COMP 303

Lecture 17

Inversion of control

Winter 2025

slides by Jonathan Campbell

Announcements

- Milestone 2 meeting schedule posted.
- Changes made to 303MUD repo (please pull frequently).
- Group coding session tomorrow (Wed.), 5:30-7 in TR3120

Today

- Inheritance
  - **DECORATOR** pattern with inheritance
  - When not to use inheritance
- Inversion of control
  - Motivation: View synchronization / MVC
  - **OBSERVER** pattern

Recap

Template Method pattern

```
public void perform() {
   aModel.pushMove(this); // step 1

   // step 2
   /* Code here to actually perform the move */

   log(); // step 3
}
```

Many different implementations of an algorithm, but all with the same steps.

In this particular example, step 1 and 3 are shared amongst all implementations, while step 2 is different.

Template Method pattern

```
public abstract class AbstractMove implements Move {
   protected final GameModel aModel;
   protected AbstractMove(GameModel pModel) {
      aModel = pModel;
   }
   public final void perform() {
      aModel.pushMove(this);
      execute(); // subclasses will override
      log();
   }
   protected abstract void execute();
   private void log() {
      System.out.println(getClass().getName());
   }
}
```

Defined as final, so that it cannot be overridden by subclasses.

execute is defined as protected, so that subclasses can override.

Also defined as abstract, because there is no behaviour for an abstract move.

Polymorphic copying w/ subclasses

- To support polymorphic copying, we must override copy() in all leaf classes.

```
public MemorizingDeck copy() {
  MemorizingDeck deck = new MemorizingDeck();
  deck.aCards = new CardStack(aCards); // error
  deck.aDrawnCards = new CardStack(aDrawnCards);
  return deck;
}
```

- Problem: We can't correctly update the aCards field in our new object, because it is private to Deck.

Solution: implement Cloneable

- To fix this, we will make all classes in the class hierarchy (i.e., everything implementing CardSource, including subclasses) implement the Cloneable interface.

- The interface defines a single method, clone(), which we must override.

clone

```
public Deck clone() {
   Deck clone = (Deck) super.clone();
   clone.aCards = new CardStack(aCards);
   return clone;
}
```

- Here, after calling to super.clone, we make our own copy of any mutable fields before returning.

Inheritance vs. composition

- Composition-based designs generally provides more runtime flexibility.

- Because we can change the configuration of our composite object at runtime (we can switch the Deck being memorized to something else, if the field was of type CardSource).

- But they provide fewer options for detailed access to the internal state of a well-encapsulated object (since it would be private).

Inheritance vs. composition

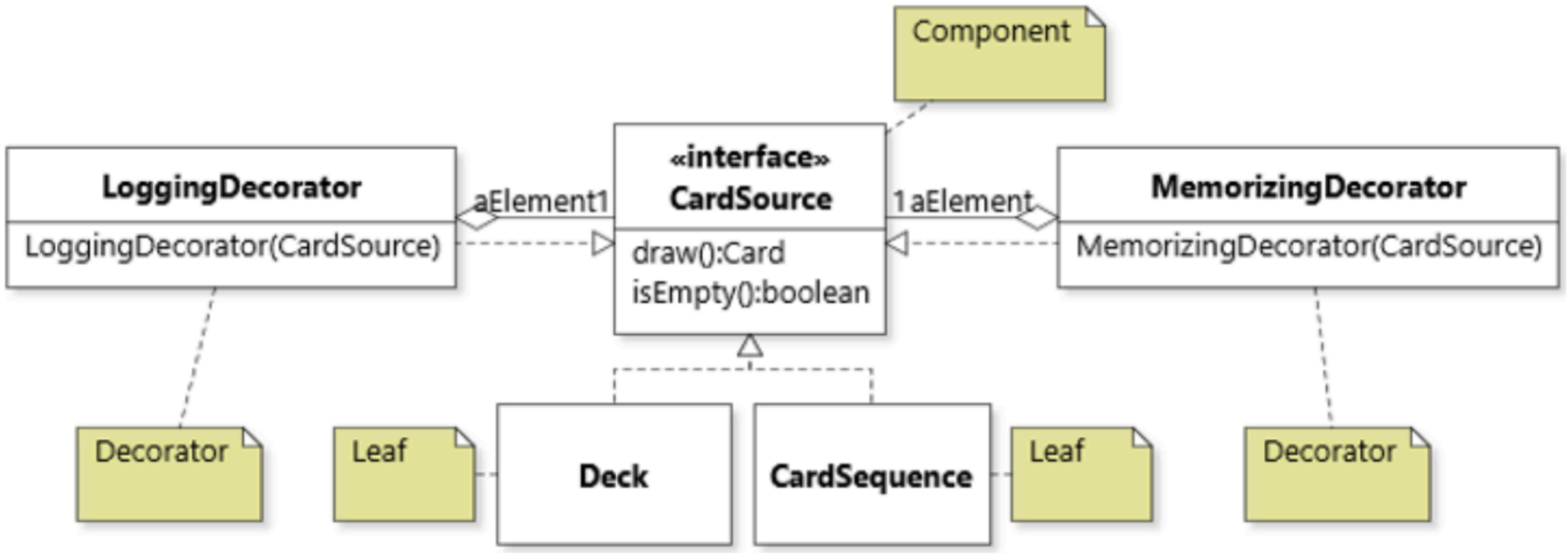
- Inheritance-based designs tend to have more flexible compile-time configuration.

- Since we can designate internal state to be protected instead of private, the subclass can access it.

