COMP 401
MODEL-VIEW-CONTROLLER (MVC)

Instructor: Prasun Dewan
PREREQUISITES

- Interfaces
- Main Console Input
- Inheritance
GENERAL PROBLEM

How to break up our program into multiple classes?
Separation of Concerns

- History Semantics
- History Display 1
- History Semantics
- History Display 2

Can change display without changing other aspects of history

Display and semantics should go in different classes
if a part A of a class can be changed without changing some other part B of the class, then refactor and put A and B in different classes
Patterns

- Recurring theme
- Bean, Vector pattern
  - Conventions for readability
- Loop patterns
  - Event-controlled
  - Counter controlled
- Design patterns
  - Helps identify the kind of classes our program should have
Design Pattern

- Reusable program decomposition pattern
- Not specific class or interface, infinite family of classes/interfaces implement this pattern.
- Usually involves multiple objects
- Language-independent
- Include architecture and frameworks
- Inspired by Architectural Pattern (Christopher Plummer)
SIMILAR

2nd story porch supported by 1st floor porch columns

DIFFERENT

No outside stairs to 2nd story
Flat (not bay) window
Wide plank siding
Screened-in porch
MVC Motivation

- History Semantics
- History Display 1
- History Semantics
- History Display 2

Can change display without changing other aspects of history

Display and semantics should go in different classes
MVC Motivation

- Object Semantics/Service
  - Object User Interface

- Object Semantics/service
  - Object User Interface
Questions

- How to reuse code among different interactive applications offering the same service?
- How to simultaneously create multiple user interfaces for same service?
  - Normal vs. Slide sorter
  - Shortcuts vs. menus vs. buttons
QUESTIONS

- How to simultaneously create multiple user interfaces for same service on different computers?
  - Facebook, email
**Questions**

- How to simultaneously create distributed user interfaces
  - multiple complete user interfaces for different users on different computers
  - Single user-interface on large computer controlled by multiple mobile devices
EXAMPLE: COUNTER

Can add arbitrary positive/negative value to an integer

Different user interfaces
CONSOLE INPUT AND OUTPUT

C:\Users\Sasa2\src\comp110>java examples.mvc.ConsoleUI
Counter: 0
1
Counter: 1
-1
Counter: 0
5
Counter: 5
CONSOLE INPUT AND JOPTION OUTPUT

![Console Input and JOption Output Example]

The figure above demonstrates how to use the `JOptionPane` class to display a message on the console. The first example shows a message indicating that the counter is 1, and the second example shows a message indicating that the counter is 0.
CONSOLE INPUT, OUTPUT AND JOPTION OUTPUT
public class MonolithicConsoleUI {
    public static void main(String[] args) {
        int counter = 0;
        while (true) {
            System.out.println("Counter: "+ counter);
            int nextInput = Console.readInt();
            if (nextInput == 0) break;
            counter += nextInput;
        }
    }
}
public class MonolithicConsoleUI {
public static void main(String[] args) {
    int counter = 0;
    while (true) {
        System.out.println("Counter: " + counter);
        int nextInput = Console.readInt();
        if (nextInput == 0) break;
        counter += nextInput;
    }
}
import javax.swing.JOptionPane;

public class ConsoleUI {
    public static void main(String[] args) {
        int counter = 0;
        while (true) {
            JOptionPane.showMessageDialog(null, "Counter: " + counter);
            int nextInput = Console.readInt();
            if (nextInput == 0) break;
            counter += nextInput;
        }
    }
}
import javax.swing.JOptionPane;
public class ConsoleUI {
    public static void main(String[] args) {
        int counter = 0;
        while (true) {
            JOptionPane.showMessageDialog(null, "Counter: " + counter);
            int nextInput = Console.readInt();
            if (nextInput == 0) break;
            counter += nextInput;
        }
    }
}
public class ACounter implements Counter {
    int counter = 0;
    public void add (int amount) {
        counter += amount;
    }
    public int getValue() {
        return counter;
    }
}
public static void main (String args[]) {
    (new AConsoleUI()).edit (new ACounter());
}
MODEL?
public class ACounter implements Counter {
    int counter = 0;
    public void add (int amount) {
        counter += amount;
    }
    public int getValue() {
        return counter;
    }
}

