Polymorphic Panelists

Byron Weber Becker, Moderator
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
bwbecker@uwaterloo.ca

Joseph Bergin
Pace University
One Pace Plaza, NY NY 10038
berginf@pace.edu

Richard Rasala
College of Computer Science
Northeastern University
Boston, MA 02115
rasala@ccs.neu.edu

Christine Shannon
Centre College
Danville, KY 40422
shannon@centre.edu

Eugene Wallingford
University of Northern Iowa
Cedar Falls, IA 50614
wallingf@cs.uni.edu

Summary

Polymorphism is an important object-oriented programming concept in which objects from two or more different classes respond to the same set of messages. For instance, `HourlyEmployee`, `SalariedEmployee`, and `ContractEmployee` all respond to the message `calculatePay()`. Instances of each class "do the right thing" to calculate their pay even though the methods to do so may be quite different. But the payroll program using these classes doesn't care – it can ask each object for the amount owed without caring what kind of employee it represents or how the amount is calculated.

The panelists are all instances of subclasses of `Professor` which will respond to the following queries. Since each of the subclasses implement these queries differently, the answers will usually be different as well!

- `polymorphPreconditions()`: The object (professor) specifies the information students must know before polymorphism is introduced in their class.
- `polymorphPresentation()`: The object (professor) describes how polymorphism is introduced in their class.
- `polymorphStudentUsage()`: The object (professor) describes how their students use polymorphism later in the course.
- `answerQuestions()`: The object (professor) responds to any questions about their approach.

Position Statements

Byron Weber Becker, Moderator

We use Karel the Robot early in our CS1 course to illustrate instantiating objects, invoking methods, extending classes, and more. After leaving Karel to explore other topics, we return to Karel late in the course to explore polymorphism. We create two very simple kinds of "dancing" robots, a "Left Dancer" and a "Right Dancer" and explore as a class what happens to them in various circumstances which build up to illustrate polymorphism. The lecture concludes with examples from the "real world" where polymorphism is useful.

Joseph Bergin

Since dynamic polymorphism and not the class concept is the big idea of object-oriented programming, the Early Bird pedagogical pattern implies that this should be taught first. However, since it is normally thought of as a difficult topic, some other pedagogical patterns come into play here, especially Spiral, Lay of the Land, and Stealth Instructor. The latter implies that, even if you don't specifically mention polymorphism early, all of your materials are designed to both reinforce it and to make its eventual understanding easy and natural. One way to do this is to emphasize the client-server nature of OO programs and the fact that each object "owns" its own code and can have no action imposed upon it. Another is to make sure that most of the decisions in the (many) programs you show your students are not done with ad-hoc polymorphism through selection statements. A Lay of the Land exercise or example can also show dynamic polymorphism in action and can show how decision making can naturally be done dynamically. Finally, a Spiral approach leads you to discuss polymorphism

repeatedly throughout the course with deeper explanations and understanding each time.

The bottom line is that everything you do should reinforce polymorphism and nothing you do should lead the students to program naturally in any other way. This takes great attention to your teaching and especially to your examples and exercises. You will also need to look out for ad-hoc decision making in your student programs and suggest better alternatives when you find it.

Richard Rasala

Polymorphism enables the fundamental computer science concepts of abstraction and encapsulation to be used in a far more powerful way than is possible in a procedural language. Multiple objects with different internal structure can invoke method calls that look identical. The method calls name a behavior by its purpose not by its implementation.

The Java Power Tools are a GUI building toolkit designed for use by first-year students. The tools maximize abstraction and encapsulation so that beginners can create GUIs quickly and without hassle. At an upper level, the tools may be used as an extensive case study in an object-oriented design course.

The panel presentation will show how Java Power Tools can be used as a case study in an Object-Oriented Design class. Students who have already used polymorphism in their programs will get a powerful lesson about the different ways of implementing polymorphic behavior and the design methods that can be used to identify where polymorphism would benefit the program design. The issues discussed here represent a capstone lecture rather than a first introduction to polymorphism. The talk will attempt to illustrate how much the ideas of interfaces, inheritance, reflection, factories, actions, and properties can deliver in object-oriented development.

Christine Shannon

At Centre, polymorphism receives attention in CS II which is currently taught in C++. From the very beginning of CS I, I point out that the plus sign is used to add integers as well as floating point numbers. In the second course we develop classes to be used with the standard data structures such as the ordered list. I teach them how to overload operators such as "<" so that we can control how the items in the list are ordered. A typical example would be to maintain a list of student names ordered by last name, first name, and middle initial.

The project for the course is written in stages over the course of the entire term. Near the end I introduce a major change which is intended to make use of polymorphism. For example, one year we did a library circulation system dealing only with books. I then introduced video tapes and CD-ROMs, each with their own data, check out periods, and fines. Students see first hand how easily these can be incorporated into the system (if it has been designed appropriately) by the use of polymorphism.

Eugene Wallingford

Dynamic polymorphism is the most important ingredient in writing extensible, adaptable programs. Our students first encounter polymorphism in a sophomore-level course on object-oriented programming. Initially, students use polymorphic objects without having studied the abstraction. They use classes that implement a common interface and then write their own classes for use in the same application.

At about the seventh week of the course, we begin to discuss polymorphism, its benefits, and its costs. In particular, students encounter polymorphism in the context of several common design patterns: substitution, recursion, decorator, and adaptor. In each of these cases, students learn how the use of polymorphism allows the programmer to resolve thorny design problems in an effective way. These class sessions rely on a mutual reinforcement between the patterns and polymorphism: understanding something about polymorphism helps students to learn the patterns, and understanding something about the patterns helps them to learn how to use polymorphism.