Lecture 13: Model-View-Controller and Encapsulating Data Structures

10:00 AM, Feb 23, 2020

Contents

1 Model-View-Controller

2 Encapsulating Representation

2.1 Replace Fixed Data Structures with Interfaces . . . . . . . . . . . . . . . . . . . . . 3
2.2 Create Concrete Classes that Implement the New Interfaces . . . . . . . . . . . . . 4
2.3 Initialize Data with Objects of the New Concrete Class . . . . . . . . . . . . . . . 4
2.4 Anticipated Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

3 Summary

3.1 Two Myths About Encapsulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
3.2 The Many Faces of Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

Motivating Questions

How do we structure applications (classes and their roles) to enable different data structures for different constraints or scales of data?

Objectives

By the end of this lecture, you will know:

• about the model-view-controller architecture for software applications

• How to set up an application to enable different choices of data structures for key data

By the end of this lecture, you will be able to:

• Implement an application in a way that allows different choices of data structures for the core data

Today, we continue with the Banking example from last class, looking for additional ways to design it well using OO principles.
1 Model-View-Controller

The Banking application code from last lecture contained three classes: Account, Customer, and BankingService. The banking application lacked any notion of an interface for communicating with a user. Now that you’ve seen input/output in lab, we’re going to add a class to the bank that handles user-communication. We’ll call this class the BankingConsole:

```java
import java.util.Scanner;

public class BankingConsole {
    private Scanner keyboard = new Scanner(System.in);
    private BankingService forService;

    public BankingConsole(BankingService forService) {
        this.forService = forService;
    }

    public void loginScreen() {
        System.out.println("Welcome to the Bank. Please log in.");
        System.out.print("Enter your username: ");
        String username = keyboard.next();
        System.out.print("Enter your password: ");
        String password = keyboard.next();
        forService.login(username, password);
        System.out.println("Thanks for logging in!");
    }
}
```

With the addition of the BankingConsole, we can talk about the overall architecture (configuration and roles) of the application and its classes. We talked about how the classes can be divided into three roles:

- **The view** (BankingConsole), which contains the code that interacts with the user (whether text I/O, audio, web interface, etc). The user gives input to the view, which executes commands in the application through the ...

- **controller** (BankingService), which contains methods for the major operations that the application provides (like logging in, withdrawing funds, etc). Once the controller knows what the user wants to do, it coordinates actual completion of a task by calling methods in the ...

- **model** (Account and Customer), a collection of classes that contain the data and perform operations on the data to fulfill application tasks.

After the model finishes actually manipulating application data, the controller can send information back to the view to pass along to the user.

This architecture, known as model-view-controller is quite common in software engineering. It reinforces the idea that the interface code should be separate from the underlying operations, and that the underlying operations should be expressible against a variety of data structures. The details of the data structures live in their own classes, with fields protected through access modifiers. This
enables updating an application with different data details without having to reimplement the core logic.

2 Encapsulating Representation

Now we consider the third problem we cited in our critique of the original code: the BankingService class fixes the assumption that accounts and customers should be stored as linked lists. If we have a lot of accounts, for example, we might want to be able to store them in an array, or perhaps a BST. Right now, the methods in BankingService have been written specifically for LinkedLists (due to the use of the for-loops). Code designed for long-term evolution and maintenance (in other words, most code in a production environment) should NOT do this.

To fix this, we need to rewrite the code to remove both the iterator-for-loops and the specific references to LinkedList. But how? This involves several steps, described over the next several subsections.

2.1 Replace Fixed Data Structures with Interfaces

In general, here is how to factor a fixed data structure out of existing code:

1. Find all variables whose type you want to generalize.
2. Introduce interfaces for the types of these variables (some variables may be able to share the same types).
3. For each place in the current code that relies on the current type of the variable, ask yourself what that code is trying to compute (i.e., figure out a purpose statement for it). Invent a method name for that computation, add it to the interface, and replace the existing code with a call to the new method.

To make this clearer, let’s apply these steps to our BankingService program.

1. Which variables to we want to generalize?: Each of accounts and customers.

2. Choose interface names for the variables: Each of these is representing a set, so IAccountSet and ICustSet are reasonable choices.

```java
interface IAccountSet {}
interface ICustSet {}

class BankingService {
  private IAccountSet accounts;
  private ICustSet customers;
  ...
}
```
3. For each place in the current code that relies on the current type of the variable, ask yourself what that code is trying to compute: Let’s take the original getBalance code as an example.

```java
double getBalance(int forAcctNum) {
    for (Account acct:accounts) {
        if (acct.numMatches(forAcctNum))
            return acct.getBalance();
    }
    return 0;
}
```

The for-loop here locates the account with the given number, then gets the balance from that account. The general purpose of the for-loop, then, is to find an account by its number. This suggests the following method on IAccountSet:

```java
interface IAccountSet {
    // returns the account whose number matches the given number
    Account findByNumber(int givenNum);
}
```

4. Replace current code on a specific data type with calls to methods in the new, general, interface: Now, we rewrite getBalance to use findByNumber.

```java
double getBalance(int forAcctNum) {
    Account acct = findByNumber(forAcctNum);
    return acct.getBalance();
}
```

Note that we have not yet addressed what happens if there is no account with the given number in the list. We will return to that in the next lecture.

Follow similar steps to generalize the withdraw and login methods. We leave these as an exercise so you can practice.