No input/output
public class ConsoleUI {
    public static void main(String[] args) {
        int counter = 0;
        while (true) {
            System.out.println("Counter: " + counter);
            int nextInput = Console.readInt();
            if (nextInput == 0) break;
            counter += nextInput;
        }
    }
}
public class AConsoleUIInteractor implements CounterInteractor {
    public void edit (Counter counter) {
        while (true) {
            System.out.println("Counter: "+counter.getValue());
            int nextInput = Console.readInt();
            if (nextInput == 0) return;
            counter.add(nextInput);
        }
    }
}
public class AMixedUIInteractor implements CounterInteractor {
    public void edit(Counter counter) {
        while (true) {
            JOptionPane.showMessageDialog(null, "Counter: " + counter.getValue());
            int nextInput = Console.readInt();
            if (nextInput == 0) break;
            counter.add(nextInput);
        }
    }
}
public class AMultipleUI implements CounterInteractor {
    public void edit (Counter counter) {
        while (true) {
            System.out.println("Counter: ", counter.getValue());
            JOptionPane.showMessageDialog(null, "Counter: ", counter.getValue());
            int nextInput = Console.readInt();
            if (nextInput == 0) break;
            counter.add(nextInput);
        }
    }
}
DRAWBACKS OF MONOLITHIC UI

- Duplicated Input Code
- Duplicated Output Code

- Counter

- AConsoleUI
- AMixedUI
- AMultipleUI
MODEL/INTERACTOR PATTERN

UI Code

Interactor

Arbitrary UI unaware methods

Computation Code

Model
MVC Pattern

Controller

View

Model

Write Methods

Read Methods

Performs Input

Performs Output

View can be on computer with big screen and controller on smart phone
MVC Pattern in Counter

- **Controller**
  - Performs Input
  - add()

- **Model**
  - Performs Output
  - getValue()

- **View**
  - Performs Output
CHANGING TO CONSOLE VIEW

Controller

Performs Input

Model

add()

getValue()

View

Perform Output

Counter: 1
MULTIPLE VIEWS

Performs Input

Controller

Performs Output

View

Model

add()

getValue()
**Multiple Views and Controllers**

Menus, buttons, shortcuts can be handled by different controllers (in same or different program)
In Http-based “MVC” a single view and controller exist in the browser and the model in the server. A Model cannot initiate actions in the browser so the controller directly communicates with the view.
Observer Pattern

Observable

Observers

Model

Changed model notifies views

View 1

View 2

View 3

View 4
MULTIPLE OBSERVERS/OBSERVABLES

A single battle simulation view observing
- Multiple planes
- Multiple tanks

How does observable know about its observers?
Observer registered with observable
Each observer is registered with observable

Each write method in observable calls a notification method in each observer

Notification method in observer reads model

Each student is registered with professor’s listserv

When web page is updated mail sent to students

Student reads web page if mailed information is not sufficient
MVC Pattern (Review)

- View can be on computer with big screen and controller on smart phone.
MVC Pattern in Counter (Review)

**Controller**
- Performs Input

**View**
- Performs Output

**Model**
- `add()` method
- `getValue()` method
**Observer Pattern**

**Observable**

Controller 1

Controller 2

Controller 3

Controller 4

Model

Changed model notifies views

Observers

View 1

View 2

View 3

View 4
**Notification Scheme (Review)**

Each observer is registered with observable

Each write method in observable calls a notification method in each observer

Notification method in observer reads model

Each student is registered with professor’s listserv

When web page is updated mail sent to students

Student reads web page if mailed information is not sufficient
**GENERAL NOTIFICATION SCHEME**

Observers may have multiple observables with common notification method.

Notification method parameter indicates which observable.

- Observable 1
  - Observer 1
  - Observer 2
  - Observer 3
  - Observer N

- Observable 2
NOTIFICATIONS IN MVC PATTERN

Controller

Notification Method

View

Write Methods

Performs Input

Performs Output

Model

Read Methods

Observer Registration Method
IMPLEMENTATION DEPENDENT ISSUES

How does controller know about model?

Model connection method invoked on it

By model or some other program
  • Main

Who registers observer registered with observable?

It registers itself if it knows about observable

Model registers it if it knows about observer

Some other code registers it
  • Main
Each observer is registered with observable

Each write method in observable calls a notification method in each observer

Notification method in observer reads model
public interface ObservableCounter {
    public void add (int amount) ;
    public int getValue() ;
    public void addObserver(CounterObserver observer);
    public void removeObserver(CounterObserver observer);
}

public interface CounterObserver {
    public void update(ObservableCounter counter);
}
public class AnObservableCounter implements ObservableCounter {
    int counter = 0;
    ObserverList observers = new AnObserverList();
    public void add (int amount) {
        counter += amount;
        notifyAllObservers();
    }
    public int getValue() {
        return counter;
    }
    public void addObserver(CounterObserver observer) {
        observers.addElement(observer);
        observer.update(this);
    }
    public void removeObserver(CounterObserver observer) {
        observers.removeElement(observer);
    }
    void notifyAllObservers() {
        for (int observerNum = 0; observerNum < observers.size();
            observerNum++)
            observers.elementAt(observerNum).update(this);
    }
}
public class ACounterConsoleView implements CounterObserver {
    public void update(ObservableCounter counter) {
        System.out.println("Counter: "+ counter.getValue());
    }
}
import javax.swing.JOptionPane;

