Anatomy of an Interactive Application

Prasun Dewan
Department of Computer Science
University of North Carolina at Chapel Hill
dewan@cs.unc.edu
Code available at: https://github.com/pdewan/ColabTeaching

Single-User UI: An Input Echoer

```
Please enter an input line or quit or history
The woods are lovely dark and deep
The woods are lovely dark and deep
Please enter an input line or quit or history
But I have promises to keep
But I have promises to keep
Please enter an input line or quit or history
And miles to go before I sleep
And miles to go before I sleep
Please enter an input line or quit or history
history
The woods are lovely dark and deep, But I have promises to keep, And miles to go before I sleep

Please enter an input line or quit or history
quit
Quitting application
```

Displays arbitrary number of messages entered by the user.

Echoes input until user enters “quit”

Implementation: Main Method

```
public class MonolithicEchoer  {
protected static List<String> history = new ArrayList();
public static void main(String[] anArgs) {
  for (;;) {
    System.out.println(PROMPT);
    Scanner scanner = new Scanner(System.in);
    String nextInput = scanner.nextLine();
     if (nextInput.equals(QUIT)) {
         processQuit();
         break;
     } else if (nextInput.equals(HISTORY))
         printHistory();
     else
         processInput(nextInput);
   }
}
```

Data

UI

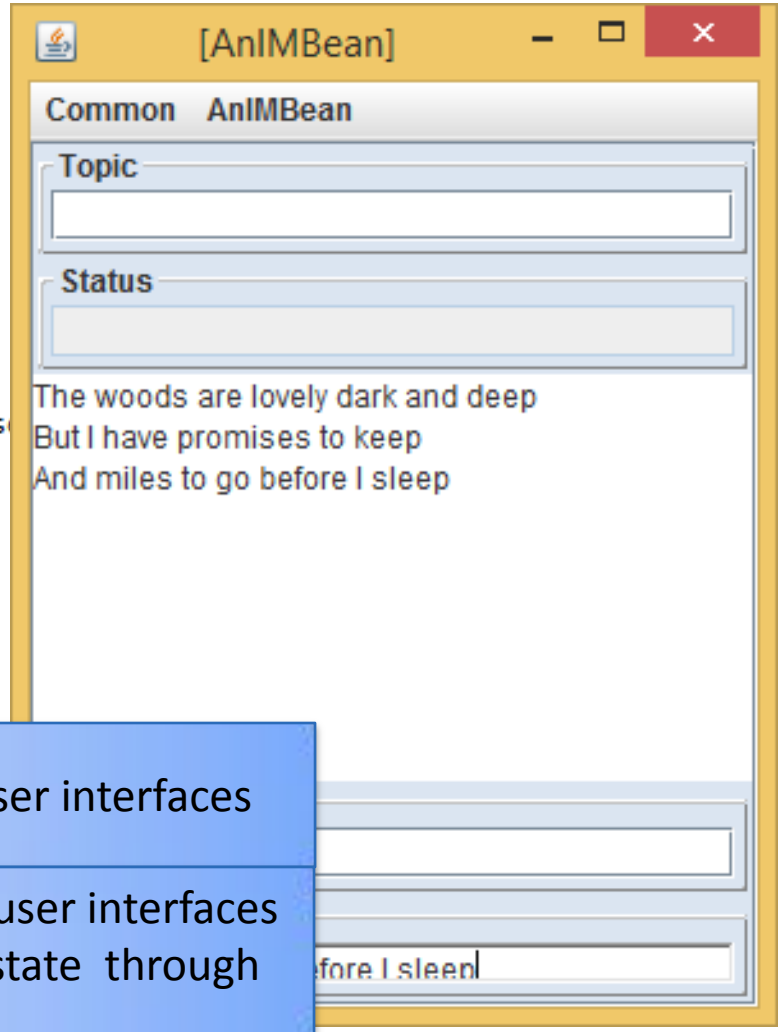
Issues with this implementation?

