McGill

COMP 303

Lecture 18

Inversion of control II

Winter 2025

slides by Jonathan Campbell

Announcements

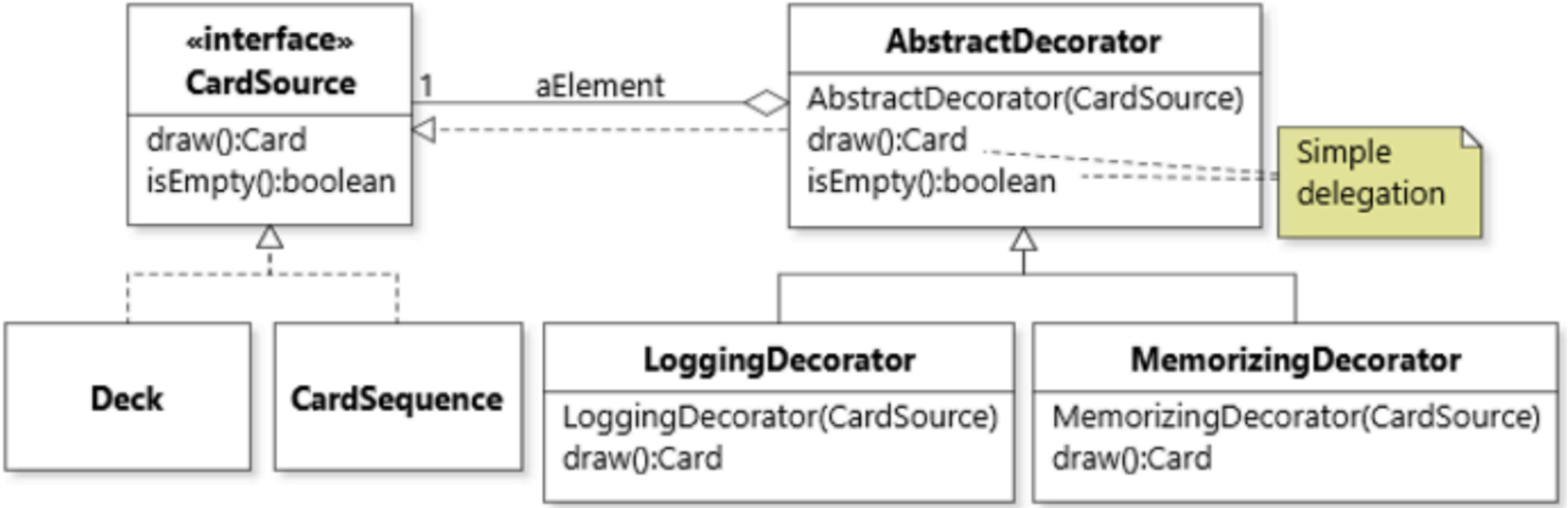
- Group coding went well yesterday!

Today

- Inversion of control
  - Observable CardStacks
  - GUIs and event handling

Recap

Revisiting DECORATOR



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

When not to use inheritance

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

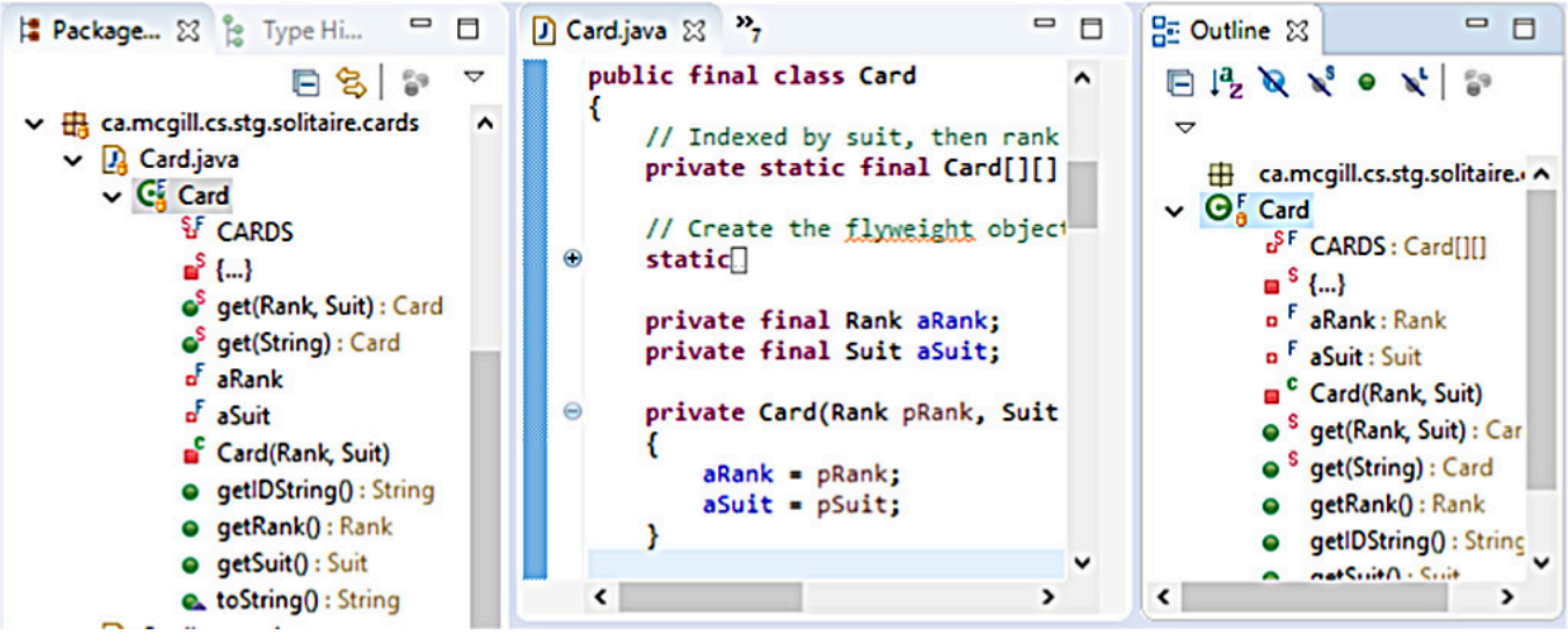
Liskov Substitution Principle

- 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.)

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.

