8

How ‘Object-Oriented’ is Our Design?

8.1 Introduction

In the course of the last two chapters, we have seen that the design process involves making several choices. This is quite typical of any engineering design, and should come as no surprise. For instance, when a bridge is designed, an engineer is starting with an architect’s plan and is making choices about the kind of materials needed. While doing this, the engineer is typically guided by well-formulated design rules.

Given the multitude of choices that we face during the object-oriented design process, it is only natural to ask if there is a set of rules that can help us make the correct decisions. More specifically, we would like some way of answering two questions a designer often grapples with:

1. Have I made the right decision in assigning responsibilities?

and

2. In case I make a mistake, how can I detect it early and correct it?

In this chapter we demonstrate via examples how an awareness of a concept known as refactoring can help answer the above questions. Refactoring is defined simply as the process of improving the internal structure (design and code) of a piece of software without altering the module’s external behavior. The process may be applied to a system in production, or we can use this process just as effectively during development. Practitioners have developed a set of rules that can be used systematically to refactor code. Some of these rules serve as means for detecting where modifications are needed, and it is not surprising that they can often be turned into guidelines for good software practice. The rules are relatively simple and the changes we make are usually small, so it is usually the case that not much goes wrong while refactoring. Familiarity with these rules can help a beginner make decisions about how to assign responsibilities, when to introduce inheritance, etc.
It should be noted that there is a vast amount of knowledge on the subject of refactoring, and our treatment of it in this book merely scratches the surface. Nonetheless, it is useful to see this as an integral part of the object-oriented design process.

8.2 A First Example of Refactoring

Our first example illustrates how the two refactoring rules, \texttt{EXTRACT METHOD} and \texttt{MOVE METHOD}, are applied during system development. To serve as the platform for using these rules, we impose some new requirements to the library system we designed and implemented in Chapter 7. After constructing an initial design and implementing the code, we refine the solution using refactoring rules.

In Section 8.2.1 we describe the new requirements and come up with an implementation. Refactoring is done in Section 8.2.2.

8.2.1 A library that charges fines: Initial solution

Consider the situation where the library decides to cut down on truancy by imposing fines. When an overdue book is returned, the librarian would like to know the amount of fine and send out a notice to the user regarding the fine payable. The system should therefore compute the fines and display the relevant information. The resulting changes in the business process are captured in the use case in Table 8.1.

This use case for \texttt{Book Return with Fines} is similar to what we had earlier, with one addition — the amount of fine owed is computed whenever a book is returned. Obviously, the \texttt{Member} class needs to be changed to track the amount of fine owed. Also, notice that the use case does not say anything about actually collecting fines from a member and updating the corresponding \texttt{Member} object after the fine is paid. These are left as exercises.

We have the following formula for computing the fine:

\textbf{Rule 5} New books (less than a year old) are charged $0.25 for the first day and $0.10 for every subsequent day. Older books are charged $0.15 cents for the first day and $0.05 for every subsequent day. If a book has a hold on it, the amount of fine is doubled.

Before we construct the modified sequence diagram, we have to decide where the amount of fine owed will be computed. There are three possible options: \texttt{Library}, \texttt{Book}, and \texttt{Member}. We can make a case for each option. \texttt{Book} would be appropriate since it is the return of the book that incurs a fine; \texttt{Member} is where the fine is stored and is therefore the place it could be computed; since both \texttt{Book} and \texttt{Member} are involved in this, \texttt{Library} is perhaps the best place to do the computation. We decide (somewhat arbitrarily) that \texttt{Library} is the place where the fine is computed. The new sequence diagram for returning books is shown in Figure 8.1.