Lecture 14: Model-View-Controllert

11:00 AM, Feb 26, 2020

Contents

Motivating Questions

How should we break up the code for an application into separate classes? How do we handle login errors?

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 Expanding the Banking Application

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 method to the bank that handles user-commnication:

```
  public class BankingService {

    // this method is new: it provides keyboard input for logging in
    public void loginScreen() {
        Scanner keyboard = new Scanner(System.in);
        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();
        this.login(username, password);
        System.out.println("Thanks for logging in!");
      }
}
```

We would call this method from within the `Main` class:

```
public static void main(String[] args) {
        BankingService B = new BankingService();
        Customer kCust = new Customer("kathi", "cs18");
        Account kAcct = new Account(100465, kCust, 150);
        B.addAccount(kAcct);
        B.loginScreen();           // <---- new method called here
        kAcct.printBalance();
        B.withdraw(100465, 30);
        kAcct.printBalance();
}
```

This leaves us with a `BankingService` class with the following high-level outline:

```
public class BankingService {
    private LinkedList<Account> accounts = new LinkedList<Account>();
    private LinkedList<Customer> customers = new LinkedList<Customer>();

    public void addAccount(Account newA) { ... }

    private Account findAccount(int forAcctNum) {
        for (Account acct:accounts) { ... } }

    public double getBalance(int forAcctNum) {
        return findAccount(forAcctNum).getBalance();
    }

    public double withdraw(int forAcctNum, double amt) {
        return findAccount(forAcctNum).withdraw(amt);
    }

    public void loginScreen() {
        Scanner keyboard = new Scanner(System.in);
        ...
      }

    private Customer findCustomer(String custname) { ... }
    public String login(String custname, String withPwd) {
        Customer cust = findCustomer(custname);
        if (cust.checkPwd(withPwd)) { ... }
        ...
    }
}
```

This class contains the core data structures and methods for banking operations, along with classes for letting a user interact with those methods. We learned last class that we should make sure that methods go into the class that has the data that they operate on: this code satisfies that criterion. But when breaking a system into classes (or other smaller collections of code), we should ask another question:

Might I want to swap out different implementations of any of these parts later?

Classes and objects are a nice structuring method because you could use different classes to provide the same functionality (as we saw with `LinkedList` versus `ArrayList`), yet with different implementation decisions and performance. So let's critique our current `BankingService` by asking whether any parts of it might be worth swapping out with other detailed code later.

Stop and Think: What might you swap out?

Three specific aspects of this code stand out as being worthy of swapping out:

1. Which data structure we use for accounts
2. Which data structure we use for customers
3. What kind of interface we provide (website, keyboard I/O, voice-driven, etc)

Our first task today is to restructure the code to handle the first and third (we'll leave the second to you as a practice exercise).

1.1 Pulling out the Accounts data structure

We'll introduce a new class to hold the specific details of the data structure for accounts. We'll move the actual data structure and any code that depends on that specific data structure to this new class. In this case, we have to bring over the `findAccount` method (as the for-loop that it uses depends on the data structure being a list) and the `addAccount` method (which depends on having an `addFirst` method):

```
public class AccountList {
    private LinkedList<Account> accounts = new LinkedList<Account>();

    AccountList(){}

    public void addAccount(Account newA) {
        this.accounts.addFirst(newA);
```

```java
    }

    public Account findAccount(int forAcctNum) {
        for (Account acct:accounts) {
            if (acct.numMatches(forAcctNum))
                return acct;
        }
        return null;
    }
  }
```

We then update the type of the `accounts` field in the `BankingService` class, and redirect uses of the moved methods to go through the `accounts` field.

```java
  private AccountList accounts = new AccountList();
```

Why again are we doing this? Because having the data structure in a separate class would let us change the specific data structure without having to modify code in the `BankingServices` class. For example, we could decide to use an `ArrayList` or a binary search tree as the data structure for storing accounts. As long as the `AccountList` class provided the `addAccount` and `findAccount` methods relative to the chosen data structure, the `BankingService` class would continue to work.

1.2 Pulling out the User Interface

To pull out the user interface, we make a new class for the interface code. Here, we're calling it `BankingConsole`. We'll move the `loginScreen` method over to this new class:

```java
import java.util.Scanner;

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

  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();
    this.login(username, password);
    System.out.println("Thanks for logging in!");
  }
}
```

IntelliJ flags the call to `this.login` as having an error. Now that we've moved the `loginScreen` out of `BankingService`, the `login` method is no longer in the `this` class. Removing `this` doesn't help: the `login` method is still in the `BankingServices` class. So what do we do? Two suggestions jump to mind:

1. move the `login` method from `BankingService` into `BankingConsole` as well

2. have the `loginScreen` method take a `BankingService` object as input (which we could then use to access the `login` method)

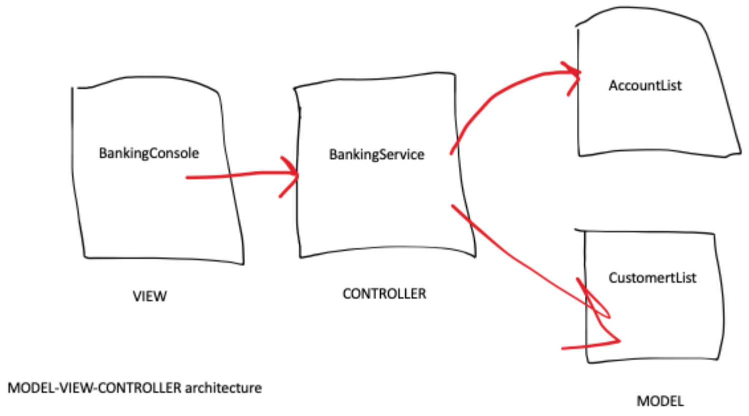
The first option ends up not making sense: the `login` method isn't really about the interface, so it doesn't seem to belong in the class for the interface code. More practically, if we move `login`, we'd then run into a similar problem with `findCustomer`, and that definitely isn't related to the user interface. So perhaps we should try the second option.

The second option would work. But we actually want to solve this problem slightly differently. It will turn out that many `BankingConsole` methods would need to access methods in `BankingService` (imagine that the user interface gave someone a way to make a withdrawal, for example). Rather than have all of them take a `BankingService` object as input, we can pass a single `BankingService` object to the `BankingConsole` constructor, then use that for all service operations. The code would look as follows:

```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!");
  }
}
```

2 Model-View-Controller

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, as shown in the following diagram:



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

The red arrows show the flow of method calls: the view classes call methods in the controller classes, which in turn call methods in the model classes. After the model finishes actually manipulating application data, the controller can return 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.

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 customters 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.

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`.