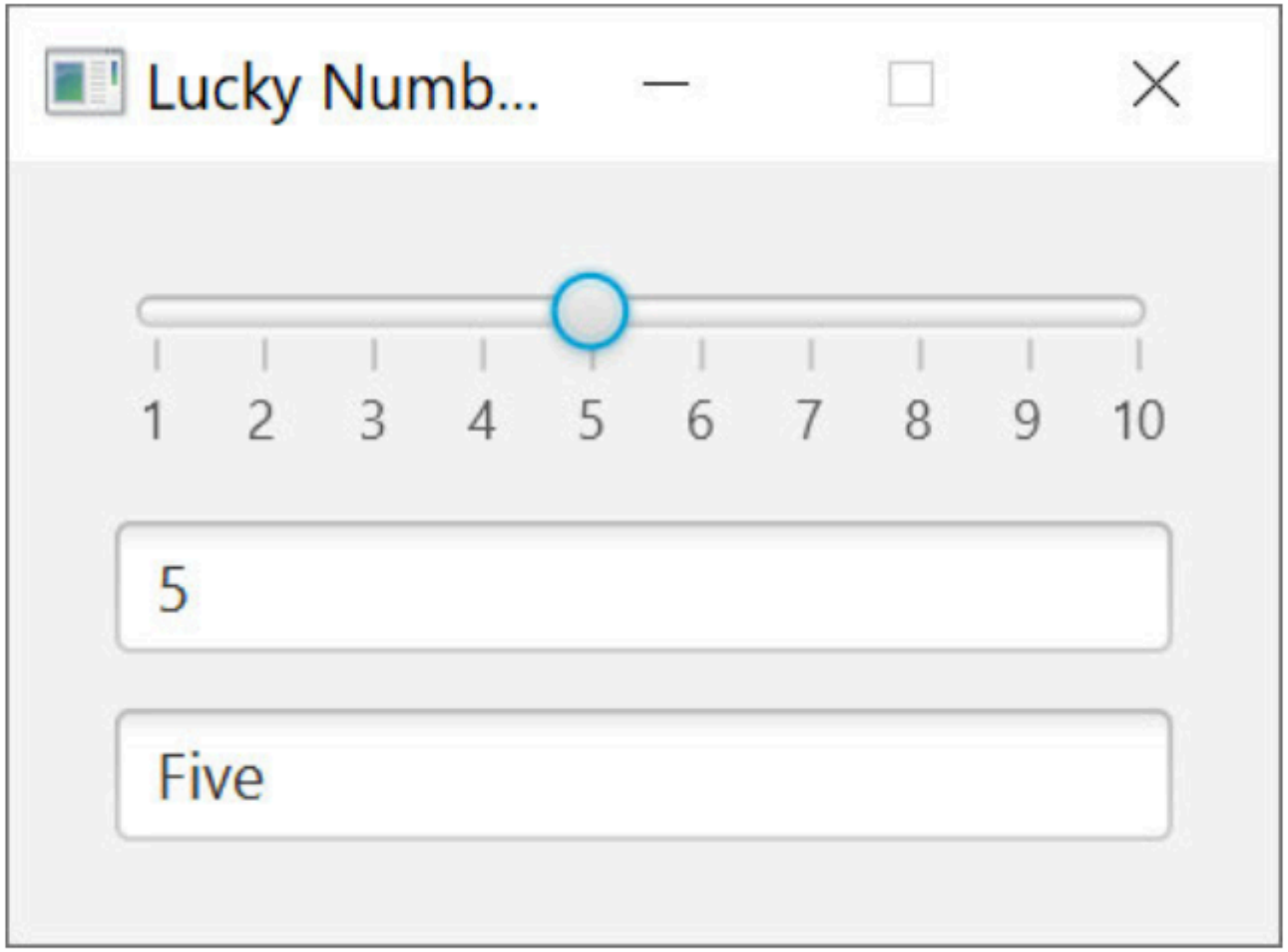
View synchronization



```
public final class Card
{
    // Indexed by suit, then rank
    private static final Card[][]

    // Create the flyweight object
    static

    private final Rank aRank;
    private final Suit aSuit;

    private Card(Rank pRank, Suit
    {
        aRank = pRank;
        aSuit = pSuit;
    }
```

