Pedagogy and Practice of Design Patterns and Objects First:
A one-act play

Carl Alphonce
Department of Computer Science & Engineering
University at Buffalo, State University of New York
Buffalo, NY 14260-2000
alphonce@cse.buffalo.edu

Dramatis Personæ
A, the apprentice
M, the master

ACT I
SCENE I
A and M are colleagues at a university. A has recently started teaching the Java-based introductory computer science course, CS1, “Introduction to Object-Oriented Programming”. A often chats with M. The scene opens with A’s head poking into M’s office.

A (dejected): Hi M, are you busy?
M (observing A with concern): Hi A! Busy? Yes. Too busy? Of course not. What’s up?
A (relieved): CS1 will be the death of me and my students!
M (frowning): Why’s that?
A: The course is too difficult for the students! They just don’t get OO. And they’re unmotivated to boot!
M (with understanding): Wow! It sure sounds like you have a challenge on your hands. What are the students having a hard time with?
A: It seems like everything is difficult for them! (A sighs) Well, maybe not everything. I must admit they seem to grasp some concepts. They do understand the basics like objects and messages now.
M: So what’s the problem?
A: They have all these little pieces, but they don’t see the big picture. They aren’t yet able to apply the concepts they’ve learned to solve any problems I give them. I thought introducing design patterns would help them with their design issues, but instead they just stare blankly. I’m not sure what to do! What would you suggest?
M: Hmm. Let me think …

M thinks for a moment, all the while looking around the office.

M (pointing to a calculator on the desk): I know! Why don’t you have them build a calculator? They’ve all used calculators – they’ll all be familiar with them.

A (picking up M’s programmable scientific calculator from the desk): A calculator – like this? That’s far too complicated – my students can’t handle that! Look at all these buttons – you’re not serious. I just told you that they aren’t able to solve the problems I give them, and you expect them to solve even harder problems? I need to give my students simpler examples!

M (gently relieving A of the calculator): Don’t worry, I heard what you said. I wouldn’t expect them to come up with anything quite this complicated to start with. Let me explain. You do know who Kristen Nygaard is, right?
A: Sure – he and Ole-Johan Dahl laid the foundations of OO with their work on Simula in the early 1960’s.
M: Good, good. Kristen always said that to teach OO you need to use sufficiently complex examples. His point was, as I understand it, that OO is for programming in the large, and in order for the use of OO techniques to be justified we must apply them to examples which have enough structure. In other words, to be effective examples must not be simplistic (there’s no interest in such examples for the students) or gratuitously difficult (students will just think you’re mean). I like to give my students simple examples with lots of potential for further development. How about starting them with a simple four-function calculator?

A (skeptical): Addition, subtraction, multiplication and division you say? What about all the graphics? I’ll have to spend days just teaching them how to get a button up on the screen! There’s just too much overhead – can’t be done.

M: How long would it take you to put a button up on the screen?
M (with a mischievous smile): Not all that long – but what does that have to do with anything? I can do it in just a few minutes because I know the Java graphics library inside and out. My students don’t. They’d have learn all about containers and components and events and event-handling and ...

M: Look, it’s all about presentation and focus. You need to decide what you want the students to focus on. If you won’t want the students to focus on the graphics then don’t let them.

A (exasperated): Sigh. I know you. There’s a lesson in here somewhere. Why don’t you just come out and tell me what it is?

M (with a mischievous smile): Oh, come now, you know better than that! If I just tell you then you won’t learn! Tell me, what do you want your students to learn?

A: I still think you’re being difficult with me. You suggested the calculator as an exercise – you obviously have something in mind. At least point me in the right direction.

M: I chose the calculator as a basis for an exercise because you can use it for so many different lessons. I think you’re absolutely right that the graphics library in its raw form is far too complicated for students to use at this point in the semester.

A: OK – so I should get them to do some simple arithmetic expression evaluation but have them build a calculator with a text-based user interface, so that they can learn about...hmmm...primitives?

