软件体系结构

《软件体系结构作业十七》

学　号 22920212204396

姓　名 黄子安

2024 年 5 月 16 日

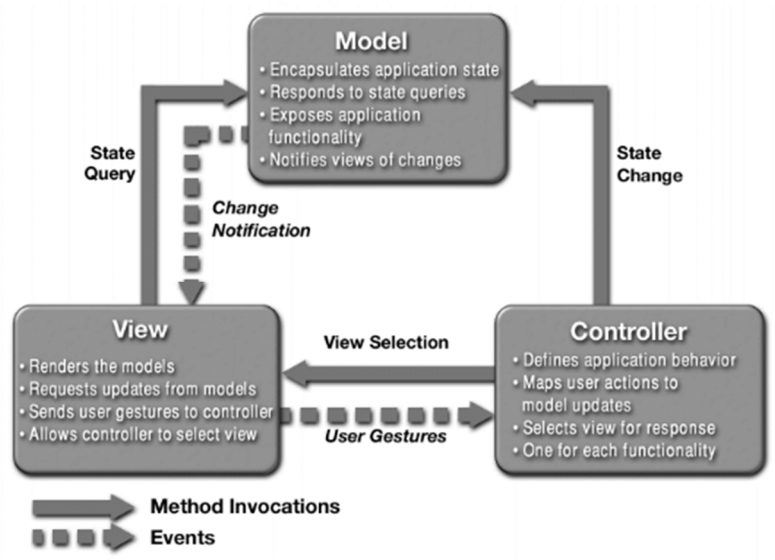
1、阅读：Java SE Application Design With MVC

https://www.oracle.com/technical-resources/articles/javase/application-design-with-mvc.html

（1）3 个组成

MVC 架构由模型、控制器、视图三个部分组成

- **模型（Model）：**模型表示数据以及管理访问和更新这些数据的规则。在企业软件中，模型通常作为对现实世界过程的软件近似。
- **视图（View）：**视图渲染模型的内容。它具体规定了模型数据应如何呈现。如果模型数据发生变化，视图必须根据需要更新其展示方式。这可以通过使用推模型（push）来实现，视图将自己注册到模型以接收更改通知，或者通过拉模型（pull）来实现，在拉模型中，视图负责在需要获取最新数据时调用模型。
- **控制器（Controller）：**控制器将用户与视图的交互转化为模型将执行的操作。在独立的 GUI 客户端中，用户交互可以是按钮点击或菜单选择，而在企业 Web 应用程序中，则表现为 HTTP 请求，根据具体情况，控制器还可能选择一个新的视图（如新的结果的网页）呈现给用户。



（2）MVC 组件之间的交互

对应文章详细介绍了 Java SE 6 中实现 MVC 的一种方法（…有点太古老了吧），一旦模型、视图和控制器对象实例化后，会发生以下情况：

视图注册为模型的监听器：模型底层数据的任何更改都会立即导致广播更改通知，视图会接收这些通知，这是一个推模型（push）。模型并不知道视图或控制器的存在，它只是向所有感兴趣的监听器广播更改通知（类似观察者模式）。

控制器绑定到视图：这通常意味着在视图上执行的任何用户操作都会调用控制器类中注册的监听方法。

控制器获得对底层模型的引用：控制器的监听方法响应的时候也会对模型产生作用

用户与视图的交互流程如下：

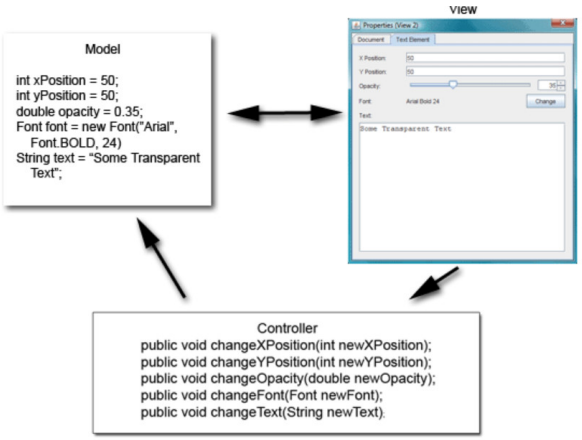
1、视图通过一些监听方法识别到 GUI 操作（按下按钮或拖动滚动条等）

2、视图调用控制器中的适当方法。

3、控制器访问模型，可能操作以适当的方式更新模型。

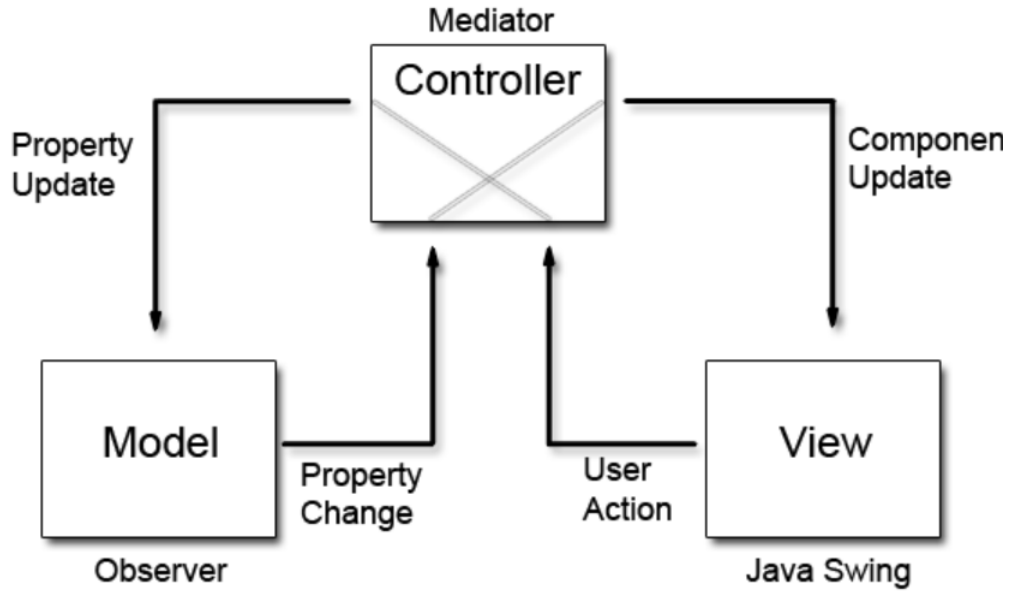
4、如果模型已被更改，它会通知感兴趣的监听器，如反作用回视图。在某些架构中，控制器也可能负责更新视图。