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

View synchronization

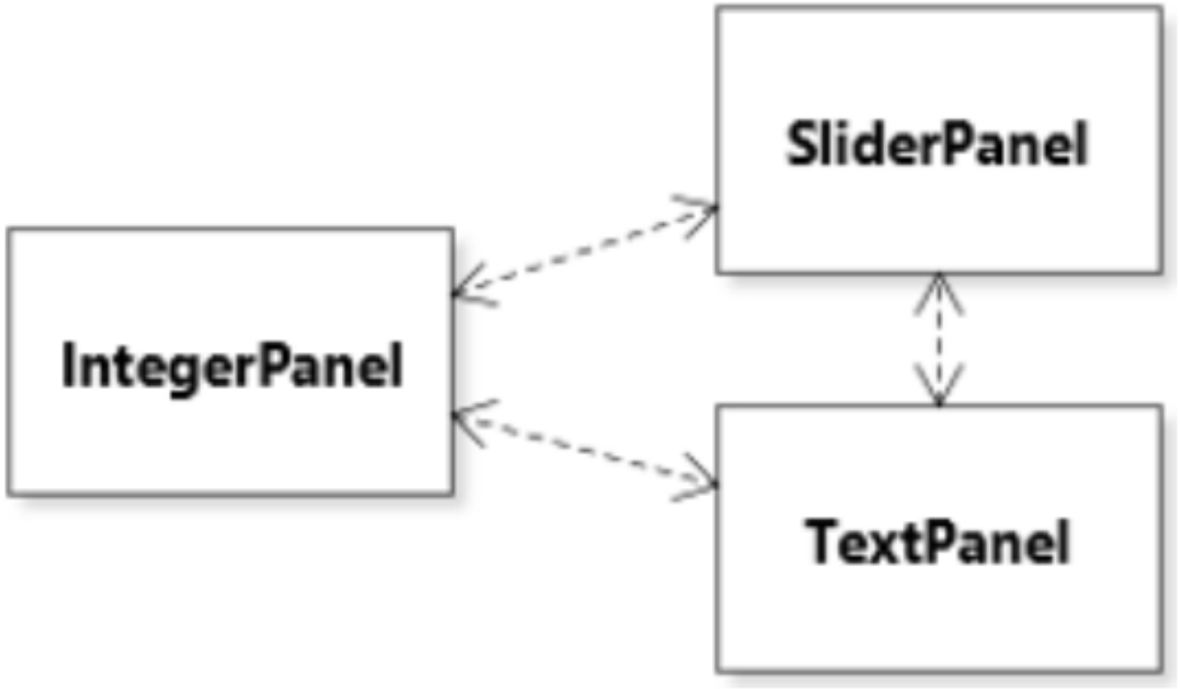


5

Five

"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

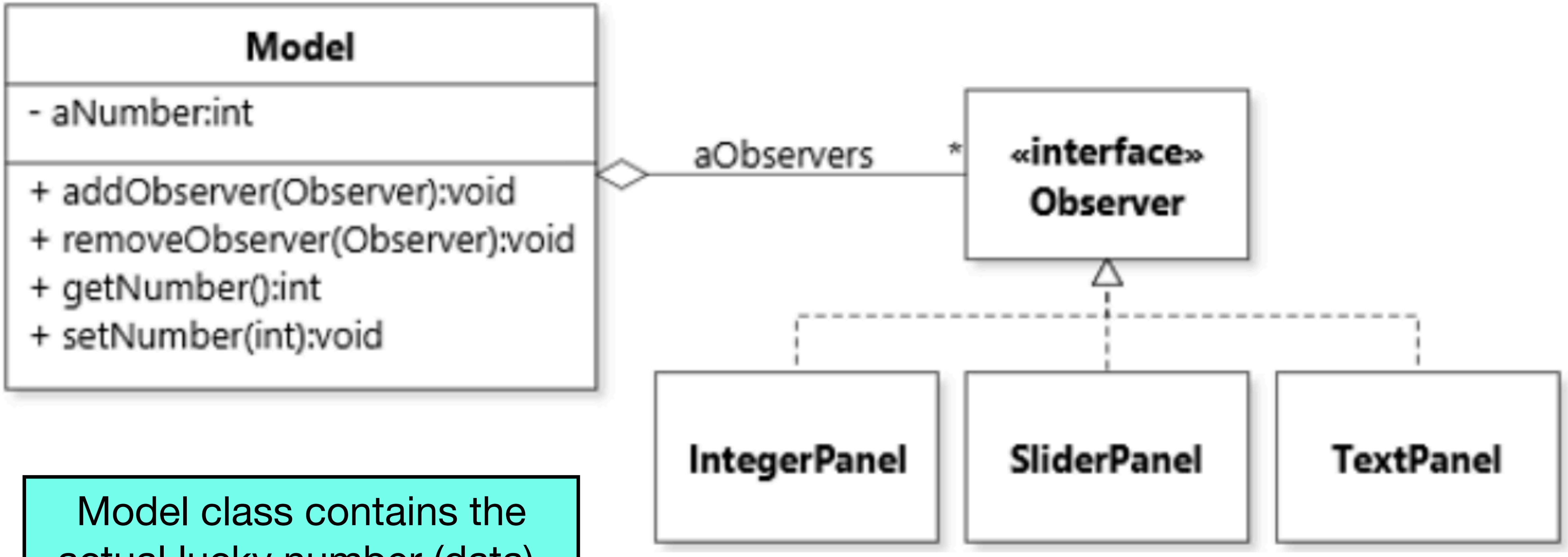


When the user changes the number in one of the panels, the panel contacts the other panels to update their view of the number.

Anti-pattern.

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.

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

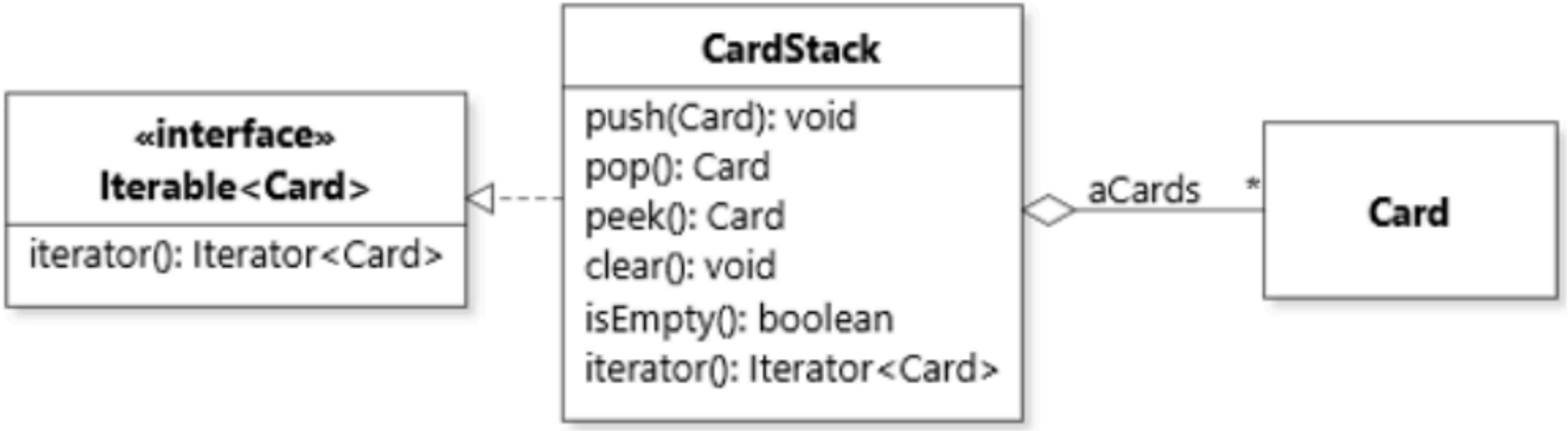
```java
public class IntegerPanel implements Observer {
  // UI element that represents a text field
  private TextField aText = new TextField();

  ...

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

Observable CardStacks

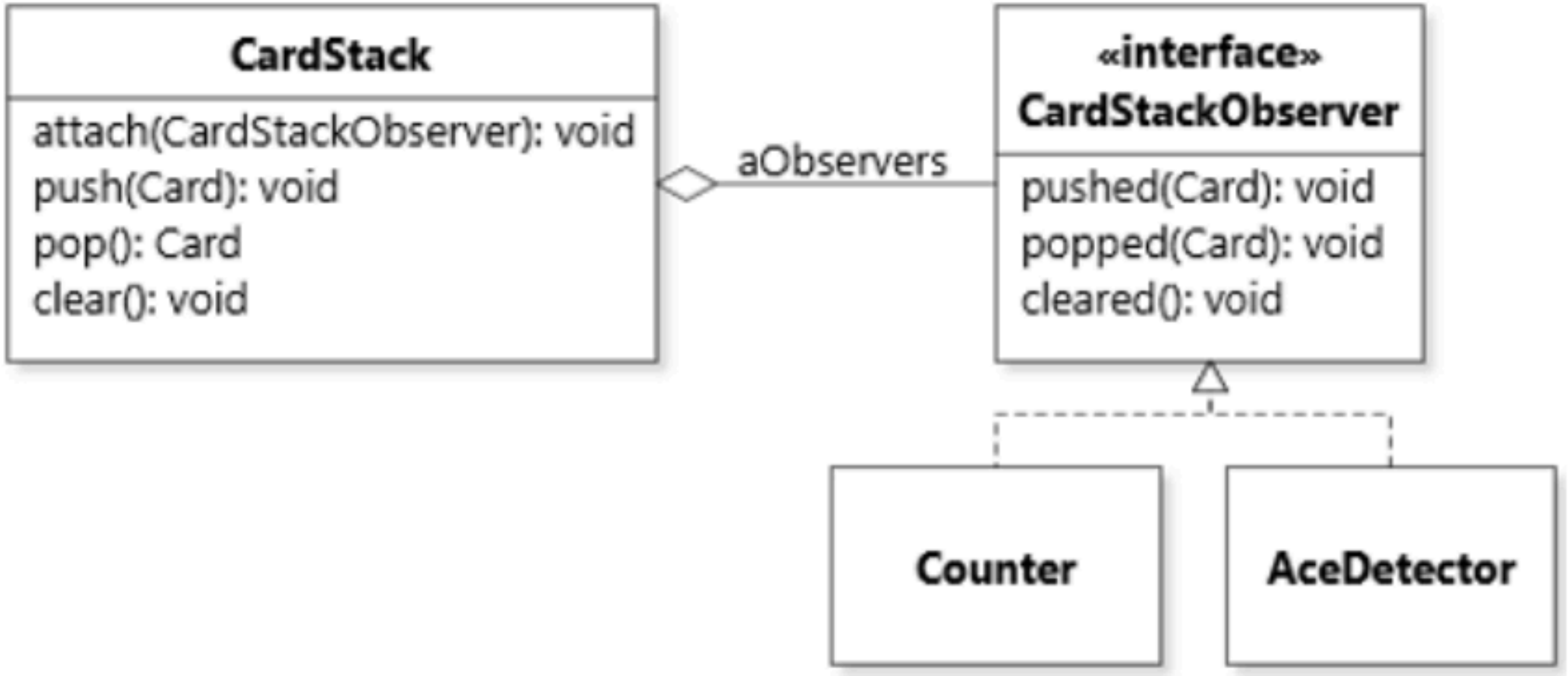
OBSERVER: design decisions

- What callback methods to implement.
- What data flow strategy (push, pull, none or both).
- How to connect observers with the model (data as parameter, Model as parameter, or ModelData).
- How to call notify (inside state-changing methods, or leave it up to the client to do so).

Example: observable CardStack



Example: observable CardStack



Counter: reports the number of cards in the stack at any point.

Ace Detector: detects whether an ace is added to the stack.

Observable CardStack