M: That wouldn’t be my choice. I’d keep this exercise as much about object-orientation as possible. You also know how I feel about text-based user interfaces, don’t you?

A: Yes, how can I forget? Why did I forget? You like to quote Ian Utting. Why, you even have the printout of his e-mail to the SIGCSE list here on your desk:

In the olden days, when I learnt programming, printing out a triangle made of asterisks (on the ASR-33 console) was a significant achievement. Today’s students, bought up on cinema-quality shoot-em-ups, are much less impressed.

M: Right. A calculator may not quite compete with “cinema-quality shoot-em-ups”, but it’s real to them – they use on-screen calculators all the time.

A: So you’re suggesting I provide the students with a graphical user interface and have them build the calculator internals.

M: Exactly! By providing them with a framework which handles the details irrelevant to your pedagogical goals you free the students to focus on what you deem is important, while achieving something in the end that is more impressive, interesting and significant than they could have done on their own if they were working from nothing.

A: That’s cheating a little bit, isn’t it?

M: It’s not really cheating – after all one skill today’s students need to have is the ability to work with code that someone else wrote. There’s a pedagogical pattern called “Fill in the blanks” which talks about this sort of thing. There’s also the idea of using frameworks more generally, like JFC, to provide better base abstraction to work from.

A: That’s true! I like it: we’re catching two butterflies with one net. We focus students on the task at hand while giving them experience working with pre-existing code.

M: So, the first lesson learned is that you need to provide enough of a framework to allow students to focus on what’s important. The second lesson learned is that today’s students aren’t as easily impressed as we were when we went to school. We have to engage the students in terms which are meaningful to them.

Now we need to figure out what you want the students to learn by working on the calculator example.

A: I’ve been thinking about that. If I provide the students with the graphical user interface it is actually quite easy to motivate the separation between model and view, thereby introducing Model-View-Controller.

M: That’s a good start. You’re motivating the use of a design pattern by showing its application in a rich enough environment where its use is appropriate.

By providing the GUI you can guide them into maintaining the model-view separation naturally, and then have a discussion about the design after the exercise is completed.

Furthermore, you’re grounding the example in something that’s real, and therefore accessible, to the students.

A: You mentioned starting the students off with a small problem, implying that we can build on this example over time, making it gradually more complex.

M: Right. But I suggest you have a clear path determined ahead of time because the example is so rich you might otherwise end up with students in too many different places. Think about how you want to develop the example, then guide them to the right place.

A: OK – I’m going to think about what you said, and build my own solution. Right now, I’ve got to run to class.

M: Good idea. Make sure you’re thinking OO when you do!

A (looking at the time): What do you mean?

M: You’ll figure it out. But here’s a challenge: write the calculator without using a single if statement or switch statement.

A: Come on – that’s impossible! The calculator needs to make decisions to handle things like operator precedence, and to detect a divide-by-zero condition.
Ah, but it is possible, at least for most of what you’re doing. You can certainly manage operator precedence and divide-by-zero without using an explicit conditional statement.

I’ll give it a shot. Thanks for your help. Gotta run!

A exits from M’s office. M sits back in his chair and ponders how he would solve the calculator problem. M logs out, and as his screensaver kicks in the lights fade to black.

SCENE II

A few days later A and M are once again chatting in M’s office.

So you spent some time thinking about the calculator problem. But tell me, did you come up with a solution yourself?

Yes but you know, it took me longer than I thought, and its your fault!

My fault?

Well, not entirely. Building a calculator is not hard, but you said I should “think OO”, just like I’m wanting my students to do. I didn’t learn to program in an OO way. I find myself falling into old habits. Not so good when I’m trying to teach OO to my students! Like you already knew, I now realize I have to “think OO” myself to do a good job teaching the students.

I sat down with a clean sheet of paper and thought about the problem, sketched UML class diagrams, investigated object interactions ... all those things I want my students to do, but for some reason thought I didn’t have to worry about myself. So I mulled the problem over for quite a bit, and now I see how the polymorphic behavior of objects used by the calculator obviates the need for if and switch statements.