- They also support finer-grained polymorphism (e.g., different compile-time types for variables), while in composition, we typically work with a common interface or abstract type (e.g., CardSource).

Decorator w/ inheritance

Revisiting DECORATOR

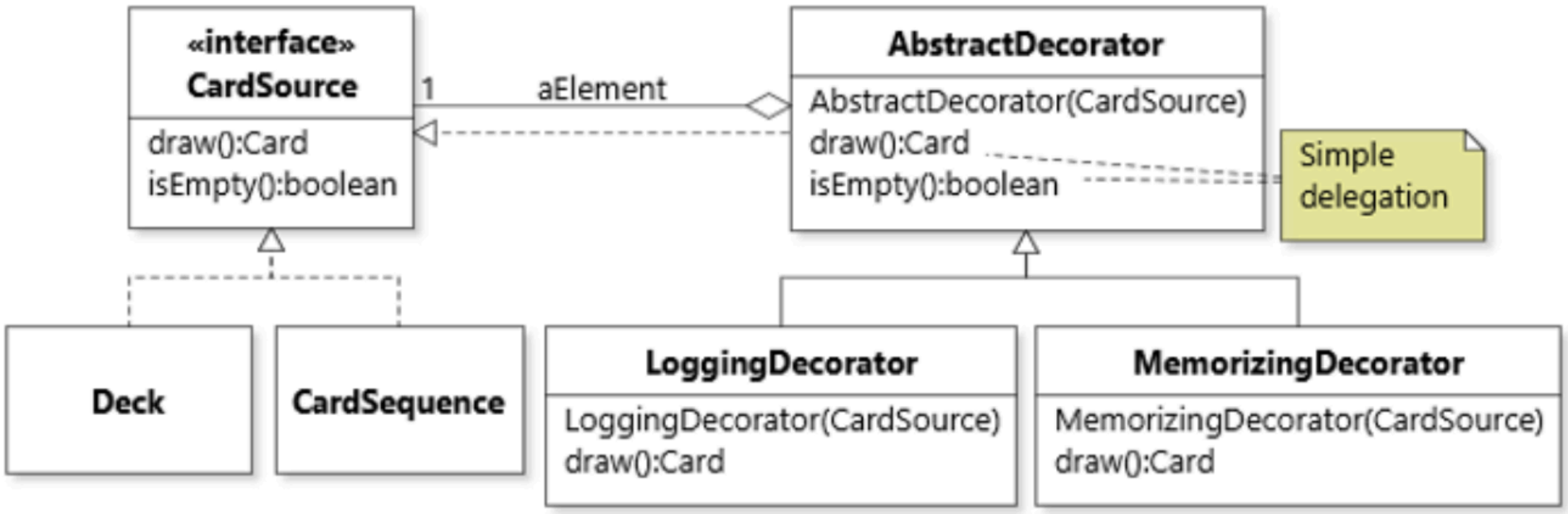


Each Decorator class must aggregate an object of the component interface (aElement).

Also, each Decorator must implement isEmpty, which will probably be the same for all decorators.

This duplicate code can be reduced using inheritance.

Revisiting DECORATOR



Now the object to be decorated (aElement) is defined in the AbstractDecorator class, and default delegation is implemented there also.

AbstractDecorator

```
public abstract class AbstractDecorator implements CardSource {
  private final CardSource aElement;
  protected AbstractDecorator(CardSource pElement) {
    aElement = pElement;
  }
  public Card draw() {
    return aElement.draw();
  }
  public boolean isEmpty() {
    return aElement.isEmpty();
  }
}
```

Decorated element is private.

Methods implementing the interface just delegate to the decorated element, with no special behaviour.

AbstractDecorator

```
public class LoggingDecorator extends AbstractDecorator {
  public LoggingDecorator(CardSource pElement) {
    super(pElement);
  }
  public Card draw() {
    Card card = super.draw();
    System.out.println(card);
    return card;
  }
}
```

Call AbstractDecorator#draw (the default behaviour).

Then add the special behaviour.

isEmpty is not implemented;
we inherit the AbstractDecorator version.

When not to use inheritance

When not to use inheritance

- **Liskov Substitution Principle** (in summary): Subclasses should not restrict what clients of the superclass can do with an instance.

Example: LSP violation

```
public class UnshufflableDeck extends Deck {
   public void shuffle() {
      /* Do nothing */
      // or: throw new OperationNotSupportedException();
   }
}
```

Deck can be shuffled, but UnshufflableDeck cannot be.
Violation of the LSP.

Consequence of LSP violation

- If we pass an UnshufflableDeck to the following method, the code won't work as expected, or raise an exception.

```
private Optional<Card> shuffleAndDraw(Deck pDeck) {
   pDeck.shuffle();
   if (!pDeck.isEmpty()) {
      return Optional.of(pDeck.draw());
   }
   else {
      return Optional.empty();
   }
}
```

Liskov Substitution Principle

- Per LSP, methods of a subclass:
  - cannot have stricter preconditions;
  - cannot have less strict postconditions;
  - cannot take more specific types as parameters;
  - cannot make the method less accessible (e.g., public -> protected);
  - cannot throw more checked exceptions; and
  - cannot have a less specific return type.
- (The last four are automatically checked by the compiler.)

Example: stricter precondition

```
public class Deck implements CardSource {
  protected final CardStack aCards = new CardStack();
  public Card draw() { return aCards.pop(); }
  public boolean isEmpty() { return aCards.isEmpty(); }
}
```

```
public class DrawBestDeck extends Deck {
  public Card draw() {
    Card card1 = aCards.pop();
    Card card2 = aCards.pop();
    Card high = // get highest card betw. card1 & card2
    Card low = // get lowest card betw. card1 & card2
    aCards.push(low);
    return high;
  }
}
```

DrawBestDeck#draw assumes that there are at least two cards in the Deck. We need to write a check for this.