```java
public class CardStack implements Iterable<Card> {
  private final List<Card> aCards = new ArrayList<>();
  private final List<CardStackObserver> aObservers = new ArrayList<>();

  public void attach(CardStackObserver pObserver) {
    aObservers.add(pObserver);
  }

  public void push(Card pCard) {
    assert pCard != null && !aCards.contains(pCard);
    aCards.add(pCard);
    for (CardStackObserver observer : aObservers) {
      observer.pushed(pCard);
    }
  }

  // Likewise for pop() and clear()
}
```

Observable CardStack

```java
public class AceDetector implements CardStackObserver {
  public void pushed(Card pCard) {
    if (pCard.getRank() == Rank.ACE) {
      System.out.println("Ace detected!");
    }
  }
  public void popped(Card pCard) {}
  public void cleared() {}
}
```

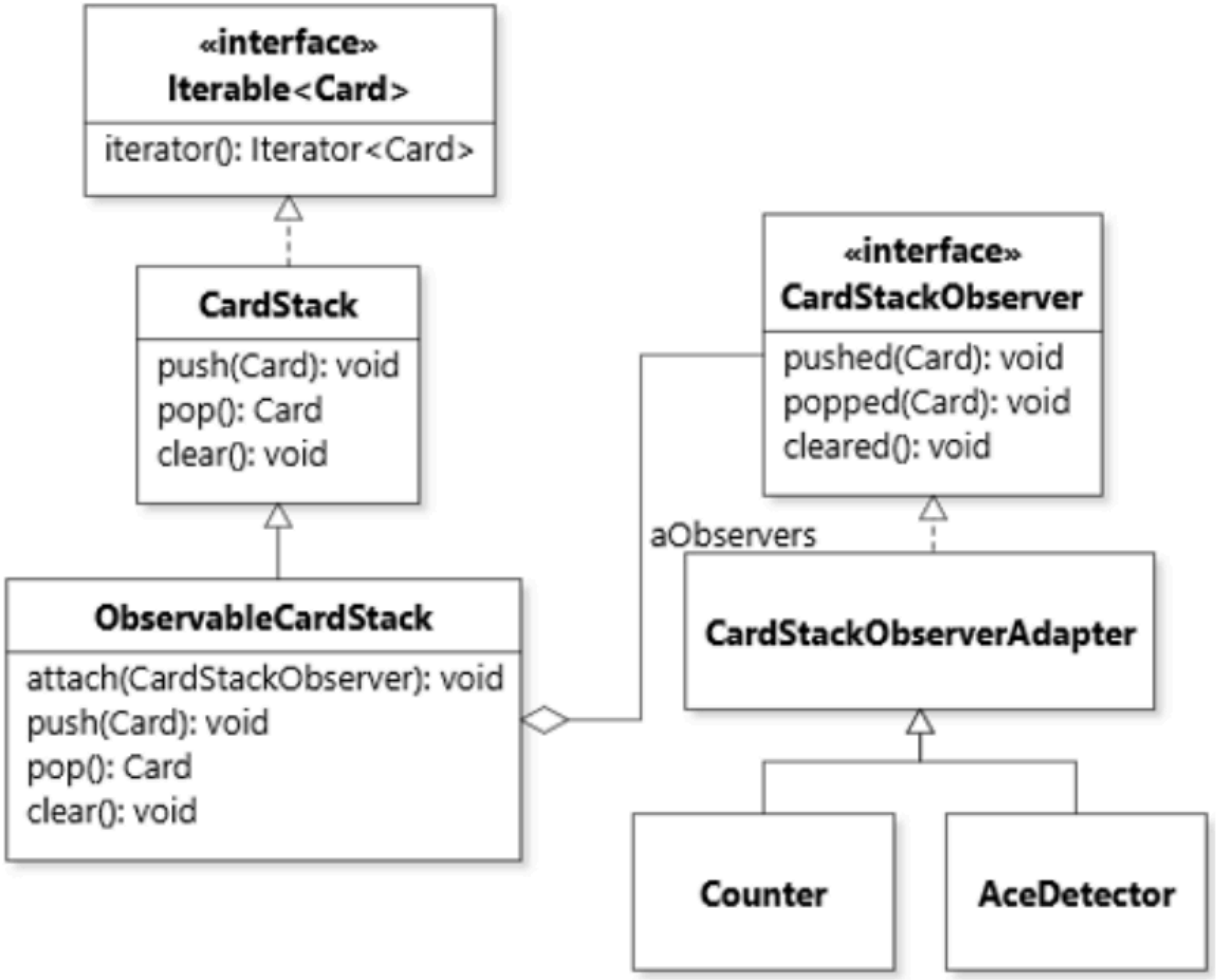
Observable CardStack