Go on.

I realized that using polymorphism I can construct different objects which behave in a manner appropriate to what they are. (pauses) Wait, that sounds confusing. Let me use an example to make things concrete.

A friend of mine really likes animals, and has quite a few pets. She has a dog, a bird, a snake, a monkey and a fish.

Your friend isn’t the zoo-keeper, by any chance?

You’d think so, wouldn’t you! But no, she just likes having a variety of animals around. She’s trained them well too. When she whistles they all come to her. Well, not all of them – the fish and the snake don’t, but let’s pretend that they respond by coming when she whistles too.

I’m intrigued to find out where this going.

My friend’s whistle functions like a “come here” message that’s being sent to each of these well-trained pets. The interesting thing is that they all come to her in their own way. The dog runs at top speed, the monkey swings from furniture piece to furniture piece, the bird flies, the snake slithers the fish swims. They each respond to the “come here” message in their own distinct manner. My friend doesn’t tell them how to come, she just tells them to come, and they move themselves in a manner appropriate to what they are. The pets don’t check some flag that tells them whether or not they can fly or swim or slither – they just act!

That’s a wonderful example. Mind if I borrow it?

Not at all!

Thanks – in fact, I just thought of another similar example which can draw from students’ everyday experiences. Students use all manner of electrical appliances these days, like cell phones, PDAs, electric razors, hair dryers, toasters, electric can openers, electric toothbrushes, etc. Students just turn these devices “on” to use them. Students do not need to tell the appliances what to do, and the appliances do not need to check what they are. When they receive a turn-on message, they just do the right thing!

Back to the calculator and polymorphism. I thought about it for a bit, and pressing the “x” key does not always result in the same action. Consider “1 × 2 × 3”. The first “×” press does something a little bit different than the second “×” press. In the first case there is no prior operation waiting to be performed. In the second case there is a pending operation.

You could perhaps unify those two cases, by introducing a null operation as the pending operation to begin with.

I suppose I could do that! Good idea: that shows a use for the Null Object and Singleton patterns too, so that’s definitely worth considering. In fact, that serves as a good excuse to bring up an interesting design discussion with the class; a different abstraction produces a simpler internal state structure for the calculator. To handle operator precedence we can contrast this example with “1 + 2 × 3”. Now pressing “×” should trigger a different behavior because the pending operation is of lower precedence than the current operation. This can be handled polymorphically.

There are other examples with the calculator too. Different things happen when you press a digit key. Sometimes when you press a digit key you want to append the digit just pressed to the number currently in the calculator’s display. If currentValue is the current value shown in the display and digit is the digit just pressed, then we can accomplish this by writing

```
currentValue = currentValue * 10 + digit;
```

However, just after an operator key has been pressed this is the wrong thing to do.

How so?

Suppose the following keys have been pressed in the given order: $\begin{array}{c} 1 \ 1 \ 2 \ 1 \ + \ 1 \ 1 \ 1 \ + \end{array}$. At this point the current value should be 122. Pressing 1 should not result in the current value becoming 1221. Instead
the current value needs to be saved and a new number must be started.

M: It sounds like you’ve done your homework and really know what you’re doing!

A: I’m getting there. I just didn’t think I would have more homework than my students!

A and M continue to chat. Lights fade to black.

SCENE III

It is early the next week. M is read his office when A rushes in.

A (excitedly): Hi M - got a minute?

M (putting aside his book): Of course. What’s new?

A: Do you remember the calculator example we talked about last week?

M: Indeed I do.

A: I gave it as an exercise to my class. We talked a bit about the Model-View-Controller pattern and I showed them the GUI I built. The exercise I assigned was to build a calculator with just the “1” key and the “+” key enabled. We’ll extend to the full four-function calculator once we have a solid design nailed down.

M: That’s a good way to start! You’re making them develop incrementally – I like it. But is there enough complexity in that problem to get them going on the right track?

