Skateboarding with the Tooth Fairy
on the Moons of Jupiter
DRAFT (Inheritance, Understanding, and OO Design) DRAFT

Robert Biddle
Victoria University of Wellington
New Zealand
Email: `robert@mcs.vuw.ac.nz`
Web: `http://www.mcs.vuw.ac.nz/~robert`

1 Introduction

This year, 1999, I had some interesting realisations about object-oriented design. They happened in conjunction with teaching, but they're not just about situations that arise in teaching. As I write this in October, I'm still uncertain how they relate to each other, or whether they do at all — but I think they do.

The ideas involved are not completely new. However, it seems a number of the concepts involved in OO design are elusive, so even reviews and reminders should not be dismissed lightly.

I think, I hope, I have this got a better grasp of some elusive concepts than I have had before. This paper is a first attempt to document the ideas, and describe what I think I learned. In the sections below, I first outline each of the main ideas, and only then try and draw them together.

2 Scenarios for Learning Design

My first teaching experiences of 1999 happened in January, my summer holidays, and after attending the International Workshop on Software Reuse in Austin, I visited Rick Mercer at the University of Arizona in Tucson. It was the start of their semester, and a busy time for Rick, but I was made very welcome and had a great time.

In Rick's CS2 course, he was just starting to talk about OO design, and I attended the lectures and tutorial sessions. Rick introduces OO design with the familiar approach of word analysis followed by CRC cards and role-play through scenarios. The students are introduced to the role-play concept by using "scripted scenarios", where students read through a script, each taking a role and reading, like actors in rehearsal. Rick chose some extroverts to do a demonstration in front of the whole class, and everyone quickly caught on.

The next step was for every student to experience the impact of the approach on design, and to do this Rick outlined a design by suggesting key classes, but asked the students to use CRC cards and role-play to complete the design. This exercise was then attempted by groups of about 4 students, 3 or 4 groups to a tutorial session, for about 10 hourly sessions. It was fascinating to watch so many groups in such a short space of time.

The most important thing I saw was how effective the role-play is, even for these first-year students, even in their first design exercise. It took effort and attention, and it took our enthusiasm, but I was impressed by the result and how quickly it was achieved. I really did get the idea that in this tight sequence of a lecture and demonstration, followed by a tutorial group, all the students developed the essential "feel" of the OO design process. I felt certain it would make them better programmers and designers, if only because of this early heightened awareness.

In the following lecture, we talked about design quality, and Rick presented several heuristics. In an experiment, we then presented one design from the previous day's effort, and asked the students to rate the design according to the heuristics. We did all this by rapid show of hands and applause, and it was impressive to see how engaged the students were, and how rapidly they developed opinions about the design alternatives and the heuristics.

What did I learn? Simply how effective role-play and scenarios are, not just for specific designs, but for learning about the *concept* of design in the first place. I also saw how the facilitation, such as scripts and demonstrations, played an important role. When my own students later tried without such facilitation, I realised how important it had been.

Role-play with CRC cards seems so good, and it seems most of the details and rationale have been there from the start.[1] Even so, it's tempting to try and make it better, so in the middle of the tutorial day, Rick and I started exploring extensions. One idea was to use a card for each object, rather than each class. We also tried documenting role-play using UML sequence diagrams so that we could compare results from different designs.

[1] Beck and Cunningham, "A Laboratory for Teaching Object-Oriented Thinking", *Proceedings of ACM OOPSLA*, 1989.

Both these ideas seemed sound and workable, but there are details to check, and interfering with the simplicity of CRC cards should probably be done only with caution. It's tempting to hope there are simple changes that would make the process even better, and I hope I have the chance to pursue these ideas further.

3 Inheritance and Skateboarding

Back home in Wellington, one of the courses I taught in the first semester was our CS1 course. The last part of the course is organised around a design theme. At the beginning of this part, I explicitly review the OO approach, and explain its advantages.

In particular, I suggest the OO model (classes, objects, methods, encapsulation, and so on) matches a common way we think about the real world. For example, I describe how a radio "object" has switches and knobs, like methods, that result in behaviour, and point out that the radio is physically encapsulated, and that the behaviour is not guaranteed if we break the encapsulation. I even use a real radio as a prop, to add some drama; my approach is described in detail elsewhere [2]

As the course continues, I often explain new topics in OO technology by discussing real-world analogies. For example, I explain "composition", and review how we can make classes that themselves use other classes. I then discuss how this is similar to the way we can manufacture products that use other products as components. The radio, for example, is made by using transistors, resistors, and so on.

The last main topic in our CS1 course is inheritance. In keeping with the real-world analogy, I discuss how we organise things in the real-world by building hierarchical taxonomical structures, and then show how inheritance works in a similar way. I've done this before, and normally stop at this point and move on to technicalities. But I pushed a bit harder this year, and asked the students *why* we create organisational hierarchies. I didn't strongly guide their answers, and some interesting viewpoints were presented, including the need for order, simplification, and the desire to make complicated things easier.