```java
public class Counter implements CardStackObserver {
  private int aCount = 0;
  public void pushed(Card pCard) {
    aCount++;
    System.out.println("PUSH Counter=" + aCount);
  }
  public void popped(Card pCard) {
    aCount--;
    System.out.println("POP Counter=" + aCount);
    if (aCount == 0) {
      System.out.println("Last card popped!");
    }
  }
  public void cleared() {
    aCount = 0;
    System.out.println("CLEAR Counter=" + aCount);
  }
}
```

Design with inheritance

- It's possible we may have some CardStacks that we want to be observable, and others not.
- For a more flexible design, we can decouple the CardStack from the observer code (attach/notify methods) by making an ObservableCardStack that inherits from CardStack.

Design with inheritance

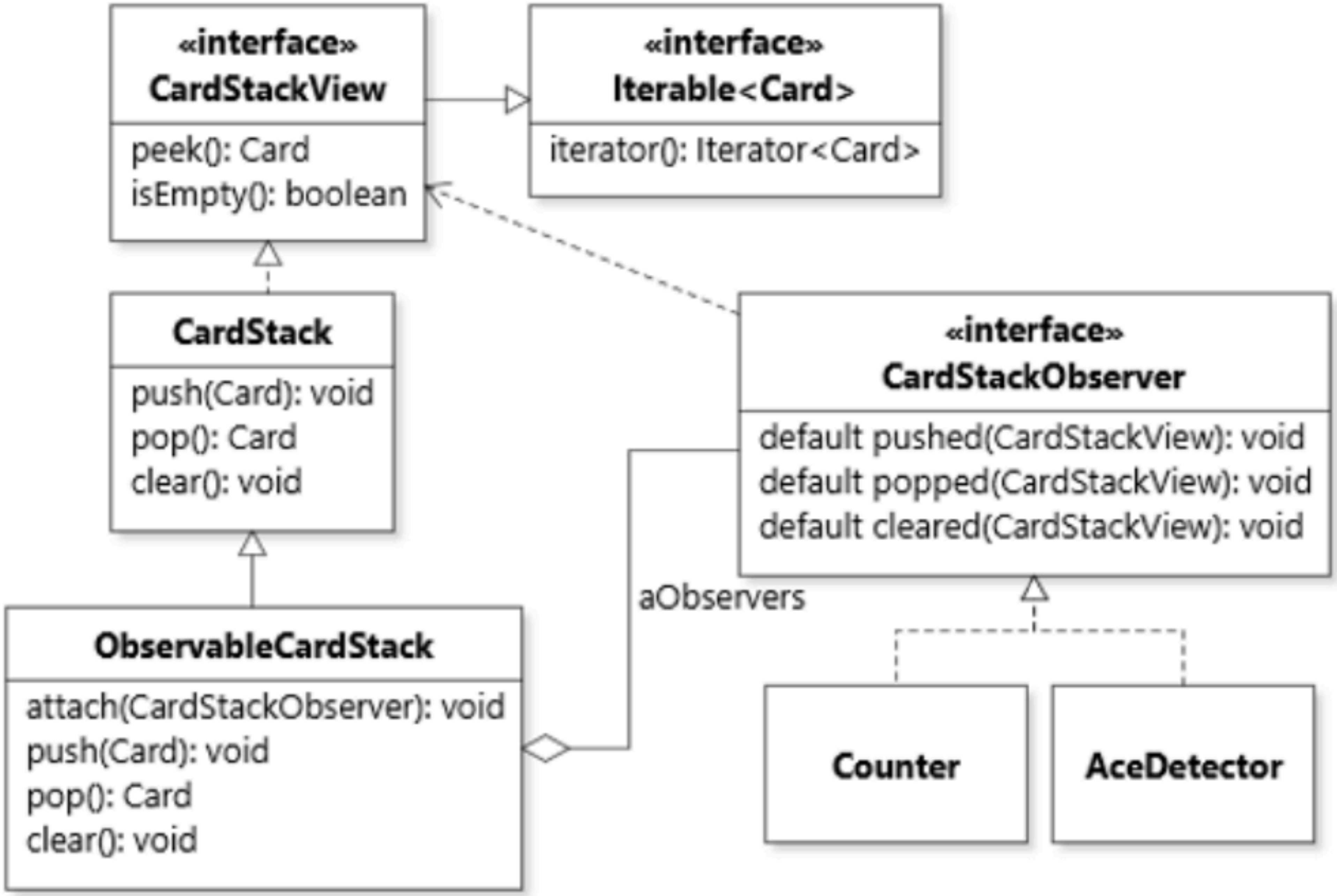


Design with inheritance

```java
public class ObservableCardStack extends CardStack {
  ...

  public Card pop() {
    Card popped = super.pop();
    for (CardStackObserver observer : aObservers) {
      observer.popped(popped);
    }
    return popped;
  }
}
```

Design with pull data flow

CardStackView: like CardStack, but with only getter methods.



Design with pull data flow

```
public class ObservableCardStack extends CardStack {
  ...

  public Card pop() {
    Card popped = super.pop();
    for (CardStackObserver observer : aObservers) {
      observer.popped(this);
    }
    return popped;
  }
}
```

Pass self to callbacks.

Design with pull data flow

