? Can a canary fly? Does a canary have skin? The reaction time of the participants were progressively longer for these series of questions leading credence to the fact that information about canaries, birds, and animals seem to be organized in an inheritance hierarchy.

The Flexibility of Delegation: While inheritance is powerful and with very little effort one can gain a lot of information, its disadvantage is its static nature. The inheritance links are traditionally predetermined and not flexible. The inflexibility of inheritance is akin to being required to ask one's parents an answer to a question when one's uncle may be more of an expert on that particular topic! An alternative to inheritance is delegation. While in most programming languages it requires more effort to setup it provides more runtime flexibility. A good example of the flexibility of delegation is exhibited in the TV game *Who wants to be a millionaire?*. The contestant has the option of dynamically choosing during the game (i.e., at run time) to whom to direct a question.

Interface vs. Implementation One of the core tenets of design is *separation of concerns*. In object-orientation it is useful to separate what we can ask an object to do (its interface) and how the object actually does what is asked of it (its implementation). What is the advantage of this separation in reasoning? In short: it fosters resilience to change. An interesting analogy of this resilience is contrasting the TV shows Law & Order and Seinfeld. Currently in its 19th season, Law & Order has had an interesting evolution. Over its long and successful run the show has changed: various actors have come and gone, but the popularity of the show has been undiminished. The audience tunes in for the roles played by the different actors (district attorney, police captain, detective, prosecutor etc.) and not necessarily the people who play the roles. Law & Order is a show programmed to its interface and is only loosely coupled to its implementation. As a result of which the show has been flexible and resilient to change. But a show such as Seinfeld is programmed to its implementation. The audience tunes in for the actors and hence the show would probably handle change less gracefully.

Overloading: Words in human language are overloaded i.e., the same word takes on different

[1] An Okapi is a peculiar animal discovered in the Congo rain forest only in the early 1900s. Amongst other oddities it is the only animal with a tongue long enough to lick its own ears http://en.wikipedia.org/wiki/Okapi.

meanings in different contexts. The same request (message) can result in different responses (method) depending on the context. This notational convenience increases the usability of natural language. Consider the request open. Based on the context, the actual action executed is different. The same message has different methods. For example "opening" a door, a can of soda, or a pen are different actions. If our use of human language was not overloaded we would have to concoct unique words for each type of opening e.g., dopen, copen, popen! Object-orientation provides a similar convenience in the context of determining method names.

Responsibility Alignment: One of the most important skills of an OO designer is to properly align responsibilities with the objects that will perform them [9, 12, 16]. Misaligned responsibilities can lead to contorted systems. Two interesting examples of responsibilities that were re-aligned are (i) The introduction of penny postage by Sir Rowland Hill.[2] When postage was first introduced the recipient had to pay which led to many problems. The postal reform of 1840 switched the responsibility of paying for postage from the recipient to the sender leading to our current system (ii) in the early days of telephony the responsibility of actually placing a phone call was with a telephone operator. Subsequently this shifted to the call initiator via the introduction of the rotary dial and subsequent automatic exchanges which in turn led to the mass scaling of the phone system.

In a similar manner we use various analogies and metaphors for conveying the essential idea behind other OO concepts (Table-1) such as polymorphism, encapsulation, the Java construct of an `interface` (which is related to but slightly different from the more general notion of an interface) etc. (Due to space limitations in the paper we are not discussing them here.)

6 IT IS A MATTER OF TIME — BUT WHOSE?

The time it takes to solve a problem is a valuable resource. Over the past couple of decades there has been an interesting shift as to which time is more valuable. In the early days of the field computer processing time was expensive. Hence the time measured was the interval between when the program was given to the computer and when it finished running. Now the more valuable resource is the programmer's time. Hence what is being measured is the duration of when the problem is given to the programmer and when the programmer develops the correct solution. Yet another reason for the popularity of object-orientation is that it contributes towards the productivity gains in programmer development time in terms of reuse, customization, frameworks etc. Languages like Ruby and Python add an extra dimension of productivity on top of their object-oriented features by being dynamically typed and thereby enhancing the productivity of a developer even more, albeit at a cost of execution speed. Then again, the critical question is not whether the program is fast, but whether the program is fast enough.

7 SUMMARY

While there are a number of reasons why object-oriented programming is a current dominant paradigm of software development, two prominent reasons are that object-orientation helps us to (1) better model our domain and (2) better manage change during the evolution of the system and after its deployment. We've discussed how we provide a conceptual 10,000 foot view of the essential ideas of object-orientation that is independent of any particular OO language. How effective is this approach? At present we have anecdotal feedback to support our approach. In both class-room as well as professional settings where we have delivered this content we have had very good interaction and feedback. To our amusement (and delight) we also see that some of our students have been using these metaphors and analogies during technical interviews to the mirth of their interviewers.

[2] Actually based on an analysis of the British postal system by computer pioneer Charles Babbage.

REFERENCES

[1] BASS, L., CLEMENTS, P., AND KAZMAN, R. *Software Architecture in Practice*, 2nd ed. Addison-Wesley, 2003.

[2] BRUCE, K., DANYLUK, A., AND MURTAGH, T. *Java: An Eventful Approach*. Prentice Hall, 2005.

[3] COCKBURN, A. *Agile Software Development: The Cooperative Game*, 2nd ed. Addison-Wesley, 2006.

[4] COLLINS, A. M., AND QUILLIAN, M. R. Retrieval time from semantic memory. *Journal of verbal learning and verbal behavior 8* (1969), 240–248.

[5] DANN, W. P., COOPER, S. P., AND ERICSON, B. *Exploring Wonderland: Java with Alice and Media Computation*. Prentice Hall, 2009.

[6] HAZZAN, O. Reflections on teaching abstraction and other soft ideas. *SIGCSE Bulletin 40*, 2 (2008), 40–43.

[7] KELLEHER, C., AND PAUSCH, R. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys 37*, 2 (June 2005), 83–137.

[8] KRAMER, J. Is abstraction the key to computing? *Commun. ACM 50*, 4 (2007), 36–42.

[9] LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Prentice Hall PTR, 2004.

[10] LEWIS, J., AND DEPASQUALE, P. *Programming with Alice and Java*. Addison-Wesley, 2008.

[11] LISTER, RAYMOND, E. A. Research perspectives on the objects-early debate. In *ITiCSE proceedings* (2006), pp. 146–165.

[12] MEYER, B. *Object-Oriented Software Construction*, 2nd ed. Prentice Hall PTR, 2000.

[13] REGES, S., AND STEPP, M. *Building Java Programs: A Back to Basics Approach*. Addison-Wesley, 2007.

[14] SOORIAMURTHI, R., SENGUPTA, A., MENZEL, S., MOOR, K., STAMM, S., AND BÖRNER, K. Java engagement for teacher training: An experience report. In *Proceedings of the Frontiers in Education Conference* (2004).

[15] WING, J. M. Computational thinking. *CACM 49*, 3 (March 2006), 33–35.

[16] WIRFS-BROCK, R., AND MCKEAN, A. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley, 2002.