2.2 Create Concrete Classes that Implement the New Interfaces

Now that we have rewritten BankingService to use IAccountSet and ICustSet, we need classes that implement these interfaces. Our original code provides an initial implementation using LinkedList.

```java
class AcctSetList implements IAccountSet {
    LinkedList<Account> accounts;

    public Account findByNumber(int givenNum) {
        for (Account acct:accounts) {
            if (acct.numMatches(givenNum))
                return acct;
        }
        return null;  //not good -- will fix when we handle errors
    }
}
```
With the generalized `findByNumber` method, it isn’t clear what to use as the return type if no account has the given number: different methods that call this search method might need different default answers. For now, we will use the *very wrong approach* of returning `null`, just so we can get the code to compile. We will discuss how to do this properly in a couple of weeks.

### 2.3 Initialize Data with Objects of the New Concrete Class

We have generalized `BankingService` and made new classes for the data structures we need. One step remains: we have to tell the `BankingService` to use our concrete classes. Where should this happen?

It should *not* happen within `BankingService` itself. The whole point of encapsulation is that `BankingService` shouldn’t know which specific data structures it is using. The only other way to get a specific object into a `BankingService` object is through the constructor. This is the answer: when you create a `BankingService` object, pass it objects of the specific data structure that you want to use.

```java
class Examples {
    BankingService B = new BankingService(new AcctSetList(),
                                        new CustSetList());
    ...
}
```

This illustrates how we create different banking services with different data structures for accounts and customers. If we had an Array-based implementation of `IAccountSet` as a class named `AcctSetArray`, we could create a different banking service using:

```java
BankingService B = new BankingService(new AcctSetArray(),
                                        new CustSetList());
```

Since `BankingService` only uses methods in the `IAccountSet` and `ICustSet` interfaces, we can freely chose a data structure without editing the code within the `BankingService` class (which was our goal).

### 2.4 Anticipated Questions

- *This banking service has no customers or accounts. How do we populate those?*
  
  In a full banking service program, the `IAccountSet` and `ICustSet` interfaces would also need methods for adding new elements. These notes do not include these in order to stay focused on the topic at hand.

- *If I have to specify the data structure to use in the constructor, how would I switch to a new data structure after my banking service had been running for a while?*
  
  These notes have not addressed this question. To do this, you would first need a method to convert your existing data from one representation to the other. Then, you would either create a new `BankingService` with the new data, or use some method provided in the `BankingService` to update the data structure.

  Something to think about: Providing a simple setter method would let you change the data structure. What’s wrong with this solution? What would a better solution look like?
3 Summary

Compare the original banking code to the version posted with today’s notes. The new BankingService is much cleaner and more maintainable. It allows the information about accounts and customers to change with less impact on the banking service methods. The banking service no longer relies on any particular data structure for accounts and customers. We achieved both of these goals by isolating data and methods in classes, and using interfaces to separate general data from implementation details.

Key take-aways from these lectures:

- Encapsulation is about putting data and the methods that operate on that data together. OO classes provide a natural mechanism for doing this.
- Encapsulation matters because it lets you change data and how it is used without editing existing code. This lecture has shown two examples of this:
  - We might add information (such as a withdrawal fee) and want to change methods (such as withdraw) to use that information.
  - Writing code that can be customized to different specific data structures (such as linked lists versus arrays).
- Java provides access modifiers that let you control which other classes can access your methods and fields. You should put explicit access modifiers on all of your fields and methods.
- Encapsulation is NOT just about protecting your data. It is about your class hierarchy (more generally, the architecture of your program). The Java access modifiers are used in conjunction with encapsulation (and indeed help reinforce encapsulation), but encapsulation is a much broader topic.

Encapsulation is an important issue no matter what language you are programming in. Different languages provide different support for encapsulation. Java’s support comes in the form of classes and interfaces (supported by access modifiers). Other languages have other mechanisms. When designing a new product in any language, it is important to ask what information you want to protect and what decisions you want to be able to change later, then understand how the language can help you achieve those goals.

3.1 Two Myths About Encapsulation

Those of you with prior Java experience may have heard two general guidelines or slogans that aren’t quite accurate:

- **MYTH: Always make a getter and setter for every private field.** No. Sometimes, we have fields that we don’t want anyone to have access to – they are for internal use only. Creating getters/setters for such fields contradicts the design goal of those fields. Would you publish a getter for the password field of Customer, for example? No – that data should only be handled within the Customer
This rule can also violate representation encapsulation. Imagine that ICustSet provided a method `getCustomers` that returned the `customers` field. That would defeat the whole purpose of hiding the representation, because a programmer could get the actual data structure and write code against it (like a for-loop against the `LinkedList`). This is a great example of where the getter is exactly the wrong thing to provide.

- **MYTH: Encapsulation equals Information Hiding.** These concepts are related, but not equal. One can encapsulate data and methods, but still not hide information (if you expose everything through getters, for example). Encapsulation is part of a solution for hiding information, but information hiding needs careful design beyond just putting all data and methods together in a class.

### 3.2 The Many Faces of Variables

At this point, it is worth stepping back to review the many uses we have seen for variables in the course so far:

- To capture fields whose values might change over time (like the balance field in Accounts)
- To track the progress of a local computation within a method, (like the `pos` variable used to track the steps of a for-loop); this includes storing the result of part of a computation so you can return it at the end of a method
- To capture the history of one computation (like logging in) for use by another (like withdraw)

Why do we point out these different uses? Because conceptually they are very different, but notationally they all look the same in Java. When you are working on a problem, it’s worth thinking about which of these uses might arise in your program so you can create and manage variables appropriately.

---

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)