The UI and data are both in one class

Multiple User-Interfaces

```
Please enter an input line or quit or history
The woods are lovely dark and deep
The woods are lovely dark and deep
Please enter an input line or quit or history
But I have promises to keep
And miles to go before I sleep
history
The woods are lovely dark and deep, But I have promis

Please enter an input line or quit or history
```

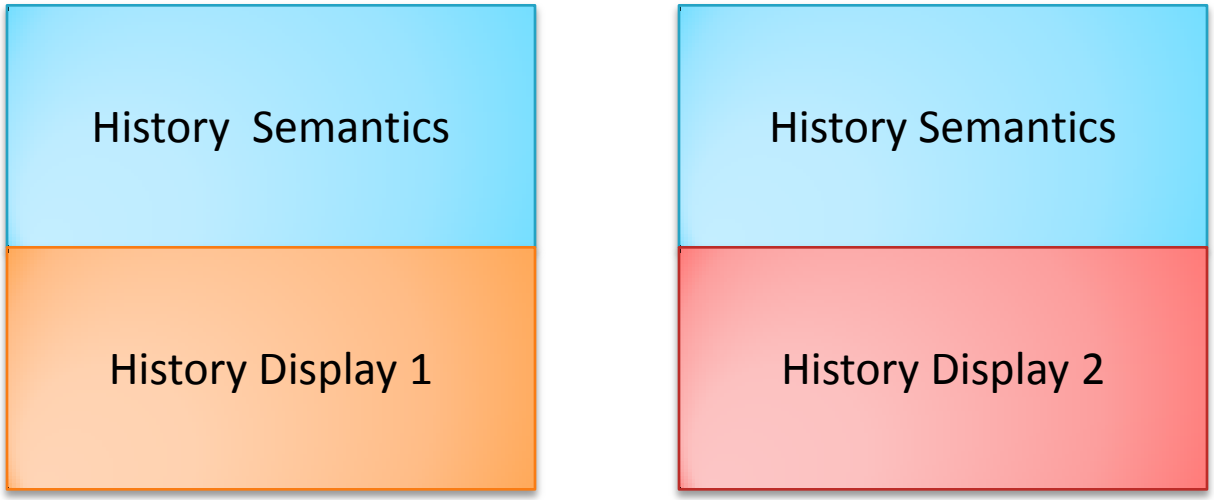


A single user may or may not want multiple user interfaces

Different users are very likely to want multiple user interfaces unless all of them are sharing the exact same state through desktop/window sharing