Example: stricter precondition

```
public Card draw() {
   Card card1 = aCards.pop();
   if (isEmpty()) {
      return card1;
   }
   Card card2 = aCards.pop();
   ...
}
```

Not as elegant, and testing becomes more complicated.

Example: stricter precondition

```
public class DrawBestDeck extends Deck {
  public int size() {
    return aCards.size();
  }
  public Card draw() {
    assert size() >= 2;
    ...
  }
}
```

Another problem: Client code cannot check size() before calling draw() on a CardSource, because size is not part of the CardSource interface.

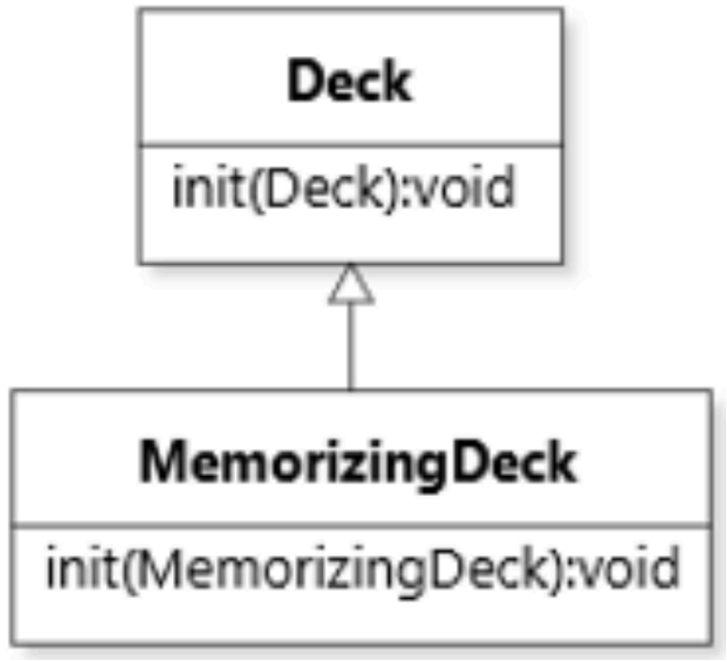
Now DrawBestDeck#draw has a stricter precondition than Deck#draw: violation of the LSP.

That's bad, because code that depends on the Deck#draw precondition will fail:

```
if (deck.size() >= 1) {
  return deck.draw();
}
```

Example: more specific parameters

Deck#init will initialize the cards in the deck to be same as the input Deck.



MemorizingDeck#init will overload Deck#init, and take a MemorizingDeck instead of Deck as input, so that it can copy both the aCards and aDrawnCards.

Example: more specific parameters

- Doing so would lead to strange behaviours.

```
MemorizingDeck deck = new MemorizingDeck();
MemorizingDeck memorizingDeck = new MemorizingDeck();
Deck mDeck = memorizingDeck;
deck.init(memorizingDeck); // Calls MemorizingDeck.init
deck.init(mDeck); // Calls Deck.init
```

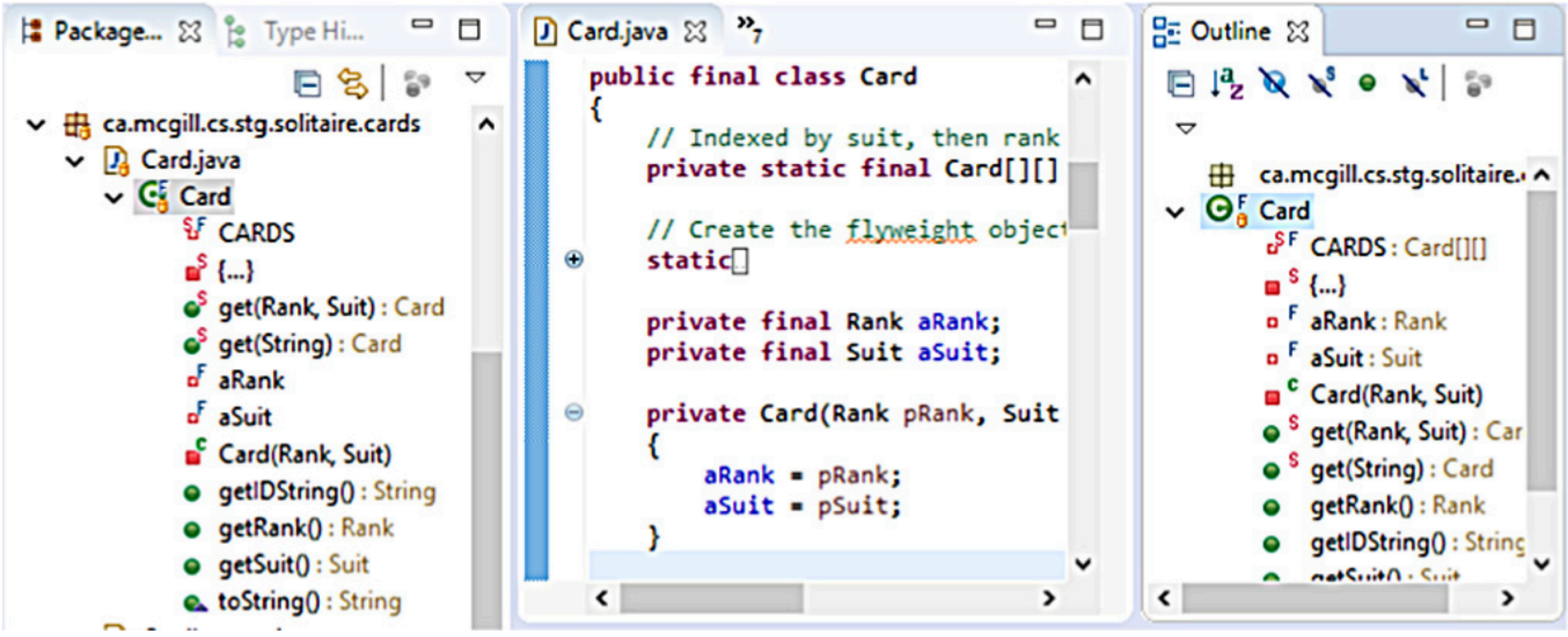
- Recall what happens when we call an overloaded method - the method called depends on the **compile-time type** of the object.