I had earlier shown several examples of taxonomies, including the familiar "vehicle" hierarchy. While hearing the discussion about why we create taxonomies, I saw a new way to proceed. I reviewed the example, and talked about cars, buses, trucks, and so forth. I suggested that by using the word "vehicle", we allow ourselves the ability to make expressions and sentences that can be the same no matter whether a car, a bus, or a truck is involved. I pointed out that by writing laws this way, we could have one law that covered all vehicles, with minimal repetition. I went over this a few times, discussing how anything that worked for "vehicle" had to work the same way for each kind, but that each kind of vehicle might have details that didn't apply to just any vehicle.

I suggested that we could gain a similar advantage in programming, because we can organise classes into similar kinds of hierarchies. As with laws, our programs might then be simpler, because we could write code that specified instances of the superclass, and then be applicable to instances of any of the subclasses.

So, I had pushed my lesson harder than usual, and it worked even better than before. Now I could explain not only that inheritance was similar to real-world categorisation, but I could also explain that it yielded similar advantages for similar reasons.

This all lines up very well with a point we typically stress a little later in the process of teaching OO — the big advantage of using inheritance is that it facilitates the *context reuse*, the reuse of context code that uses classes and their subclasses.[3]

It also happened that the legal community helped me further. There had been discussion in newspapers and on TV about possible dangers involving skateboards. It seemed skateboarders could behave dangerously, and the legal system didn't really know how to deal with the problem. You can probably guess what happened next: the government proposed that skateboards be declared vehicles. This was apparently a simple and useful legal step, because it simply involved reuse of an exisiting legal system. This made an excellent opportunity for me to further empahsise the advantages of inheritance.

4 The Framework Tooth-Fairy

Later in the year, I helped my colleague Ewan Tempero teach a graduate course on the object-oriented paradigm. Usually the course is taught as a survey covering a wide range of issues in OO, concentrating on programming language issues. This year we decided to focus on two themes: he was to concentrate on topics relating to inheritance and I was to concentrate on topics relating to design. As this was a graduate course, the students

[2]Biddle and Tempero, "Explaining issues in software reusability via an audio system analogy", *Proceedings of the Second Australasian Computer Science Education Conference, 1997* — also available from my web page in early form as a technical report.

[3]Biddle and Tempero, "Explaining inheritance: A code reusability perspective", *Proceedings of the Twenty-Seventh ACM SIGCSE Technical Symposium*, 1996.— also available from my web page in early form as a technical report.

were all experienced, but there are still many graduate students who did not grow up with OO, so we cover some basic as well as advanced material.

One topic we discussed in detail was OO frameworks where abstract classes are used to create a partly abstract design structure that can then be completed using concrete classes. [4] We read papers, looked at examples, and compared frameworks with other ideas. One issue that arose was that people who had some experience with frameworks had found difficulty in understanding how to use them. That is, given a particular framework and a particular task, people saw difficulties in figuring out how to use the framework to accomplish the task.

Frameworks are real code, designed for reuse, but they invert the normal ordering with which most people are familiar. To reuse a normal component, your code calls it, and passes it some information. To reuse a framework, it calls your code, and passes it some information. This is the inversion of control sometimes called the "hollywood principle": you don't call it — it calls you.

So why is this difficult? It seems that at least part of the problem involves knowing what is and what isn't happening, or not knowing. In order for your code to be called, it has to be written in a certain way. For example, your class may need a method with a certain name. However, you may not realise you are supposed to provide such a method. Or you may simply forget that method, or misspell the name of the method. The result is that the framework cannot call your code. The real problem is that it is often the case that this problem goes unreported: there is no error message or warning, either at compile time or run time. After all, it is often simply a result of creating a subclass and not overriding a method: not usually an error situation.

We discussed frustrations of this kind: where the rules are unclear or unknown, and breaking them results in no immediate notification, but ultimate disappointment. We thought it resembled the situation of a child, who on hearing from friends about the tooth-fairy, leaves a tooth under their pillow, but does not find money the next morning. The problem is that they did not tell their parent. But they don't know that's the problem, and nothing suggests it might be.

Frameworks seem very attractive, both as a way to reuse a design context, and a way to create reusable design contexts. But it seems that our traditional component-oriented apparatus doesn't properly support their context-oriented nature. We have developed the notions of "interface", "parameter", and "encapsulation" all with a viewpoint looking at component reuse, and the notions need further development to properly support the "context reuse" involved in frameworks.

5 OO Design on the Moons of Jupiter

When I started my section of the graduate course, I reviewed the case for OO. In particular, I had talked about the "real-world" connection that I present when teaching about OO to beginners. A number of the graduate students in the course had been tutors in my introductory course, so that material was very familiar to them. In the introductory course, I not only make use of the real-world match to explain the technology; I also make use of it to explain why OO has *advantages* as an approach to software design.