public class ACounterJOptionView implements CounterObserver {
    public void update(ObservableCounter counter) {
        JOptionPane.showMessageDialog(
            null, "Counter: " + counter.getValue());
    }
}
public interface CounterController {
    public void setModel(ObservableCounter theCounter);
    public void processInput();
}
public class ACounterController implements CounterController {
    ObservableCounter counter;
    public void setModel(ObservableCounter theCounter) {
        counter = theCounter;
    }
    public void processInput() {
        while (true) {
            int nextInput = Console.readInt();
            if (nextInput == 0) break;
            counter.add(nextInput);
        }
    }
}
public static void main (String args[]) {
    ObservableCounter model = new AnObservableCounter();
    model.addObserver(new ACounterConsoleView());
    CounterController controller = new ACounterController();
    controller.setModel(model);
    controller.processInput();
}
```java
public static void main (String args[]) {
    ObservableCounter model = new AnObservableCounter();
    model.addObserver (new ACounterJOptionView());
    CounterController controller = new ACounterController();
    controller.setModel(model);
    controller.processInput();
}
```
public static void main (String args[]) {
    ObservableCounter model = new AnObservableCounter();
    model.addObserver(new ACounterJOptionView());
    model.addObserver (new ACounterConsoleView());
    CounterController controller = new ACounterController();
    controller.setModel(model);
    controller.processInput();
}
What if observer is in USA and observable in China?

Update must make a “long distance” call to read method (getValue()) to update counter state
public interface CounterObserver {
    public void update(ObservableCounter counter, int newCounterVal);
}

public class ACounterConsoleView implements CounterObserver {
    public void update(ObservableCounter counter, int newCounterVal) {
        System.out.println("Counter: " + newCounterVal);
    }
}
**ObjectEditor Update?**

```
public interface CounterObserver {
    public void update(ObservableCounter counter, int newCounterVal);
}
```

Can ObjectEditor become a view of Counter so no need to call refresh?

ObjectEditor does not know about CounterObserver and cannot implement it.
**JAVA.UTIL.OBSERVER AND OBSERVABLE**

`public interface java.util.Observer {`  
  `public void update(Observable o, Object arg);`  
`}`

“Standard” observer interface talking arbitrary change Object argument

`public class java.util.Observable {`  
  `public void addObserver(Observer o) { ... };`  
  `public void notifyObservers() { ... };`  
`}`

Model must be subclass of Observable
**CIRCULARITY**

```java
public interface ObservableCounter {
    public void add (int amount) ;
    public int getValue() ;
    public void addObserver(CounterObserver observer);
    public void removeObserver(CounterObserver observer);
}
```

Cannot compile ObservableCounter without CounterObserver and vice versa

```java
public interface CounterObserver {
    public void update(ObservableCounter counter);
}
```
**Breaking Circularity: Multiple Stages**

```java
public interface ObservableCounter {
    public void add (int amount) ;
    public int getValue() ;
}
```

CounterObserver references compiled ObservableCounter

```java
public interface CounterObserver {
    public void update(ObservableCounter counter);
}
```
public interface ObservableCounter {
    public void add (int amount) ;
    public int getValue() ;
    public void addObserver(CounterObserver observer);
    public void removeObserver(CounterObserver observer);
}

public interface CounterObserver {
    public void update(ObservableCounter counter);
}
Circularity and Breaking it

Circularity

- Two types reference each other
- Neither can be compiled without the other

General approach to breaking it

- Create both types as empty and compile them so they are known to Java
- Next add references to each other
Observers that are not views

- Spreadsheet cell
  - observes cells on which it depends
- Monitoring of appliance usage
  - Each time I do setChannel() on TV event logged
- Eclipse quiz/activity plug-in
  - Observers Eclipse events
- Any big brother app!
- Counter observer?
Observers that are not views

- Spreadsheet cell
  - observes cells on which it depends
- Monitoring of appliance usage
  - Each time I do `setChannel()` on TV event logged
- Eclipse quiz/activity plug-in
  - Observers Eclipse events
- Any big brother app!
- Counter observer?
public class ARocketLaunchingCounterObserver
    implements CounterObserver {
    public void update(ObservableCounter counter) {
        if (counter.getValue() == 0)
            launch();
    }
    private void launch() {
        System.out.println("LIFT OFF!!!");
    }
}
Instances created and composed

Console View added before rocket launching observer

AnObservable Counter

ACounterConsole View

Controller

ARocketLaunchingCounterObserver

ARocketLauncher
public static void main (String args[]) {  
    ObservableCounter model = new AnObservableCounter();  
    model.addObserver (new ACounterConsoleView());  
    model.addObserver(new ARocketLaunchingCounterObserver());  
    CounterController controller = new ACounterController();  
    controller.setModel(model);  
    controller.processInput();  
}
**Basic Notification**

```java
public interface CounterObserver {
    public void update(ObservableCounter counter);
}
```

Called when observer is updated

Updated Observable
**IMPLICIT OBSERVER**

```java
public interface CounterObserver {
    public void setObservable(ObservableCounter counter);
    public void update();
}
```

Assuming observer has only one observable.
What if observer is in USA and observable in China?