When not to use inheritance

- To make a subclass, we must require:
  - reuse of the class member declarations of the base class, and
  - a subtype-supertype relation ("is-a") between the subclass and superclass.
- If we only inherit for one purpose, but not the other, it is considered an abuse of inheritance.
  - In such case, composition should be used instead of inheritance.

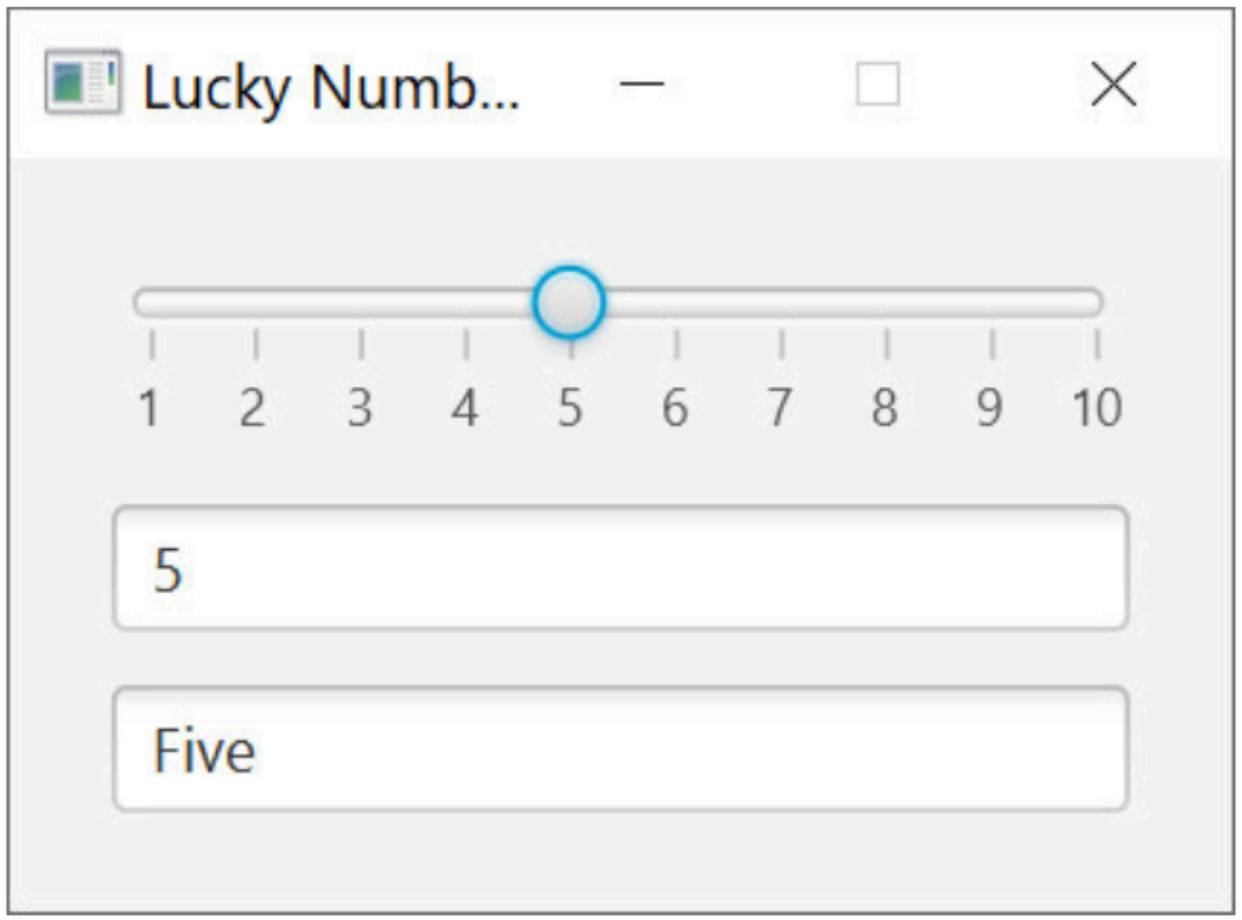
Inversion of control

View synchronization



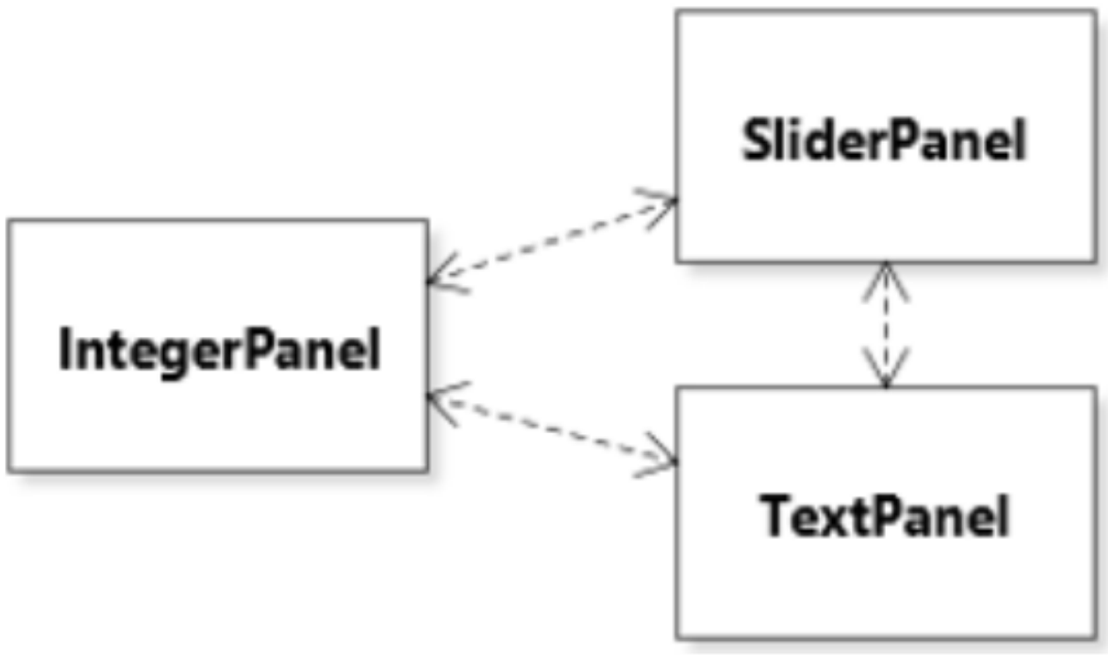
In an IDE, making a change in one view (package, code, outline) should be reflected in the other views.

View synchronization



"Lucky Number" program: the user can input their lucky number using a slider, entering a digit or the word for that number; changing any should automatically change the other two.

Pairwise Dependencies



When the user changes the number in one of the panels, the panel contacts the other panels to update their view of the number.