这里模型当中不显示含有视图，而是类似观察者模式通过事件驱动的方式，这样好处就是可以作用到多个视图上



（3）一个修改

现在的 MVC 架构是将控制器放在模型和视图中间的位置，使用这种修改后的 MVC 有助于更彻底地解耦模型和视图。在这种情况下，控制器可以决定将视图的变化作用到制定的模型上。此外，控制器也可以为一个或多个注册到它的视图提供模型变化产生的影响（更进一步发挥，前后端完全分离（？））。



2、 LoD 原则强调“只和朋友通信，不和陌生人说话”。请举例说明 “朋友圈” 认定依据是啥？

LoD（Law of Demeter，Demeter 定律）原则强调“只和朋友通信，不和陌生人说话”，即一个对象应当只与其**直接相关**的对象（朋友）进行交互，而不应该直接访问其他对象（陌生人）的内部结构。

在 LoD 原则中，“朋友圈”的认定依据包括以下几个方面：

1、当前对象本身（this）；

2、被当作**当前对象的方法的参数**传入进来的对象；

3、当前对象的方法所**创建或者实例化的任何对象**；

4、当前对象的**任何组件**（被当前对象的实例变量引用的任何对象）。

前三个意思相对比较清晰，下面先举个例子说明下第 4 个的含义，比如在该例子中 engine 和 wheels 就是当前对象的组件，它们是通过构造函数引入的成员变量，在当前对象创建时被显式地赋值的，因此当前对象理应可以直接去访问这些对象

```
class Car {
    private Engine engine;
    private List<Wheel> wheels;

    public Car(Engine engine, List<Wheel> wheels) {
        this.engine = engine;
        this.wheels = wheels;
    }

    public void start() {
        engine.start();  // 当前对象的组件，可以直接访问和操作
    }

    public void checkWheels() {
        for (Wheel wheel : wheels) {
            wheel.checkPressure();  // 当前对象的组件，可以直接访问和操作
        }
    }
}
```

最后给出符合迪米特原则和不符合迪米特原则的例子，该例子中有三个类，分别为 Library、Book、Author，根据迪米特原则 Library 和 Book、Book 和 Author 是直接朋友关系，而 Library 和 Author 就不属于直接朋友关系，按照下述写法并不符合迪米特原则，在 Library 对象中要打印所有的作者名字的时候通过遍历 Books 直接获取了对应的 Author 对象，从而导致了 Library 直接访问了非直接朋友关系的 Author 对象，并不符合迪米特原则

```java
public class Library {
    private List<Book> books;

    public Library() {
        books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }

    public void printAuthors() {
        for (Book book : books) {
            System.out.println(book.getAuthor().getName());
        }
    }
}
```

```java
public class Book {
    private Author author;
    private String title;

    public Book(String title, Author author) {
        this.title = title;
        this.author = author;
    }
    public Author getAuthor() {
        return author;
    }
    public String getAuthorName() {
        return author.getName();
    }

    public String getTitle() {
        return title;
    }
}
```

```java
public class Author {
    private String name;

    public Author(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

下面修改对应的方法，在此处 Library 不再直接访问 Author 对象，而是让对应的 Book 去访问 Author 对象，这个时候都满足访问对象之间是直接朋友关系，因此符合迪米特原则

```java
public class Library {
    private List<Book> books;

    public Library() {
        books = new ArrayList<>();
    }

    public void addBook(Book book) {
        books.add(book);
    }

    public void printAuthors() {
        for (Book book : books) {
            System.out.println(book.getAuthorName());
        }
    }
}
```

最后运行一下程序

```java
public class Main {
    public static void main(String[] args) {
        Author author1 = new Author("J.K. Rowling");
        Author author2 = new Author("J.R.R. Tolkien");

        Book book1 = new Book("Harry Potter and the Sorcerer's Stone", author1);
        Book book2 = new Book("The Hobbit", author2);

        Library library = new Library();
        library.addBook(book1);
        library.addBook(book2);

        library.printAuthors();
    }
}
```

二者的输出虽然相同，但是从类的关联关系来看，采用迪米特法则后当一个类发生修改时，会尽量少的影响其他的类，扩展会相对容易，能减小系统内部的耦合性

```
D:\Java17\bin\java.exe "-javaagent:D:\IdeaU\IntelliJ IDEA 2023.1.3\lib\idea_rt.jar=50341:D:
J.K. Rowling
J.R.R. Tolkien

Process finished with exit code 0
```