Collaboration ➔ Data/UI Separation

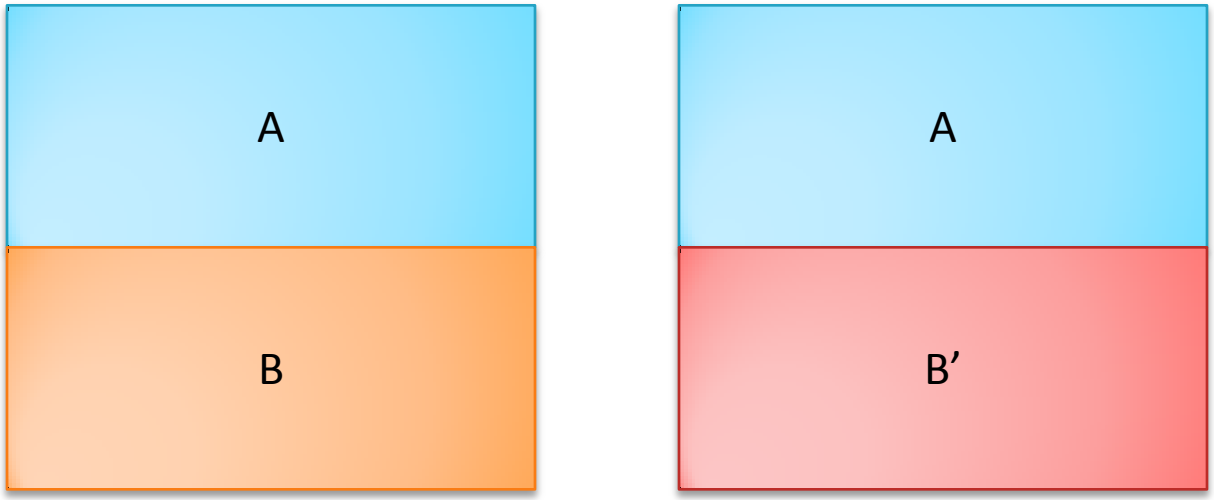
Separation of Concerns



Can change display without changing other aspects of history

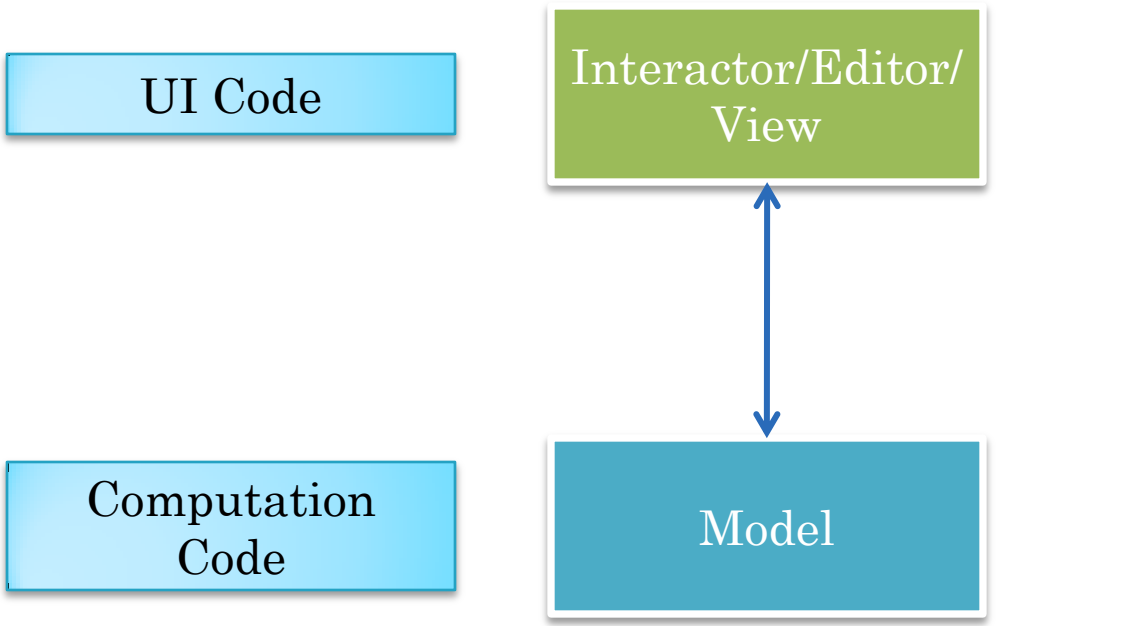
Display and semantics should go in different classes