Anti-pattern.

Pairwise Dependencies

- Pairwise Dependencies is an anti-pattern because of:
  - **high coupling**: each panel explicitly depends on (potentially many) other panels, each maybe requiring some different method to be called (setDigit on one panel, setSliderValue on another).
  - **low extensibility**: to add or remove a panel, it is necessary to modify the code of all other panels.
- The impact of these issues increases quadratically with the number of panels.

MVC: Model-View-Controller

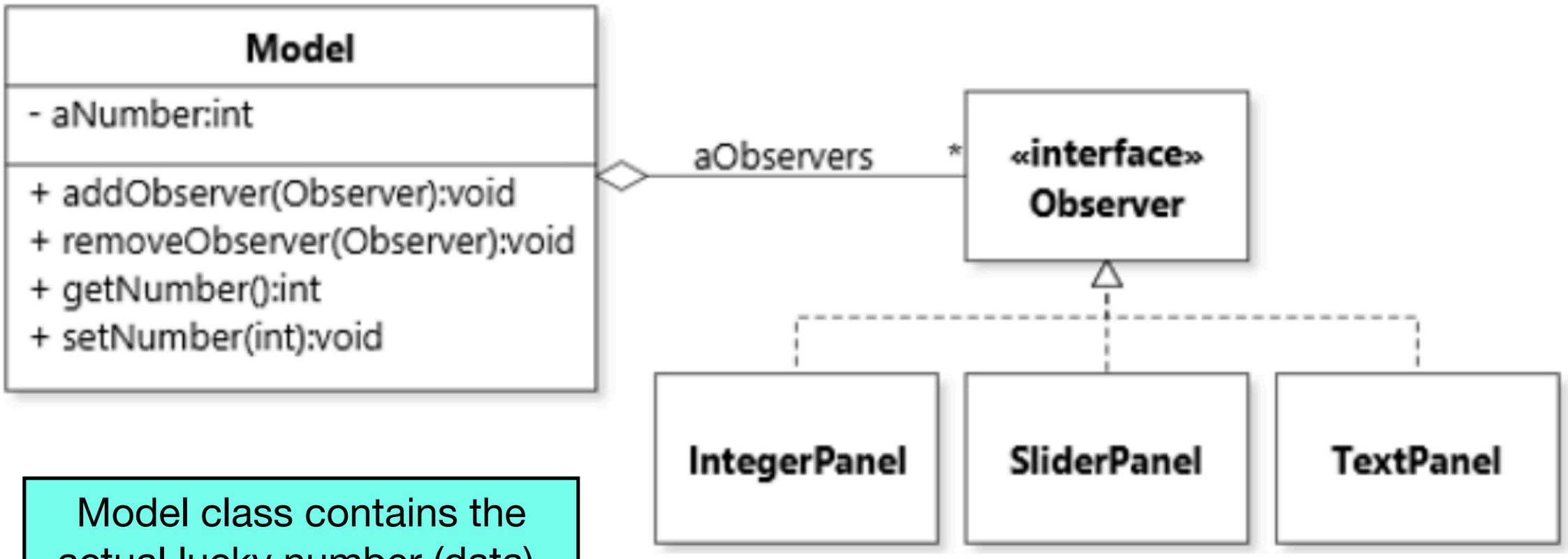
- Separate the abstraction responsible for storing data from the one responsible for changing data.
  - Model: Keeps the unique copy of the data of interest (e.g., the lucky number).
  - View: Represents a view of the data (e.g., a UI panel). (There can be multiple views.)
  - Controller: Functionality to change the data stored in the model.
- But: has no well-defined solution template.