My usual explanation is that the match between the OO approach and the real world is not one of metaphysics, but of practicalities. For example, I outline how OO design can involve the way knowledge is organised in the real-world domain as a foundation for an initial design. This makes starting the design easy for a programmer familiar with the domain, and easy for programmers later attempting to understand the design. I also explain how this approach allows modification in the design to follow the structure of changes when they are made in the domain itself.

Over the weeks of the graduate course, we then looked at a range of topics involving OO design and evaluation. When the time came for my final lecture, I reviewed what we had seen, and tried to summarise. I returned to the early introductory material about the analogy between OO design and the real world, and the advantages that followed. I asked for comments about the case as I had made it, in light of the material we had studied afterward.

In preparing my review session, I had thought it would be interesting to return to my introduction, and was then surprised at what I saw. I was reassured when some students saw the same thing, and without any prompting from me. What we saw was that much of our later study on OO design involved inheritance hierarchies and abstract classes, whereas there is no hint of that in the real-world emphasis of my introduction.

What does this mean? How does the apparent relationship between the real world and the OO paradigm relate to the concern with inheritance and abstraction in more advanced OO design? In retrospect, my experience in the CS1 course might have taught me this lesson. My observation is this: in the real world, we are *very* concerned with taxonomical structure and categorisation. Is the world made up of objects? In my usual introduction to OO, I suggest that this is a common and useful everyday approximation to a complex truth. Some of the

[4] Johnson, *Frameworks = (Components + Patterns)*, Communications of the ACM, October 1997.

complexities of the real world cannot easily be handled in a programming language. However, a lot of our discourse and consideration of the real world does involve abstraction of concrete objects and their behaviour.

Let's say we travel to a moon of Jupiter, and need to describe what we find. (Once I would have suggested the earth's moon, or Mars, but students these days know what those look like!) So, there we are on Io, say, looking around, needing to describe the scene. It doesn't help much to report simply "there are rocks". In order to effectively communicate, we will probably want to start classifying the objects we see, and describing them and their behaviour in terms of taxonomies familiar back on earth. In this way we can harness existing scientific knowledge in much the same way that declaring skateboards to be vehicles harnessed existing laws. Even language itself is a framework.

In both these cases, we fit new objects into existing category structures. The advantage is that by doing this we can reuse some useful context, such as laws or scientific knowledge. The process is similar to how we use frameworks, and the result is also similar. We introduce new concrete specialisations into a framework based on abstractions, and thus reuse the context of the framework itself. In the two examples, the laws and the scientific knowledge are both *frameworks*.

6 Looking for the Big Picture

In the sections above, I have described several realisations I had while teaching about OO design. Each one of them involved me seeing in a new light something I thought I already understood. In brief, the ideas are:

- role-play of scenarios is an effective way to learn OO design
- inheritance can be motivated using the advantage of real-world categorisation
- OO frameworks can be difficult to use because they don't provide direct feedback
- OO design involves abstraction and framework reuse similar to the real world

Seen itemised as above, I find these ideas lose some of their suggestive power. As I have reviewed them earlier, however, I do find that together they suggest ideas for better OO design. They are a reminder that OO design can capture a rich set of higher-level semantic relationships involving taxonomical structure. In code, this can become a framework of reusable context that can simplify software and later software development.

This might not seem very different from the original framework idea. That in itself was a powerful idea, especially in explaining the usefulness of "libraries" involved in environments such as Smalltalk. What I wonder now, however, is whether the framework even now has the right place in the scheme of things.

Does inheritance really support frameworks that well? I recounted above that frameworks do present some difficulties to the user. However, it may be that we need better support for such higher-level semantic structure, and for such frameworks. We should be careful to dissociate the framework concept from the current tool and language technology. As we better understand the role frameworks could fulfill, we might also better understand how we should support them.

Should every use of inheritance be within the structure of a framework? In Rebecca Wirfs-Brock's "Responsibility Driven Design" [5], there are steps that involve going back over an initial design in order to draw out higher-level and abstract classes. Should this instead be part of a process to identify the taxonomical structure that will allow a framework to be identified that can then form the backbone of the design?

In the earlier stages of Responsibility Driven Design, the use of scenario walkthroughs is encouraged as a way to determine and check how functionality is distributed. Perhaps higher-level walkthroughs would be appropriate in the later stages, to determine and check the more abstract functionality involved in an emerging framework backbone. These "abstract scenario" walkthroughs might harness the power of the walkthrough process, even with beginners, to illuminate the mysteries involved in the use of inheritance in design.

To conlude this paper, I offer these speculations:

- the main role of inheritance is supporting frameworks
- we should teach frameworks and inheritance together
- real-world frameworks offer a useful teaching analogy
- abstract scenario walkthroughs may help learning frameworks
- we need to better support both frameworks and framework use

[5] Wirfs-Brock, Wilkerson, and Wiener, *Designing Object-Oriented Software*, Prentice-Hall, 1990.