Update must make a “long distance” call to read method (getValue()) to update counter state

public interface CounterObserver {
    public void update(ObservableCounter counter);
}
package models;
public interface CounterObserver {
    public void update(ObservableCounter counter, int newCounterVal);
}
public interface java.util.Observer {
    public void update(Observable o, Object arg);
}
public interface CounterObserver {
    public void update(ObservableCounter counter, int newCounterVal);
}
public interface CounterObserver {
    public void update(ObservableCounter counter,
            int counterIncrement);
}

Notification with Change

Difference between new and old value of observable attribute

Observer may display change to user

Observer interested in change does not need to keep old value to determine change

Observer interested in absolute value must keep old value
Notification with New and Old Value

```
public interface CounterObserver {
    public void update (ObservableCounter counter, int oldCounterValue, int newCounterValue);
}
```

- Old and new value of observable attribute
- Observer interested in change does not need to keep old value to determine change
- Observer interested in absolute value need not keep old value
- Makes observer harder to code
**Notification with Single Event Object**

```java
public interface CounterObserver {
    public void update(
        CounterChangeEvent event);
}
```

```java
public interface CounterChangeEvent {
    ObservableCounter getCounter();
    int getOldCounterValue();
    int getNewCounterValue();
}
```

- Easy to pass single object to different methods handling event
- Can make event info very elaborate
  - Time when event occurred
  - Unique ID for event
  - ....
- Callee does not have to declare parameters for event information fields not of interest
- Caller does not have to fill every value – can put null for object values such as counter and illegal values for primitives
**Java Action Event**

```java
import java.awt.Event;

public interface java.awt.ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

When you edit text and hit return this event sent by JTextField, TextField widget to its listeners such as ObjectEditor.

When you press a button, this event sent by Button/Jbutton to its listeners such as ObjectEditor.
Observing Multiple Properties

```java
public interface BMISpreadsheet {
    public double getHeight();
    public void setHeight(int newVal);
    public double getWeight();
    public void setWeight(int newWeight);
    public double getBMI();
}
```
SINGLE COARSE-GRAINED UPDATE

public interface BMISpreadsheet {
    public double getHeight();
    public void setHeight(int newVal);
    public double getWeight();
    public void setWeight(int newWeight);
    public double getBMI();
    ....
}

public interface BMIObserver {
    public void update(BMISpreadsheet bmiSpreadsheet);
}

Coarse grained updated
Each setter sends the whole object
Observer must determine which property changed
MULTIPLE FINE-GRAINED UPDATES

```java
public interface BMISpreadsheet {
    public double getHeight();
    public void setHeight(int newVal);
    public double getWeight();
    public void setWeight(int newWeight);
    public double getBMI();
...
}
```

```java
public interface BMIObserver {
    public void updateHeight (BMISpreadsheet bmi, int oldHeight, int newHeight);
    public void updateWeight(BMISpreadsheet bmi, int oldWeight, int newWeight);
    public void updateBMI(BMISpreadsheet bmi, double oldBMI, double newBMI);
}
```
**SINGLE FINE-GRAINED UPDATE METHOD**

```java
public interface BMISpreadsheet {
    public double getHeight();
    public void setHeight(int newVal);
    public double getWeight();
    public void setWeight(int newWeight);
    public double getBMI();
    ...
}

public interface BMIObserver {
    public void update(
        BMISpreadsheet bmi, String propertyName,
        Object oldValue, Object newValue);
}
```

New methods not needed as new properties added

Different setters calls the same update method with different types of values.

Can be used for arbitrary property values

Can make mistakes and must process property name to determine what changed
Custom Single Fine-grained Update Method

```java
public void setHeight (int newVal) {
    int oldVal = height;
    height = newVal;
    notifyAllObservers(this, “height”, oldVal, newVal);
}
```

```java
public void notifyAllObservers(BMISpreadsheet source, String propertyName, Object oldValue, Object newValue) {
    for (int index = 0; index < observers.size(); index++) {
        observers.elementAt(index).update(source, propertyName, oldValue, newValue);
    }
}
```

BMIObserver

Can make mistake