Abstraction the key to CS1

Peter Sprague & Celia Schahczenski
Montana Tech of the University of Montana
Computer Science Department
1300 West Park Street
Butte, MT 59701
(406) 496-4383
psprague@knosysinc.com / cschahczenski@mtech.edu

Abstract

It is commonly agreed that a first-year programming class should not so much be a class in programming as a class in problem solving. In this paper we argue that problem solving is better taught using the object-oriented paradigm than the traditional procedural programming paradigm. We argue that abstraction is key to problem solving and that object-orientation facilitates, and even forces, a higher level of abstraction than does procedural programming. We discuss the benefits of abstraction and those topics that should be presented in a first year "objects-first" programming course.

Abstraction the Key to CS1

Abstract

It is commonly agreed that a first-year programming class should not so much be a class in programming as a class in problem solving. In this paper we argue that problem solving is better taught using the object-oriented paradigm than the traditional procedural programming paradigm. We argue that abstraction is key to problem solving and that object-orientation facilitates, and even forces, a higher level of abstraction than does procedural programming. We discuss the benefits of abstraction and those topics that should be presented in a first year "objects-first" programming course.

Introduction

Linda Wilkens, in listing essential concepts for computer science students, ranks abstraction first. [1] Abstraction is “the act or process of separating the inherent qualities or properties of something from the actual physical object or concept to which they belong.” [2] By “abstracting” we can use a general idea or word to represent a physical concept. In this paper we support the current academic trend of migrating the first year programming course to the object-oriented paradigm because we believe that this paradigm forces a higher level of abstraction than the procedural paradigm. We argue that object-orientation enables better programming instruction by encouraging the use of

scalable, real world problem solving skills such as abstraction, encapsulation and modular design.

We begin this paper with an historical discussion of the transition to object-oriented programming. Next we describe the benefits of using abstraction within and outside the field of computer science. We examine how object-oriented programming teaches and reinforces abstraction skills and review some problems that may arise from teaching programming with emphasis on syntactic constructs. We end with a list of what concepts belong in an “objects-first” introductory programming course and present some effective programming examples for such a course.

Historical Perspective and Trends

Abstraction and symbol manipulation are fundamental to computer science. The complexities involved in executing the simplest program could never be managed without layers and layers of abstractions. Back in 1968 Dijsktra argued for structured programs rather than the unstructured "spaghetti" code. [3] Structured programs are divided into functional pieces that may in turn be divided into smaller functional pieces. This division allows the programmer to conceptualize the whole program as a sum of a limited number of abstract pieces. In contrast, an unstructured program has no layers of abstraction separating the program from the individual commands.

In a similar vein, object-orientation allows increased abstraction over procedural programming. In the procedural paradigm programmers create functional abstractions, called procedures, of the tasks the computer is to perform to solve a problem. Procedures

can be nested within procedures creating abstractions for larger portions of the code. While dividing a problem into functional abstractions is useful, software consists of data and functions. In procedural programming the only data abstractions the developer has available are those provided by the native data types of the language and structures aggregating those native types.

Object-orientation, on the other hand, allows programmers to create program structures that combine functions and data - thus creating abstractions based on "things". Bundling software components in this way aids in the creation of powerful software abstractions. The extent to which the functionality of a software “object” aligns with the functionality of that in the physical world that the object is meant to mirror, shows the strength of the abstraction.

This bundling of functions and data has a further advantage. Exchange between software components (or objects) can be seen as client-server relationships where one component (the client) wants information that the other component (the server) possesses. Joseph Bergin argues: " In procedural programming, with only compile-time dispatch, the client always chooses the specific service and thus in its implementation -- the client is in control. In object-oriented programming the server is in control. Server control is more appropriate since if the client must choose then it requires inappropriate knowledge about the implementations of the various services that might be used." [4] Thus when object-orientation is applied wisely, strong encapsulation may be achieved resulting in better abstractions.