```java
public class AceDetector implements CardStackObserver {
  public void pushed(CardStackView pView) {
    if (pView.peek().getRank() == Rank.ACE) {
      System.out.println("Ace detected!");
    }
  }
  public void popped(Card pCard) {}
  public void cleared() {}
}
```

Call methods on ModelView parameter to get state.

Design with pull data flow

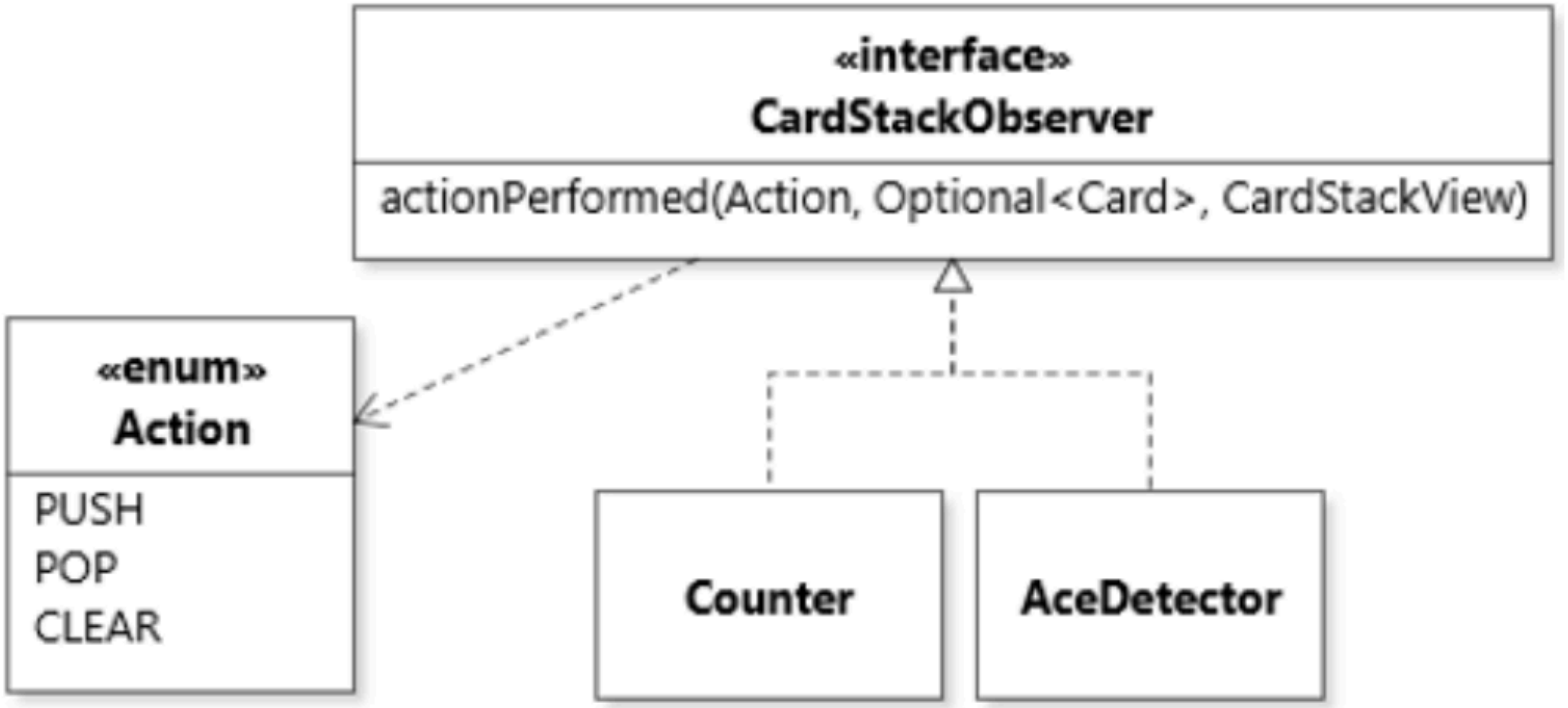
```
public class Counter implements CardStackObserver {
  private static int size(CardStackView pView) {
    int size = 0;
    for (Card card : pView) {
      size++;
    }
    return size;
  }
  public void popped(CardStackView pView) {
    System.out.println("POP Counter=" + size(pView));
    if (pView.isEmpty()) {
      System.out.println("Last card popped!");
    }
  }
  ...
}
```

No longer need to maintain own counter; can just check size of cards directly.

Single callback, push+pull

- Suppose we only have a single callback actionPerformed. It will take as parameters:
  - an object of type Action, an enum which represents the different possible actions,
  - and an Optional<Card> and the CardStackView (thus supporting both push and pull data flows).

Single callback, push+pull



Single callback, push+pull

- Each observer will implement actionPerformed and check if the Action is one that they should respond to.

- E.g., for AceDetector:

```
public void actionPerformed(Action pAction, Optional<Card> pCard,
                            CardStackView pView) {
  if (pAction == Action.PUSH && pView.peek().getRank() == Rank.ACE) {
    System.out.println("Ace detected!");
  }
}
```

Single callback, push+pull

- For Counter:

```
public void actionPerformed(Action pAction, Optional<Card> pCard,
                            CardStackView pView) {
  switch(pAction) {
    case PUSH:
      System.out.println("PUSH Counter=" + size(pView));
      break;
    case POP:
      ...
    case CLEAR:
      ...
  }
}
```

(Switch statement anti-pattern.)

Example on project server

- Keybinds are callback methods.

GUIs

GUI

- GUI: Graphical user interface.

- Makes heavy use of Observer pattern.

- Split into two parts:

  - **framework code**: consisting of a component library (reusable types and interfaces that provide typical GUI functionality like buttons, windows, etc.) and application skeleton (low-level aspects of GUIs such as monitoring events).

  - **application code**: using the framework code, a GUI for a particular application is built.

GUI control flow

- Unlike a regular script which runs (starting in the main method in Java, e.g.), starting a GUI-based app involves launching a framework, which starts an **event loop** that continually monitors for input.

- Once input is detected, the application code is executed in response to a call by the framework.

- Inversion of control: application code does not tell the framework what to do; instead, it waits for the framework to call it.

LuckyNumber

```java
// Application: a class in a GUI framework
public class LuckyNumber extends Application {
  public static void main(String[] pArgs) {
    // launches GUI framework
    launch(pArgs);
  }