Separation of Concerns



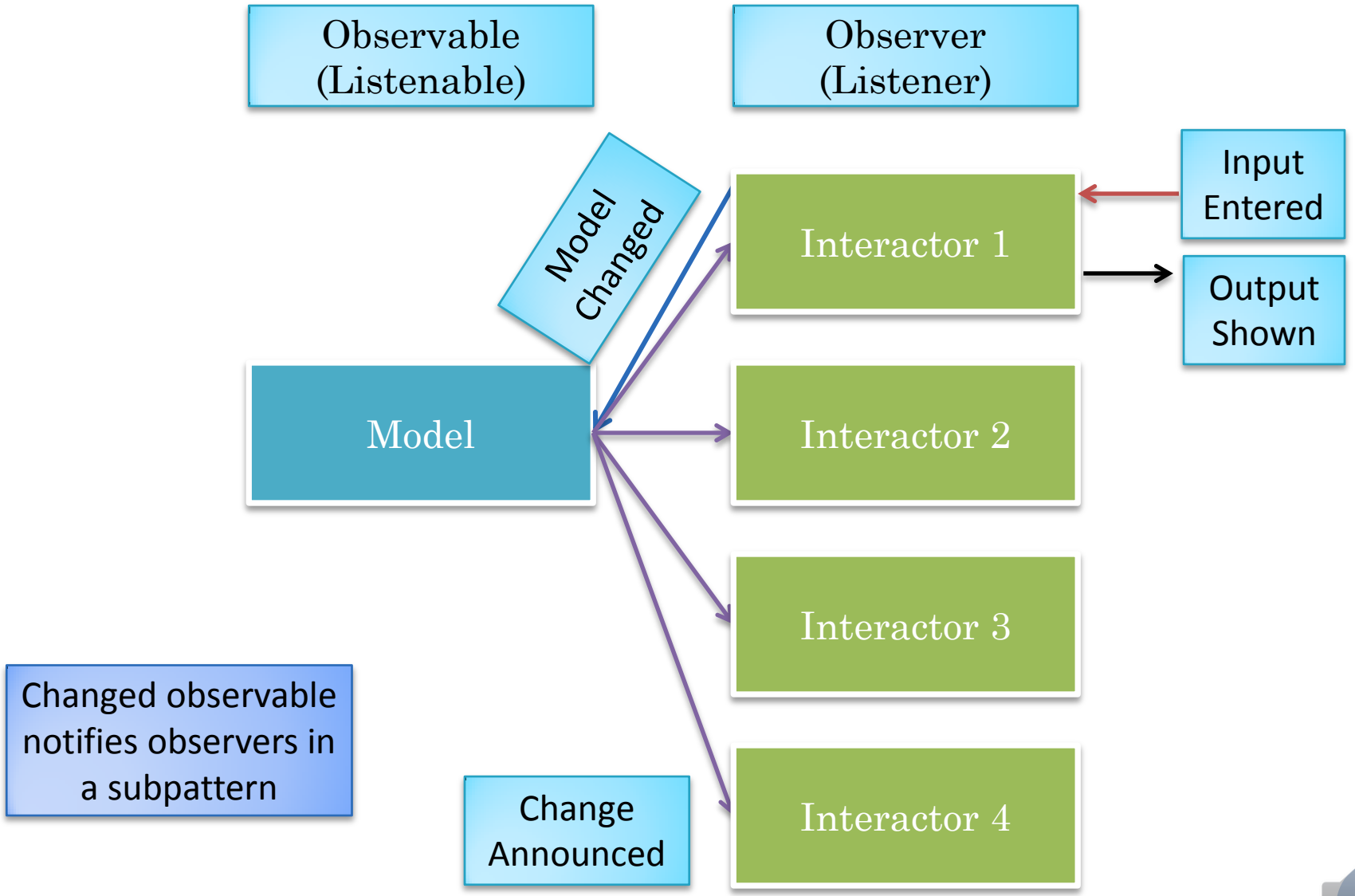
if a part B of a class can be changed without changing some other part A of the class, then refactor and put A and B in different classes

Model/Interactor(Editor, View) Pattern



Interactor(Editor, View) = MVC View + MVC Controller

Multiple Consistent Interactors



List Model

```
public class ASimpleList<ElementType>
        implements SimpleList<ElementType> {
  List<ElementType> simpleList = new ArrayList();
  List<ListObserver<ElementType>> observers = new ArrayList();
  public void add(ElementType anElement) {
    simpleList.add(simpleList.size(), anElement);
  }
  public void observableAdd(int anIndex, ElementType anElement
    add(anIndex, anElement);
    notifyAdd(anIndex, anElement);
  }
  public void notifyAdd(List<ListObserver<ElementType>> obser
    int index, ElementType newValue) {
    for (ListObserver<ElementType> observer:observers)
      observer.elementAdded(index, newValue);
   }
…
}
```

History ArrayList encapsulated

Observer list

Model Changed

Change Announced

UI Unaware Code

List Interactor

```
public class AnEchoInteractor implements EchoerInteractor {
  protected SimpleList<String> history;
  public AnEchoInteractor(SimpleList<String> aHistory) {
    history = aHistory;
  }
  …
  protected void processInput(String anInput) {
    addToHistory(computeFeedback(anInput));
  }
  protected void addToHistory(String newValue) {
     history.observableAdd(newValue);
  }
  public void elementAdded(int anIndex, Object aNewValue) {
     displayOutput(history.get(anIndex));
  }
  protected void displayOutput(String newValue) {
    System.out.println(newValue);
  }
…
```

