A Problem-Based Approach to Teaching Design Patterns

Mel Ó Cinnéide and Richard Tynan
Department of Computer Science
University College Dublin
Dublin 9, Ireland
mel.ocinneide@ucd.ie, richard.tynan@ucd.ie

Abstract

The traditional lecture-based approach to course delivery is particularly inappropriate in teaching design patterns effectively. In this paper, we describe our efforts to develop a problem-based approach to the introduction of design patterns in the undergraduate curriculum. Our principal contribution is the development of a set of pattern exercises that enables students to experiment with patterns and to see clearly the advantages accrued by using patterns.

Keywords: Design patterns, problem-based learning

1. Introduction

Design patterns have been one of the most significant developments in software design in the past decade, so it not surprising that many undergraduate curricula now include exposure to them [e.g., **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**], in some cases even in the first year [0, **Error! Reference source not found.**]. Our experience in teaching design patterns to final-year Computer Science students has indicated that the traditional lecture-based approach to course delivery is wholly inadequate in conveying a real feeling for design patterns to students. We have also found that although the introduction of UML-level exercises ameliorated the situation, the students still did not clearly grasp how patterns become code.

This led us to endeavour to develop a set of programming exercises that highlight the benefits of design patterns in a concrete way. Our work has been motivated by ideas from *Problem-Based Learning* as applied to Software Engineering [0]. Problem-Based Learning is a method of teaching that uses real-world situations and exercises that focus a student on a particular area that the tutor wishes to emphasize. In solving the problems, the student will not only understand what the instructor is trying to convey in a theoretical sense, but will also see a practical application of the newfound knowledge.

The pure Problem-Based Learning approach involves using real-world problems that are not constrained for pedagogic purposes and which expose the student to the full complexity of a thorny problem that does not admit just one clean solution. We took a more limited approach where we used guided programming exercises to highlight the flexibility provided by design patterns. For each pattern, we developed two functionally identical applications, one that exploited the pattern, and one that did not. The students were requested to extend both applications with a series of new requirements. Each requirement was of course chosen to illustrate the flexibility of the pattern. In this way, we aimed to let the student see how the pattern is implemented, to appreciate the flexibility provided by the pattern and to work with patterns at an implementation level.

In Section 2, we describe a number of the design pattern exercises we developed, while in Section 3 we present the results of a class questionnaire and finally conclude in Section 4.

2. Design Pattern Exercises

In this section, we describe in some detail two of the design pattern exercises we used to validate the usefulness of our approach to the teaching of design patterns. As well as the Observer and Abstract Factory patterns described here, we also developed exercises for Adapter, Composite, Factory Method, Strategy, Template Method, and Visitor.

Each of the following examples was developed for Java 1.2 using Swing for the user interface. We decided that using realistic user interfaces would provide a more compelling and perhaps more enjoyable context for the students in working with the code.

2.1 Observer

The Observer pattern [**Error! Reference source not found.**] allows an object (an Observer) to register its interest in another object's internal state (the Subject). Whenever the Subject changes state, each registered Observer object is notified of the change and can act accordingly. This pattern enables a number of objects to elegantly maintain consistent state.

In our approach we used a similar example to the motivating example described in [**Error! Reference source not found.**], namely an application that provides a graphical view of a simple data model. Figure 1 presents the interface we used: a bar chart representation of the number of students taking various courses (physics, chemistry, etc.). The values in the underlying model can be changed using an intuitive slider interface for each course. As the slider is moved to the right, the number of students taking that course is increased, and consequently the bar chart is changed appropriately. As described earlier, we provided the students with two functionally-identical programs, one that used an ad-hoc approach to maintaining the consistency between the model and the views, and one that used the Observer pattern to maintain this consistency.

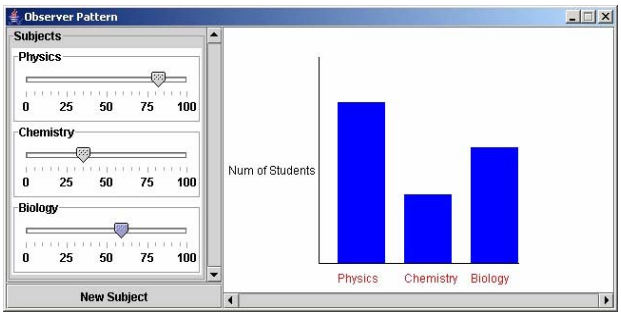

Figure 1: Interface to the Observer pattern example

In the non-pattern example, the coupling between the Subjects and Observers was written in an ad-hoc fashion. It was not intentionally obfuscated, but was written using an approach that a competent programmer who had not encountered patterns might take. The model and controller were amalgamated together, and as the sliders change the bar chart is updated directly. In the pattern example, each course was represented as a Subject and each bar in the bar chart was modelled as an Observer attached to the appropriate Subject.

The first requirement involved extending both implementations by providing another proportional view of the data, in this case a pie chart. In the non-pattern example, the students were forced to understand the precise flow of control within the program to decide where exactly to insert the code that provides the pie-chart view. Extending the design pattern implementation was much simpler. All that was necessary was to attach the pie-chart object to the model and allow the usual Observer interaction to synchronise this view with the data. In both cases nearly identical code excerpts needed to be inserted, however, in the pattern example, it was far clearer where this code should be inserted. Complex control flow in the non-pattern example made it extremely difficult to determine exactly what code should be put where.