  @Override
  public void start(Stage pPrimaryStage) {
    // create windows, buttons, etc.
  }
}
```

HelloWorld in Python Tkinter

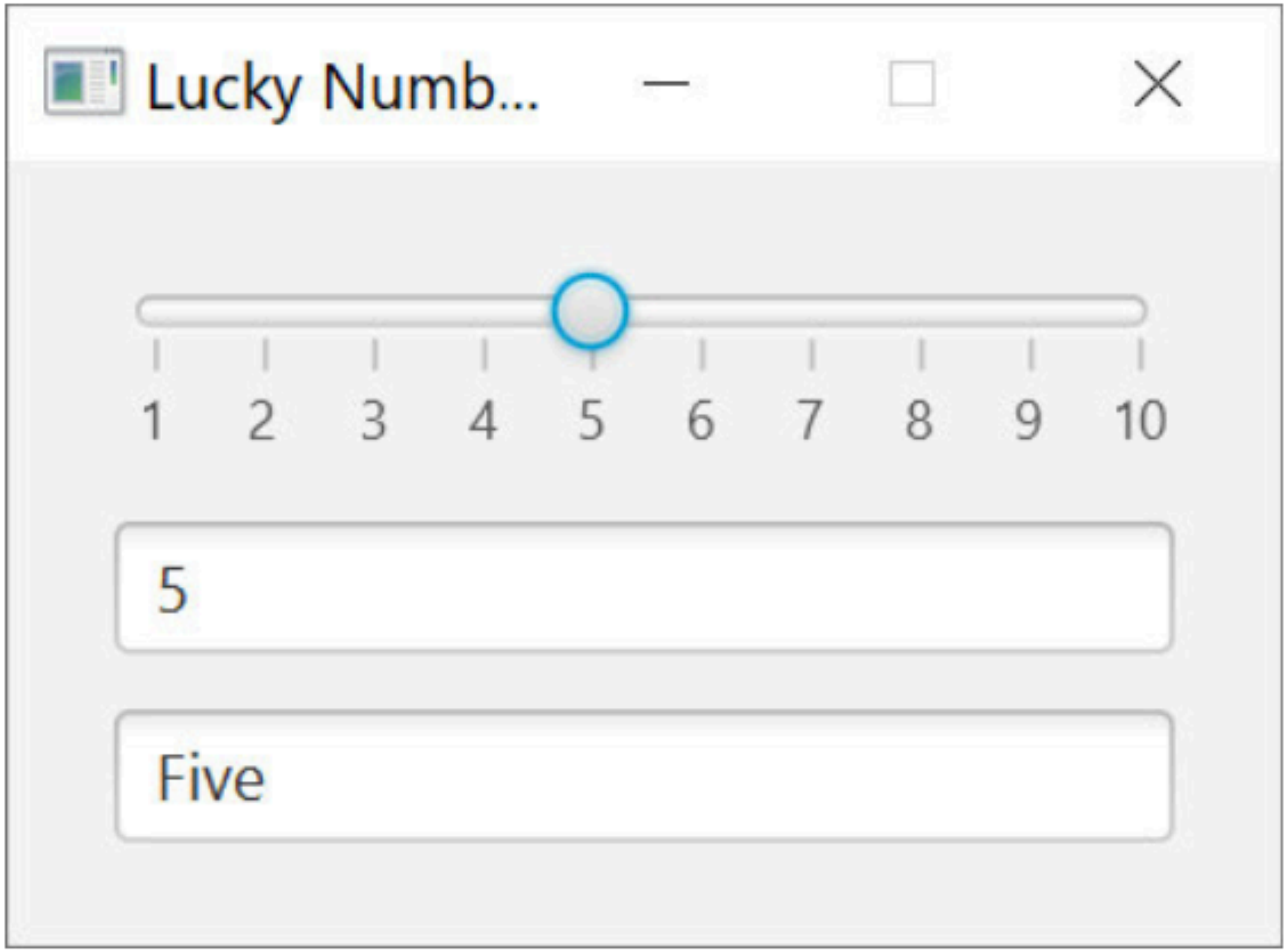
```
class HelloWorldApp:
  def __init__(self, root):
    self.root = root
    root.title("Hello World App")

    self.label = tk.Label(root, text="Hello, World!")
    self.label.pack()

    self.close_button = tk.Button(root, text="Close",
                                  command=root.quit)
    self.close_button.pack()