Model bound

Input Entered

Model Changed

Change Announced

Output Displayed

Who created and connected the model and interactor

Composer

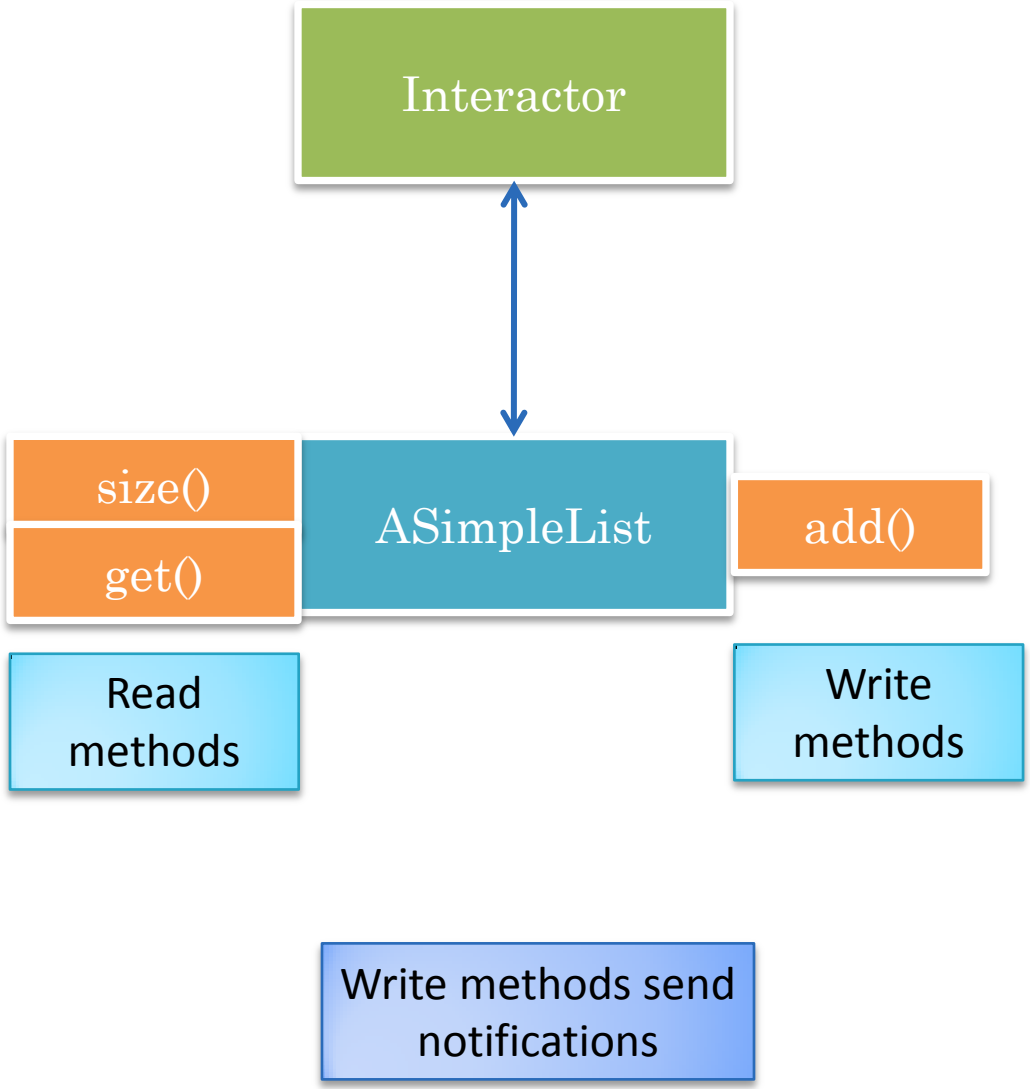
```
public class AnEchoComposerAndLauncher implements
EchoerComposerAndLauncher{
  protected SimpleList<String> history;
  protected EchoerInteractor interactor;
  // factory method
  protected SimpleList<String> createHistory() {
    return new ASimpleList();
  }
  // factory method
  protected EchoerInteractor createInteractor() {
    return new AnEchoInteractor(history);
  }
  protected void connectModelInteractor() {
    interactor = createInteractor();
    history.addObserver(interactor);
  }
…
```