Observer pattern

- A design pattern typically used within MVC.
- Context: we want to manage multiple objects that must be aware of state changes in the same data.
- Subject/model/observable: object that stores the data
- Observer(s): the views (e.g., panels) that must observe the data.
- (Controller is not defined in observer pattern; we could integrate it into the model class.)

OBSERVER pattern

Observer pattern for the Lucky Number example.



Model class contains the actual lucky number (data), and aggregates a number of observers, which it updates when needed.

Registering observers

```
public class Model {
  private int aNumber = 5;
  private List<Observer> aObservers = new ArrayList<>();

  public void addObserver(Observer pObserver) {
    aObservers.add(pObserver);
  }

  public void removeObserver(Observer pObserver) {
    aObservers.remove(pObserver);
  }
}

// class IntegerPanel implements Observer { ... }
```

Observers

- The Observer pattern uses polymorphism: all observers implement the Observer interface, thus allowing for loose coupling between the model and its observers:
  - the model can be used without any observer;
  - the model is aware that it can be observed, but its implementation does not depend on any concrete observer class;
  - it is possible to register and deregister observers at runtime.

Notifying observers

- The model should let the observers know whenever there is a change in the model's state worth reporting.
  - To do so, it should call some method on them, defined in the Observer interface, which is called a **callback** method.

```
public interface Observer {
  void numberChanged(int pNumber);
}
```

Callback methods

- In other words, the observers **wait** until they are notified before doing anything.
- A callback method implies an **inversion of control**: the observers do not call a method on the model, but instead wait for the model to "call" them "back".
  - The Hollywood Principle: "don't call us, we'll call you"
- The callback method is not to tell observers what to do, but rather to inform observers about some change in the model.

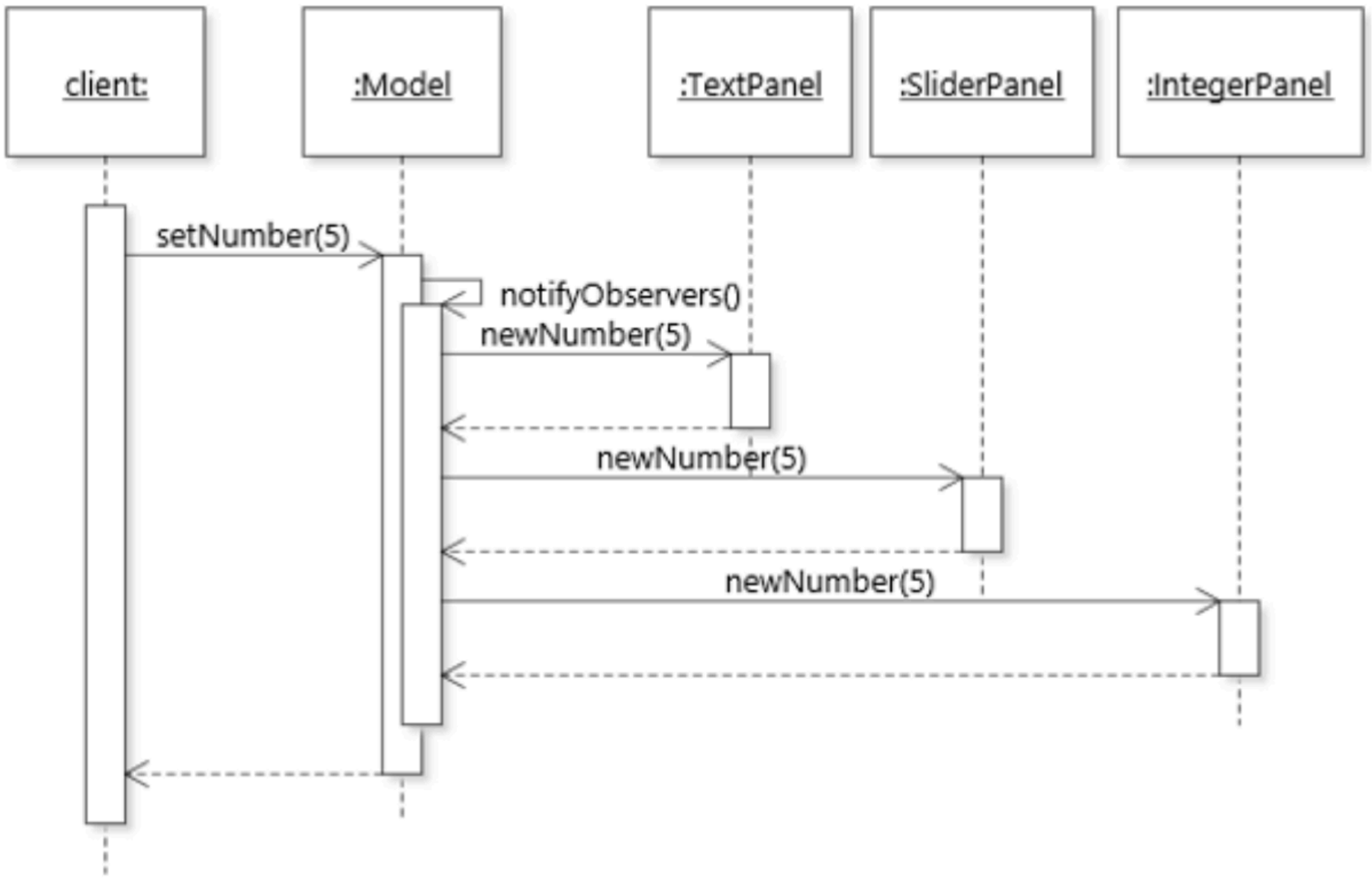
Notifying observers

```
public class Model {
   // notification method
   private void notifyObservers() {
      for (Observer observer : aObservers) {
         observer.numberChanged(aNumber);
      }
   }
}
```

Notifying observers

- When to call the notification method?
  - in every state-changing method (simplest, but possibly slow), or
  - specified in documentation that when a state-changing method is called, the notification method should be called afterwards (more flexible).
    - If we make a bunch of changes at once, we may only want to notify at the end, for example, instead of each time.