As the benefits of object-orientation became evident, industry began calling for students trained in object-orientation. In response the academic community began experimenting with the introduction of object-orientation into its curricula. Panel discussions “When To Object – Introducing Undergraduate Students To Object Oriented Concepts” [5] and “Objects: When, Why, and How?” [6] were held at previous CCSC conferences and numerous papers addressed the subject. In “Lessons Learned In Teaching Object-Oriented C++ Programming in CS1” Ghafarian gives an excellent overview of several of these papers and recommends waiting until the second semester of CS1 to introduce objects. [7] We disagree, believing as Wolz says "objects are useful in the first courses because they can help students organize their thinking." [8].

One difficulty of teaching objects-first is that most computer science professors learned to program procedurally and only learned object-orientation afterwards. This means that most professors must teach programming differently than how they were taught. This paper's focus on abstraction is meant to help professors in their search for new ways to teach introductory programming courses with object-orientation.

Benefits of Abstraction

Abstraction is key to problem solving. It is arguable that all problem solving involves some form of abstraction. Through abstraction problem solvers reduce a complicated idea to a simpler concept that contains only the details relevant to the problem. Being able to understand what details are important to the problem is a difficult skill that requires considerable practice.

Abstraction is used throughout the field of computer science. Mid-level and high-level languages provide an obvious example. These languages aid computer scientists in solving problems. The software developer describes a solution for a problem using the abstraction of the programming language. The details of translating the solution into a form that the computer can act upon are left to the compiler and assembler. Classic algorithms such as *binary search* or *bubble sort* provide another example. Computer scientists choose and apply these algorithms focusing only on those aspects of the algorithm that are relevant to the questions they are answering. These algorithms are highly effective abstractions for software development.

Abstraction is also applicable to non-computer science disciplines. Examples include physics experiments conducted in hypothetical *frictionless vacuums*, chemistry's *ideal gasses* and the economist's supply and demand in a *perfect market*. Students not majoring in computer science but taking CS1 may have more appreciation for, and may benefit more from, understanding and being able to use abstraction than understanding and being able to use programming constructs such as loops and selection. By emphasizing abstraction as a universal problem solving technique, students from other disciplines may be more motivated to learn the material in CS1.

Understanding that computation is merely symbol manipulation, and that the power of computers is predicated on a tremendous amount of abstraction, is crucial in

understanding what computers can, and can not, do. Both computer science and non-computer science majors should appreciate this.

Teaching Abstraction through Object Oriented Programming

After several years of teaching object-oriented programming, it has become clear that a tremendous amount of knowledge must be mastered by the student in order to understand even a basic "hello world" program. In acquiring this elementary background, however, students are developing skills that they will need to become successful problem solvers throughout the remainder of their careers. In other words, the skills that are taught in an quality object-oriented programming course "scale-up." The skills that are taught to the novice writing that first "hello world" program should be the same skills that computer professionals use when developing million-line systems.

When using object-orientation the structure of the software system mirrors that portion of the real world that the software is meant to facilitate. Thus, the students' previously acquired knowledge helps them in writing their first programs. For instance students writing an airline reservation system can begin to identify entities (flights, seats, customers) and identify characteristics of those entities (flight number, departure location, departure time, seat number, customer name, etc.) that are relevant to the problem. The classic object-oriented design goals of information hiding, loose coupling and high cohesion start to become relevant to the student at this early date.

In addition, students will be familiar and comfortable with software objects from their previous experience as computer application users. Many familiar applications are object-

oriented. Word processors contain strings, fonts, and embedded data objects (OLE). Spreadsheets contain cells, columns, rows and graphs. Games contain tokens, obstacles and monsters. The most fundamental structure of these applications can be easily understood when expressed as objects. In contrast, implementing an equivalent program procedurally requires a great deal of programming specific knowledge.

Second, data flow through the system is more intuitive in the object-oriented paradigm than in the procedural paradigm. The data will move through an object-oriented system in a way similar to how data flows through the real world. Again students can use their knowledge of the real world to determine how the software needs to be written.