Model Created

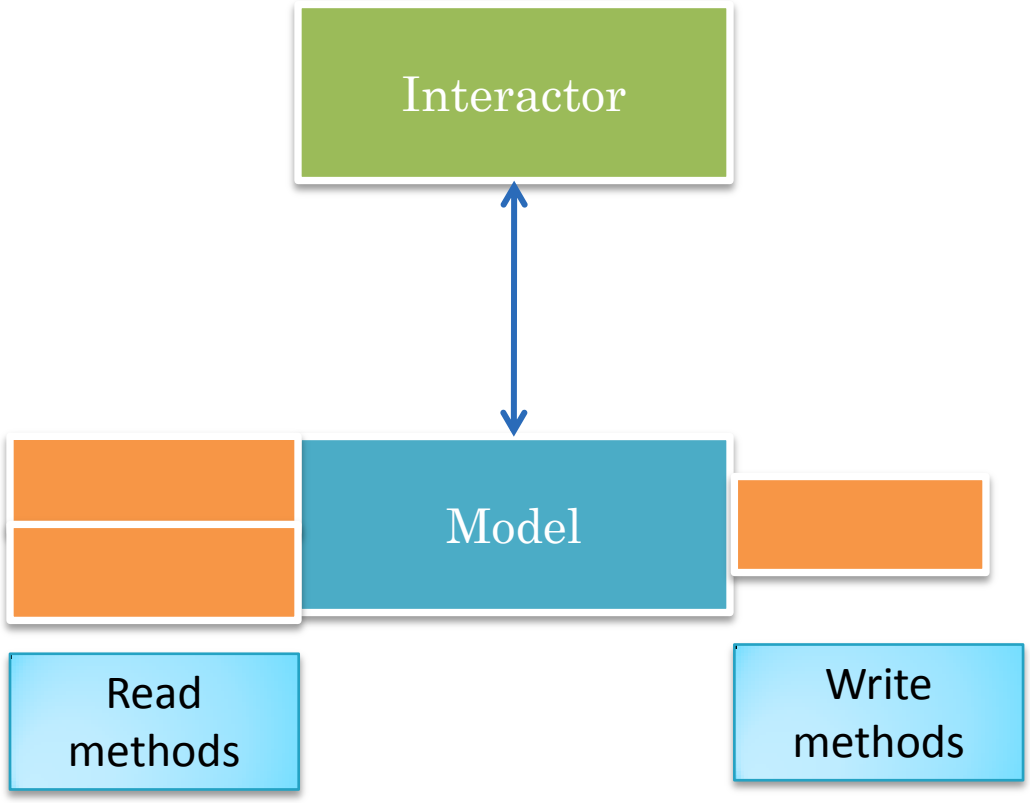
Interactor Created

Interactor Registered

Example