Notifying observers



Providing state to observers

- How should observers access the updated state? (Known as the data flow strategy.)

- **Push** strategy: As a parameter in the callback method (easiest, but then the particular data given to all observers is fixed), or

- **Pull** strategy: Pass the Model itself to the callback method, and the observer can use getter methods on it to access any kind of data.

Push strategy

```
public class IntegerPanel implements Observer {
  // UI element that represents a text field
  private TextField aText = new TextField();

  ...

  public void numberChanged(int pNumber) {
    aText.setText(Integer.toString(pNumber));
  }
}
```

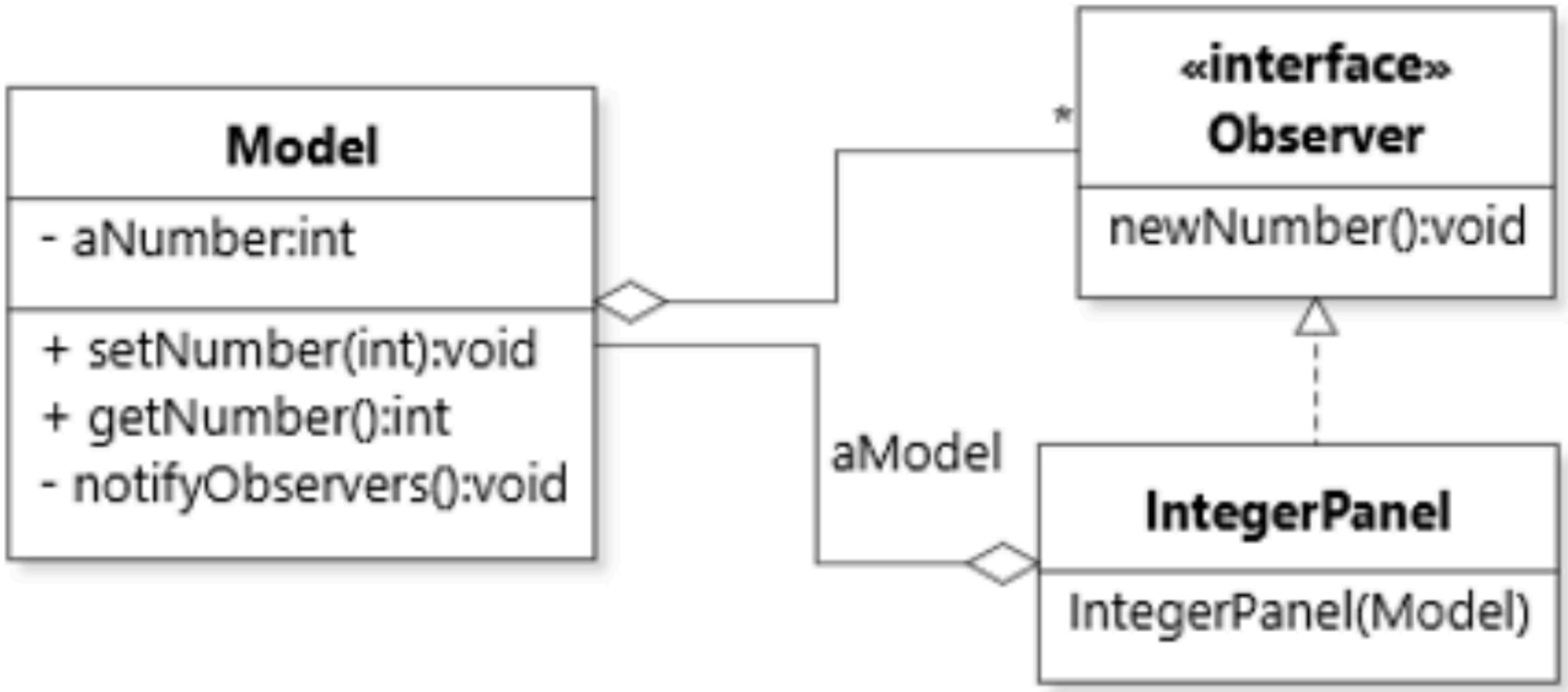
Pull strategy

```
public class IntegerPanel implements Observer {
  // UI element that represents a text field
  private TextField aText = new TextField();

  ...

  public void numberChanged(Model pModel) {
    aText.setText(Integer.toString(pModel.getNumber()));
  }
}
```