Third, in the object-oriented paradigm a program is broken into classes that have data and method members. Students implementing these classes quickly see the advantage of having a design for the class before they begin typing. Students see the advantage of design early. They begin to understand through experience why one design may not be as good as another design.

Finally, there are libraries of components that first time programming students can immediately use. This use of abstraction not only allows them to understand how large programs are pieced together, but also creates a reason to read code before writing it. Learning to read code before writing it has been proposed as a method for teaching programming, but is rarely carried beyond small code samples. Reading class libraries gives students examples of the abstractions made by the library authors and, if presented properly, may even give them insight into the library authors' design decisions. Learning

to read code before writing it also motivates good documentation, not only for the sake of others understanding the code, but also for spelling out what the code is to do before thinking about how to do it.

Problems with syntax focused instruction

A programming language serves as a means to teach how to program a computer and enables students to experiment with programming and enjoy direct feedback from the machine. The language should not, however, become an end to itself. The popularity of Pascal in the late 70s and early 80s demonstrates this. Many people learned programming with Pascal, not because it was a useful language for industry, but because it was a good language for teaching programming concepts. Many schools have begun teaching CS1 in Java. This is not because Java is becoming a language of choice by industry, but because Java is similar to C++ (a current language of choice for industry) with fewer pitfalls, fewer, and possibly cleaner object-oriented constructs, and an attractive, easy to use, graphics library.

By focusing upon abstraction rather than syntax, CS1 can avoid misconceptions of computer science and programming frequently made by the student. Teaching in a procedural language often focuses upon syntactic structures. Teaching focused upon syntax may inadvertently emphasize less-relevant aspects of computer science and can lead the student into common pitfalls. Object-orientation can diminish:

- Memorization – students tend to memorize syntactic constructs rather than understanding how the program construct solves the problem.

- Language Dependency – Students tend to compartmentalize their programming skills by programming languages. They do not appreciate the importance of problem solving skills independent of a language.
- Skill Obsolescence – As technical artifacts programming languages have life cycles. Students trained via syntax-focused instruction will have to be re-trained when that language is no longer required.
- Vocational Emphasis – Instruction focused on syntactic constructs tend to teach towards vocational goals rather than educational ones. An introductory computer science course should prepare the student to learn a broad spectrum of computer science related course material. It should not focus on preparing the student for a specific job function upon entering industry.

Teaching objects-first

How should our introductory programming courses change to accommodate object-orientation? Object-extraction should be taught and used continually throughout the course. Considerable explanation of the design process should take place before the class becomes mired in implementation details. The instructor could code a design developed by the class so that early in the first semester the students can see the relation between design and code. A later lab could divide the students into groups that create a problem design. The groups would then present their designs to the class and discuss the advantages and disadvantages of different design decisions.

The software development life cycle should be taught early to motivate the use of objects. An investment of class time in the design and implementation portions of the

development cycle can allow students to work with larger programming examples. Larger example programs are needed to demonstrate the organizational complexity and reuse benefits of object-oriented programming. This investment of class time also demonstrates the importance of planning and design to students.

We present an ordered breakdown of topics for the first and second semester of an objects-first introductory programming course. We present this not as a completed masterpiece, but as a starting point for discussion.

First semester:

1. Introduction: Components of a computer, areas of computer science
2. Software development cycle
3. Client-server model and message passing
4. Software design: information hiding, high cohesion and loose coupling
5. Encapsulation
6. Object extraction and definition
7. Identifiers
8. Data types, variables, literals
9. Syntax for class definition & method definition
10. Edit-compile and run cycle
11. Syntax for data declarations & expressions
12. Syntax for object creation, method invocation & parameter passing
13. Syntax for simple input & output
14. Libraries (possibly cover the ‘help’ facilities here as well)

15. Inheritance and visibility modifiers
16. Accessor, mutator & constructor methods
17. Sequencing, selection and repetition
18. Hand tracing an algorithm
19. Introduction to Inheritance

Second semester:

1. Exception handling
2. Arrays
3. Truth tables.
4. Evaluating complex Boolean statements
5. Constructors
6. Ways to communicate information between methods (instance variables, passing parameters, placing the information into another object)
7. Inheritance versus containment
8. Internal data representation
9. Recursion
10. File input and output
11. Sorting algorithms and algorithm complexity
12. Queues and stacks
13. Linked lists and binary trees
14. Event driven programming
15. Introduction to Polymorphism

Effective object-oriented examples

The benefits of object-orientation are less apparent in small programs than they are in large ones. Instructors teaching object-oriented programming need to have a slate of examples that use few programming statements but demonstrate advantages of object-orientation. In "Using Children's Songs to Teach Abstraction Techniques In An Introductory Programming Course" Snyder describes assignments where students are asked to display the words to the songs "Mary had a Little Lamb" and "Old Macdonald had a Farm." [7] The students are to note redundancies in the words to the song and to avoid these redundancies in the code by calling suitable print statements multiple times. These problems can be given even before assignment statements have been covered.

Here are other possibilities:

- A *Dog* class with the member element *bark* ("Bowwow", "Arff Arff", "Grrr") and functions *loud_bark* ("BOWWOW!", "ARFF ARFF!", "GRRR!"), *soft_bark* ("bowwow", "arff arff", "grrr") and *repeated_bark* ("bowwow bowwow", "arff arff arff arff", "grrr grrr").
- A *Stick_Figure* class with member elements *x_position*, *y_position*, *hand_position* ("out", "up" or "down") and *facial_expression* ("smiling", "frowning" or "flat"). This class could have the member functions *move_up*, *move_diagonal*, *change_arm_position* and *change_facical_expression*.
- A *Song* class with member elements *artist, title, length, year, album*. This class would have the member function *display_song* as well as accessor and mutator member functions for each element. This class would then be used by the student to

create an *Album* class that contain several *Song* members elements. The *Album* class would contain the member functions *add_song* and *display_album*.

Conclusion

We have argued that programming is best taught using the object-oriented paradigm because it enables students to use higher level abstractions and hone scaleable skills that they will use throughout their careers. Finding appropriate abstractions is crucial to effective problem. We hope that our breakdown of topics for an introductory course will spark further discussion of what should be included in a first year programming course.

References

1. Wilkens, Linda in panel discussion "What Concepts of Computer Science are Essential for Students Entering the Field." *Proceedings of the Fourth Annual CCSC Northeastern Conference*. Volume 14, Number 4, Page 225. May 1999.
2. Morris, William Editor, *The American Heritage Dictionary of the English Language*. American Heritage Publishing Co., Inc., New York, 1971
3. E.W. Dijkstra, "GO TO Statement Considered Harmful," *Communications of the ACM*, 11 (1968) pp. 147-148.
4. Bergin, Joseph in panel discussion "Objects:When, Why, and How?" *Proceedings of the Fourth Annual CCSC Northeastern Conference*. Volume 14, Number 4, Page 217. May 1999.
5. "When to Object - Introducing Undergraduate Students to Object Oriented Concepts" panel discussion in *Proceedings of the Third Annual CCSC Northeastern Conference*. Volume 13, Number 5, Pages 2-4. May 1998.

6. "Objects:When, Why, and How?" panel discussion in *Proceedings of the Fourth Annual CCSC Northeastern Conference*. Volume 14, Number 4, Pages 216-219. May 1999.
7. Ghafarian, Ahmed, "Lessons Learned in Teaching Object-Oriented C++ Programming in CS 1." *Proceedings of the Thirteenth Annual CCSC Southeastern Conference*. Volume 15, Number 2, Pages 278-286. January 2000.
8. Wolz, Ursula in panel discussion "Objects:When, Why, and How?" *Proceedings of the Fourth Annual CCSC Northeastern Conference*. Volume 14, Number 4, Page 219. May 1999.
9. Snyder, Robin, "Using Children's Songs to Teach Abstraction Techniques in an Introductory Programming Course." *Proceedings of the Twelfth Annual CCSC Southeastern Conference*. Volume 14, Number 2, Pages 278-286. January 1999.