A further extension was given to the students to enable an observer to control the state of the model. They were asked to produce a table representation of the model which can be updated through valid user inputs to the table. The tables would display, for each course, the number of students taking it in textual form. Valid numbers could be entered into the table directly to update the model. The sliders and table now become both observers and controllers. The complex interaction between objects could be integrated elegantly in the pattern code but led to complex, bulky code in the non-pattern version.

In order to help the students understand the inner workings of the pattern we then asked students to extend further the pattern version. They were required to change the update model used by the Subjects. This was originally implemented using the push model and we required the students to implement the pull model. Having struggled with the non-pattern version, the simplicity offered by the Observer pattern was highlighted to them in a real coded example, rather than just explained theoretically in a textbook.

2.2 Abstract Factory

The Abstract Factory pattern is used when the necessity exists to create diverse objects of the same general family, e.g., various interface widgets. Clients of the Abstract Factory can request the creation of a member of that family but need never know the concrete class they are dealing with. They can manipulate these abstract members created by the factory and, simply by changing the abstract factory implementation, a different but related family of objects will be created. This does not require any extensions to the client code.

There are various components that are invariably used in the development of a text editor such as menu bars, tool bars, and document views. For this exercise, we presented the students with a simple text editor, one with similar functionality to Microsoft's WordPad. An editor of this sort can have many different but related components such as components with different look and feels for example. It had very limited functionality, and so allowed the students to focus on the pattern and not get bogged down in implementation details.

The non pattern example coupled the instantiation of the various components with some of the behaviour and layout code for the editor. Our pattern version, on the other hand, decoupled this so the instantiation was managed by a subclass of an abstract factory object. The client of the factory was responsible for the layout and sizing of the components, effectively a framework. A screen shot of the simple editor we developed is given below in Figure 2.

We required only one extension from the students to highlight this pattern, namely to produce an editor that can open HTML pages remotely, edit them and save them locally. The extension would require them to update many of the components of the editor, e.g., the menu bar so the user could enter the URL and the document so that it could read HTML remotely. In the non-pattern code this required tinkering with many classes and figuring out the order of

instantiations, which proved time consuming and tedious. The alternative pattern framework took care of all these details and required only that students define a subclass of the Abstract Factory to instantiate each of their HTML editor family of components.

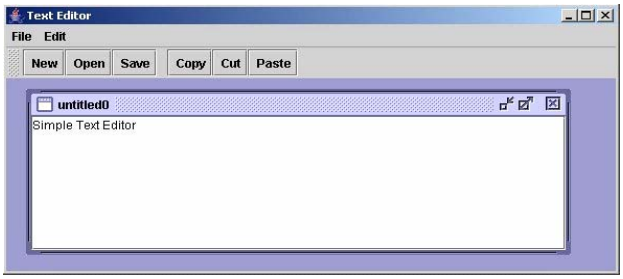

Figure 2: Simple text editor

3. Experiences and Survey Results

The pattern exercises took place in a closed laboratory context as part of a final-year Object-Oriented Design unit. The students worked in pairs and were aided by the authors (the course lecturer and senior tutor). Although there were no marks allocated for attending or participating, attendance at the laboratory sessions was close to 100%.

All the students were able to complete the exercises in the two-hour laboratory session. This was as anticipated, since the exercises were not designed to be a programming challenge as such, but a challenge in pattern comprehension. The atmosphere in the laboratory was extremely positive during these sessions and from the questions asked it was clear that the benefits of patterns were sinking in.

To establish a quantitative basis for assessing this new venture, we asked the students to fill out a questionnaire at the end of the series of exercises. The feedback we had received during the exercises themselves was very positive; the results of the survey were less so[1]. 74% of students said they enjoyed the exercises while almost 80% found them helpful in understanding patterns.

One interesting aspect was that only 20% of students found working with a partner to be useful, a fact that does not augur well for the future of pair programming. In designing these exercises, we had been cautious about the use of the Swing toolkit, as many students had not used it previously. However, in the survey 88% said it made the examples more interesting while only 6% said it made the examples harder to understand.

4. Conclusion

Design patterns are essential material to cover in the CS curriculum, and offer the possibility a non-traditional approach being taken to their presentation. We developed a set of programming exercises that explore the flexibility and inner workings of patterns at a programming level, by placing them in stark contrast to functionally identical software that has been developed without patterns. We enjoyed the experience of helping the students to work through the material and our class survey showed that the students themselves gained a lot from the experience.

We are currently developing this material further by tidying up the existing examples, developing examples for more patterns, and porting the code and exercises to the Eclipse environment. The code for these exercises is available by emailing the authors.

References

[1] Alphonce, C, Pedagogy and Practice of Design Patterns and Objects First: A one-act play, *ACM SIGPLAN Notices*, May 2004.

[2] Armarego, J., Advanced Software Design: a case in problem-based learning, *Proceedings of the 15th Conference on Software Engineering Education and Training*, Covington, Kentucky, February, 2002.

[3] Armarego, J., Student perceptions of quality learning: evaluating PBL in Software Engineering, *Proceedings of the Teaching and Learning Forum*, Murdoch University, 2004.

[1] It was interesting to note in this context that in [0] a survey of students who had participated in a problem-based learning approach to Requirements Engineering produced quite ambivalent results.