Pull strategy

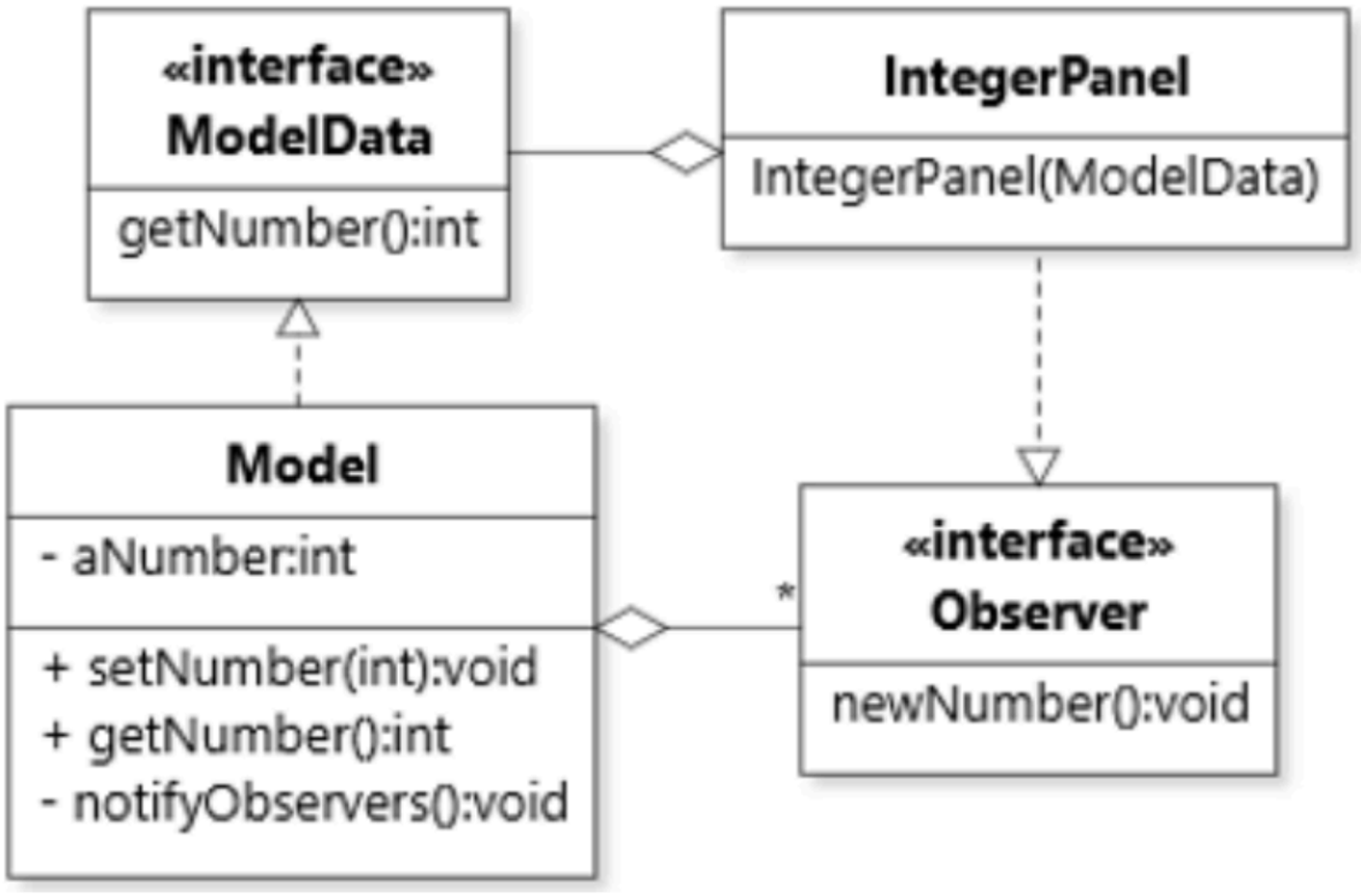


In another variant of the pull strategy, we could give the Model to the Observers when they are initialized, instead of passing to the callback each time.

Pull strategy

- Drawback of the pull strategy: the observers can call any public method, not just the getter methods. They have too much access to the model, more than they need.
  - They could call, e.g., setNumber.
- To solve this, we could create a ModelData class, which includes only the getter methods, and give this to the observers instead of the full Model.

Pull strategy



Flexibility in callback methods

- We can make our callback methods really specific if needed. For example, instead of a single numberChanged callback for our LuckyNumber, we could define:

```
public interface Observer {
   void increased(int pNumber);
   void decreased(int pNumber);
   void changedToMax(int pNumber);
   void changedToMin(int pNumber);
}
```

Flexibility in callback methods

```
public interface Observer {
   void increased(int pNumber);
   void decreased(int pNumber);
   void changedToMax(int pNumber);
   void changedToMin(int pNumber);
}
```

- If observers need to know if the value has increased, and there is only numberChanged, then they would have to store the number and check if it increased on their own. Here, it's done for them.

Flexibility in callback methods

- For observers that don't need to use some callback methods, we can make a default do-nothing callback.

```
public interface Observer {
  default void increased(int pNumber) { }
  default void decreased(int pNumber) { }
  default void changedToMax(int pNumber) { }
  default void changedToMin(int pNumber) { }
}
```

Flexibility in callback methods

- Or, we could define two observer interfaces (which would better respect the ISP, but make the Model code more complicated):

```
public interface ChangeObserver {
  void increased(int pNumber);
  void decreased(int pNumber);
}
public interface BoundsReachedObserver {
  void changedToMax(int pNumber);
  void changedToMin(int pNumber);
}
```

OBSERVER: summary

- Use when many objects need to observe some data.

- The model class stores the data, and aggregates a number of abstract observers.

References

- Robillard ch. 7-9-7.11, p.181-193.
  - Exercises #4-10: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter7.md
- Robillard ch. 8-8.3, p.195-208
  - Exercise #1-5: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter8.md

Coming up

- Next lecture:
  - Inversion of control - GUIs