A: Some students got started on the right track, but not all. Those that started off on the wrong track will be able to refactor their design after the next class. I have selected two solutions they came up with. I want to discuss them in class – compare their strengths and weaknesses. I’d like to get your take on them before that, if you don’t mind.

A offers some printouts to M. M takes the printouts and studies them for several minutes. (Portions of these printouts are shown in figures 1 to 5.)

M: This first one uses the Strategy pattern to model the contextual determination of behavior that you were talking about. The AccumulateStrategy is used to build up numbers while the SaveStrategy is used to save the current value and start a new one.

A: Now let’s have a look at the second one. This one uses the State pattern to model the contextual determination of behavior. There is an accumulate state (state one) and a calculate state (state two). Saving the value happens in the transition between state one and state two when the operator key is pressed. See here, this solution uses the notion of a null operator to simplify the state structure: _pendingOp is initialized to _noOp, which is a null operator (it doesn’t change the current value of the calculator when it is applied).
public class AccumulateStrategy implements NumberStrategy {

    private CalculatorModel calculator = null;

    public AccumulateStrategy(CalculatorModel calc) {
        calculator = calc;
    }

    public void apply(int value) {
        int newValue = calculator.value() * 10 + value;
        calculator.setDisplay(newValue);
    }
}

public class SaveStrategy implements NumberStrategy {

    private NumberStrategy decorated = null;
    private CalculatorModel calculator = null;

    public SaveStrategy(NumberStrategy decorated, CalculatorModel calc) {
        this.decorated = decorated;
        calculator = calc;
    }

    public void apply(int value) {
        calculator.saveValue();
        decorated.apply(value);
        calculator.resetStrategy();
    }
}

public class Calculator {

    private int _acc; // accumulated computation
    private String _display; // current display
    private IBinOp _pendingOp; // pending operation
    private final IBinOp _noOp = new IBinOp() {
        public int compute(int n1, int n2) {
            return Integer.parseInt(_display);
        }
    };

    private int _curState; // current state
    // INITIALIZATION OF _states IS SHOWN IN FIGURE 5

    public Calculator() {
        _acc = 0;
        _curState = 0;
        _pendingOp = _noOp;
        _display = "0";
    }

    public String enterDigit(char c) {
        _states[_curState].enterDigit(c);
        return _display;
    }

    public String enterOp(IBinOp op) {
        _states[_curState].enterOp(op);
        return _display;
    }
}

Figure 3: Strategy solution - Strategies

Figure 4: State solution – Calculator model class

Figure 5: State solution – States
A : So one solution uses Strategy while the other relies on State. I want to discuss these solutions in class, but I'm not quite sure what I should say about them. They both seem to work. What are some differences between Strategy and State? Their UML class diagrams look the same to me.

M : That's a good question. I think you're right that you could use either pattern. As you've observed, their class diagrams show the same structure. The class relationships in the two patterns may be the same, but notice that the intent of each of the patterns is different.

A : How do you mean?

M : In each pattern the context object knows of a state or strategy object. Part of the distinction is who determines which state or strategy object the context knows. With strategy the determination is made exogenously (external to the system) by the client, while with state its made endogenously (internal to the system) without the client's involvement or perhaps even awareness.

A : When you put it that way it sounds like state is more appropriate for the calculator.

M : Perhaps. But there are other ways of looking at the two patterns, and perhaps the calculator as a whole isn't the context; perhaps the system I am referring to is smaller than the calculator as a whole.

A (studying the Strategy code): Let me see ... the initial strategy is set by the calculator model itself in the strategy solution, and (looking at the State code) the initial state is set by the calculator model in the state solution.

M : The patterns are flexible and can fit a wide variety of situations. It depends on how you model your problem and how you realize your patterns.

A : So it might be that either Strategy or State is reasonable to use in a given situation, and the choice between them is to some extent determined by the perspective one takes on the problem.

M : You do need to be careful to not try to eliminate the differences between the two patterns, though. There are reasons they are considered distinct patterns. But that's perhaps a discussion for another day.