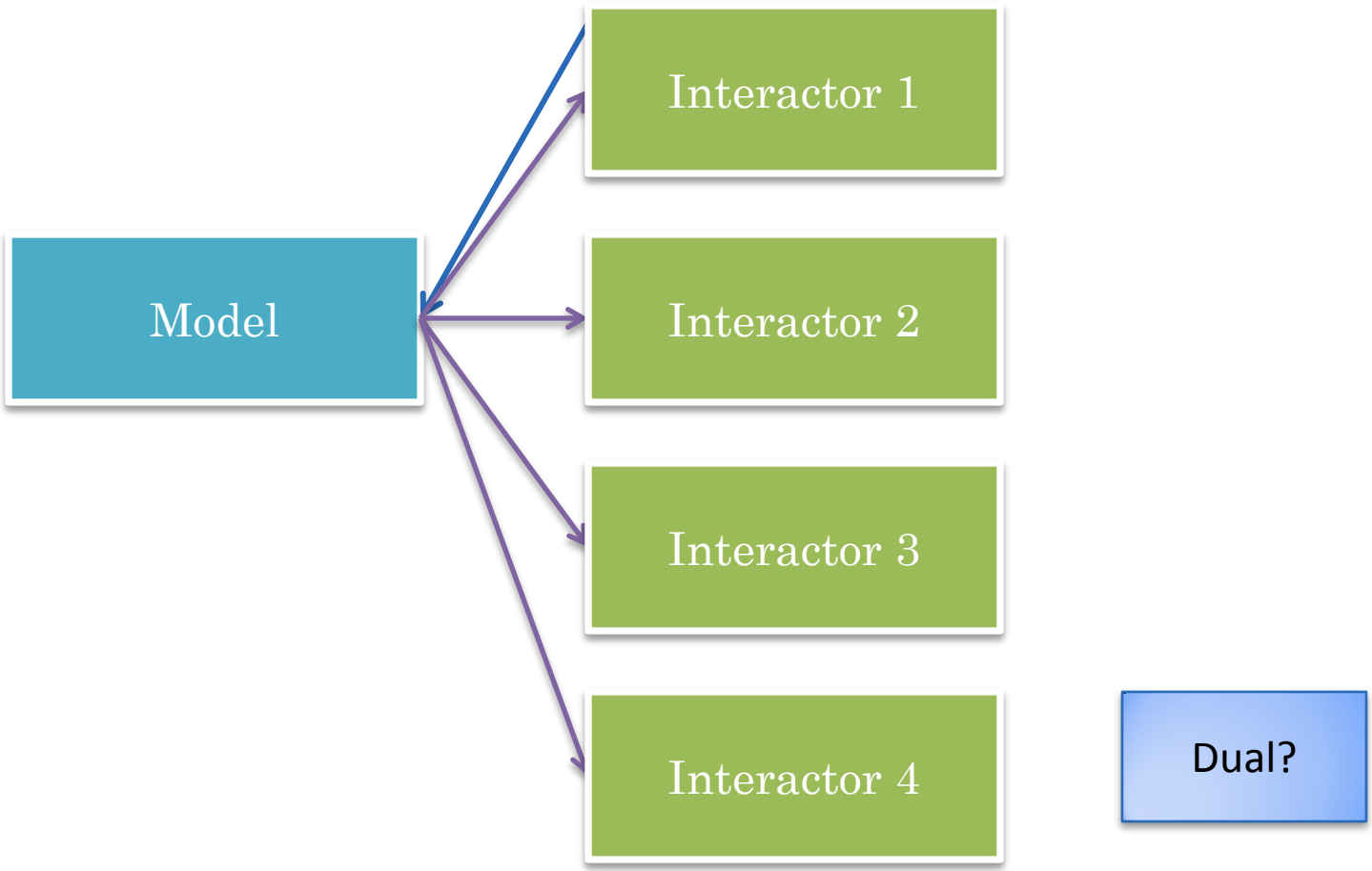

Interactor
size()
get()
ASimpleList
add()
Read methods
Write methods
Write methods send notifications

General Pattern

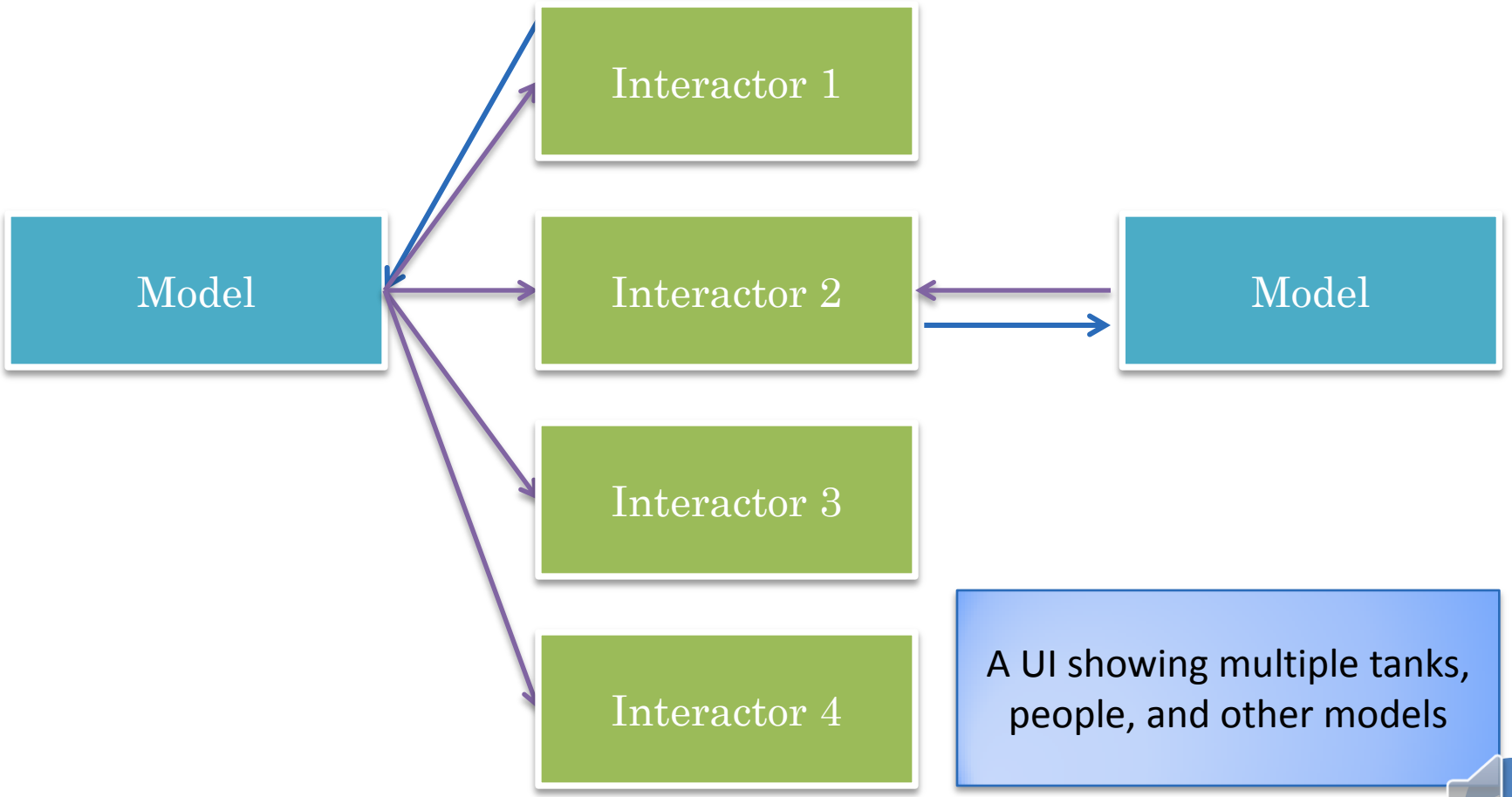


Write methods send notifications

Model with Multiple Interactors



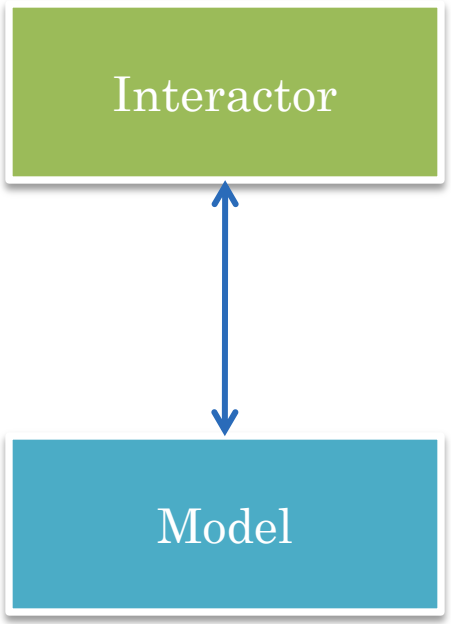
Interactors with Multiple Models



Summary of Concepts in Creating Extensible Interactive Application

- Collaborative ➔ Multi-View Apps
- Separation of Computation and UI Code
- Model /Interactor Pattern
- Observer/Observable Subpattern
- A model can have multiple interactors and vice versa

Next



Types of interactors?

Types of models?