A : Fair enough – I have taken quite a bit of your time already. These discussions we have are always very enlightening.

M : I enjoy them as well. What have you gotten out of these last few conversations?

A : That's easy: teaching is hard work! There is a great deal of preparation that goes into everything that is done in the classroom. Its a real art to do all this preparation and still have a relaxed and informal atmosphere in the classroom that invites student participation.

Writing good code examples is also quite an undertaking. Before I spoke to you I would have written up my half-baked ideas about the calculator and missed a tremendous opportunity to get my students more involved in the class and also to talk about significant OO design issues. Instead I would have shown my students code which hindered my presentation of OO principles and concepts.

M : This is a very important point. You must be consistent in your presentation: your lecture notes, your example code, your projects, even your exams must reinforce what is important.

A : One way to do this effectively for the students is to provide frameworks which allow students to focus on the relevant notions rather than become distracted by irrelevant details.

M : Right.

A : Examples cannot be simplistic. They must be rich enough that they motivate what's being presented (e.g. a design pattern) but must remain accessible to the students, both in terms of being at a level they can comprehend and also in terms of being of interest to them.

A and M : (together, facing audience) Our experience has been that teaching design patterns in CS1 and CS2 makes learning OO in CS1 and CS2 easier for students because they see the benefits of using OO. As with any topic, teaching OO and design patterns well requires much forethought and preparation. For those of us who were ourselves trained in a non-OO paradigm teaching OO and design patterns well requires more forethought and preparation than we might expect, because we must learn to “think OO”.

A : We have found teaching OO with design patterns very rewarding.

M : We hope you do too, but remember,

A and M : (together) your mileage may vary!

Stage lights dim as the curtain closes.

Acknowledgments

The idea for this column came from Joseph Bergin, who was a participant in the second OOPSLA “Killer Examples for Design Patterns and Objects First” workshop. The organizers of the workshop were Stephen Wong, Michael Wick, Phil Ventura, Dung “Zung” Nguyen and myself. The 2003 participants were Joseph Bergin, Michael Caspersen, Richard Rasala and Asher Sterkin.

A and M's conversation is a reflection of part of discussions that took place in pre-workshop conversation on a listserv as well at the workshop. I hope it is understood that in distilling a conversation amongst nine people conducted over several months to a few pages many details are lost. As such, I can take credit only for any inaccuracies or misrepresentations; all the good ideas have come from others. In particular the calculator example is Joseph Bergin’s. He contributed a strategy-based solution. In response Zung
Nguyen contributed a state-based solution. A’s students are clearly very talented to have come up with such good code! I have edited their code to deal only with “1” and “+”, and have compressed comments to fit the code compactly into the column.

Michael Wick and I both decided to use the calculator example in our classes, and some our experiences have crept into the picture. Stephen Wong shared his significant insights into the intent of the patterns. Richard Rasala, Asher Sterkin and Phil Ventura all shared their views based on their use of these and other patterns in their own teaching and software development.

Joseph Bergin’s idea of writing the calculator without if or switch statements as an exercise in polymorphism was very motivating for all of us participating in the workshop. It was also very motivating to my students - many told me afterwards that they finally understood polymorphism after having worked through the exercise.

Zung Nguyen remarks that while setting the states up as inner classes has the advantage of embedding these definitions within the scope of the class definition it also ties them to the calculator model, with the downside that anyone wishing to modify the set of calculator states must modify and recompile the calculator model. The code that served as the basis for the workshop discussion was by design incomplete in several respects (e.g. only some functionality was available), so that the discussion could focus on particular aspects of the design. However, a design in which the context is invariant and the states are decoupled from the context is preferable in many respects. Zung Nguyen contributed a version of his calculator with this design; this example and the code for the examples from the workshop discussion can be downloaded from,

http://www.cse.buffalo.edu/faculty/alphonce/SIGPLAN/

Finally, I stole the idea of structuring this column as a conversation from Gödel, Escher, Bach, though clearly not Hofstadter’s masterful execution of that style.