if __name__ == "__main__":
    root = tk.Tk()
    app = HelloWorldApp(root) # create windows, buttons, etc.
    root.mainloop() # start GUI framework
```

Application code

- Application code can be split into two parts:
  - the **component graph**: the actual UI - buttons, etc. Organized as a tree (buttons go on windows, etc.).
    - Heavy use of the Composite and Decorator patterns.
  - the **event handling code**: the code to execute when a button is clicked, when the mouse is moved around, etc. ("events").
    - Application of the Observer pattern.

GUI component graphs

From the user's perspective.



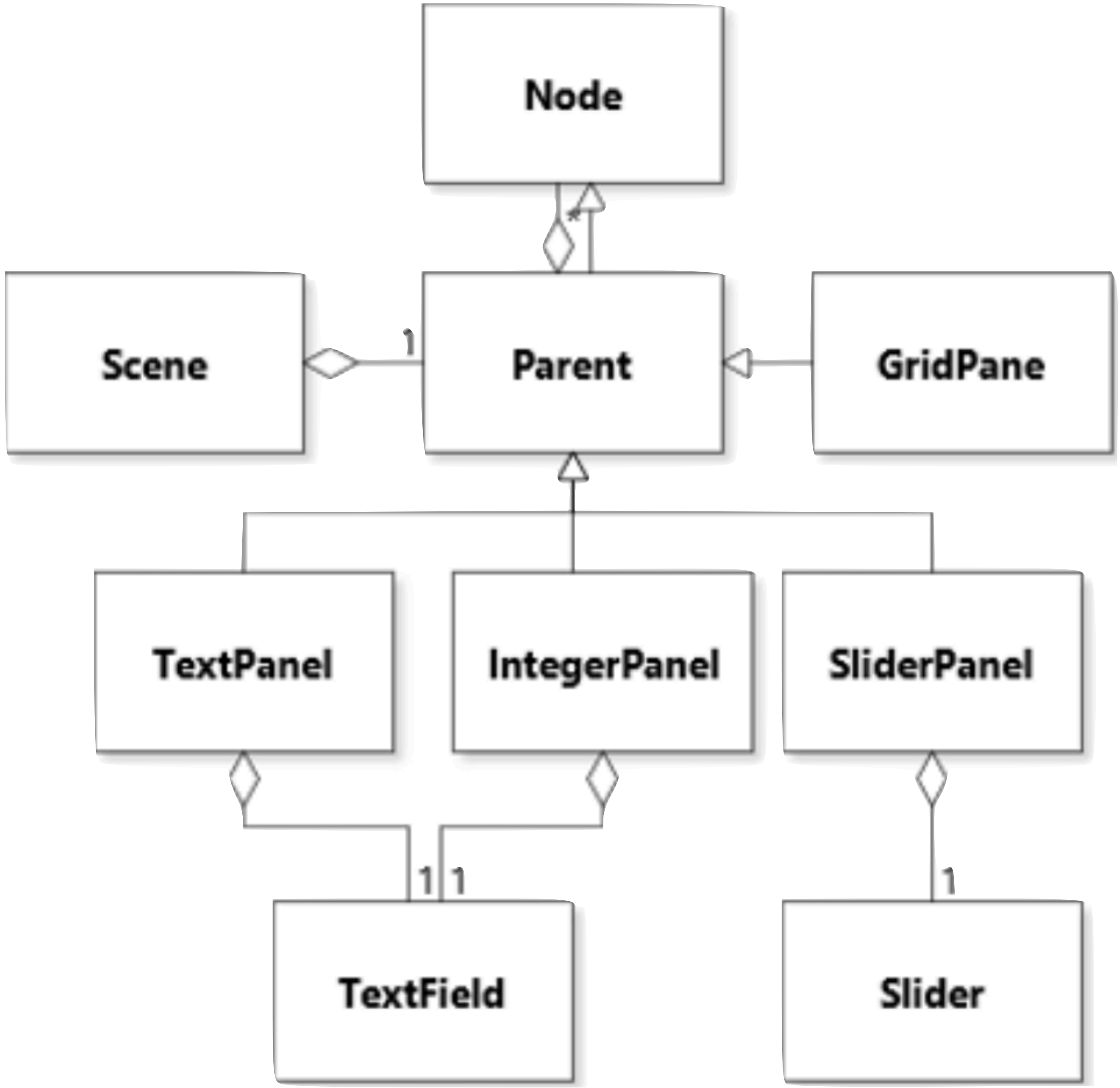
5

Five

GUI component graphs

From the source code perspective (a class diagram.)

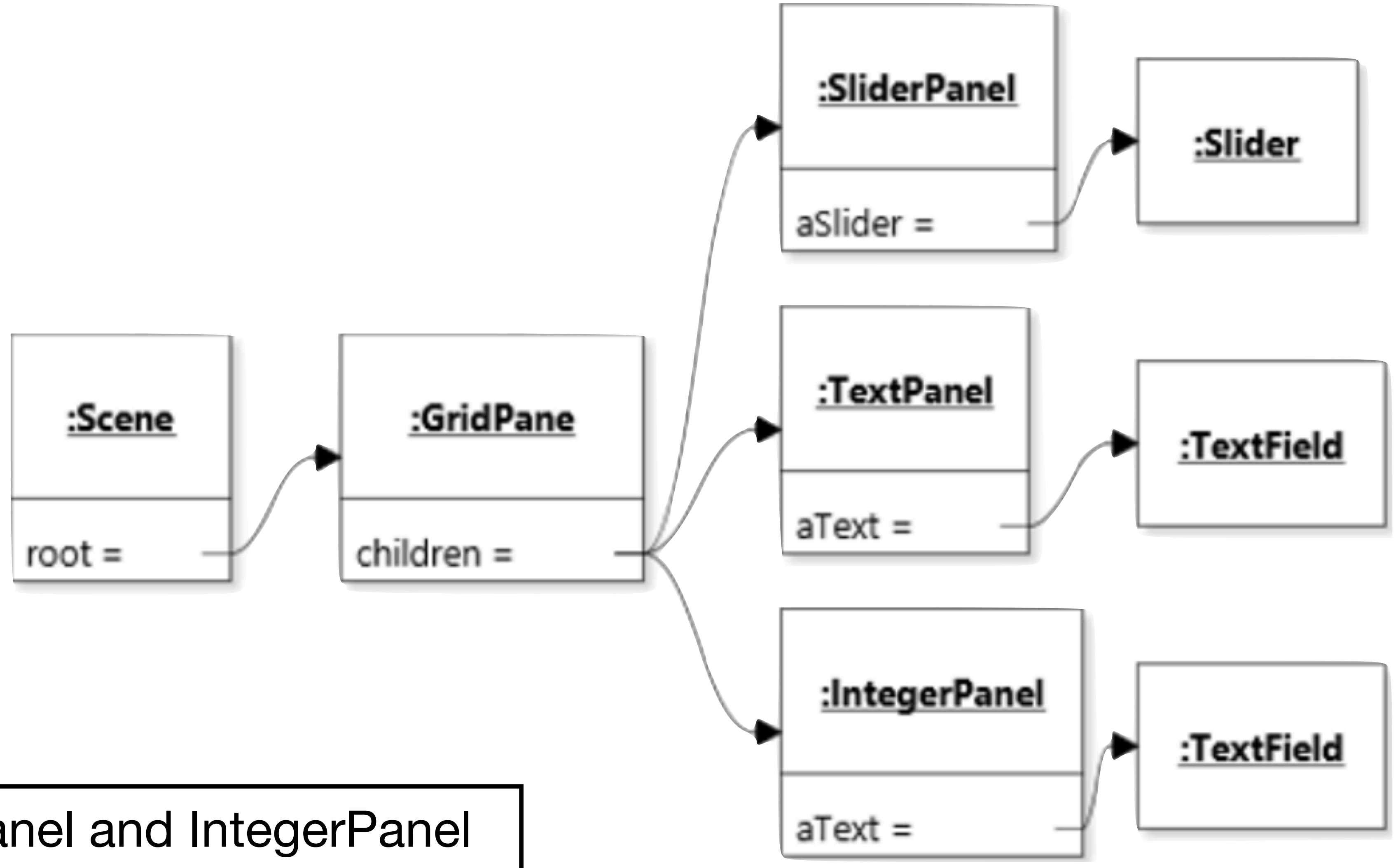
Some general class (e.g., Scene) will aggregate a parent node.



The parent node will aggregate all the UI element objects.

GUI component graphs

Runtime perspective (object diagram).



The TextPanel and IntegerPanel have their own TextField.

Defining the object graph

```
public class LuckyNumber extends Application {
  public void start(Stage pStage) {
    Model model = new Model(); // observer
    GridPane root = new GridPane();

    // Panel classes defined earlier.
    root.add(new SliderPanel(model), 0, 0, 1, 1);
    root.add(new IntegerPanel(model), 0, 1, 1, 1);
    root.add(new TextPanel(model), 0, 2, 1, 1);
    pStage.setScene(new Scene(root));
    pStage.show();
  }
}
```

IntegerPanel

```java
public class IntegerPanel extends Parent implements Observer {
  private TextField aText = new TextField();
  private Model aModel;

  public IntegerPanel(Model pModel) {
    aModel = pModel;

    // register as an observer of the model
    aModel.addObserver(this);
    aText.setText(new Integer(aModel.getNumber()).toString());

    // add the text field to the component graph
    getChildren().add(aText);
    ...
  }

  // will be called when notified by the model that number has changed
  public void numberChanged(int pNumber) {
    aText.setText(new Integer(pNumber).toString());
  }
}
```

IntegerPanel

```python
class IntegerPanel(tk.Frame):
  def __init__(self, parent, model):
    super().__init__(parent)
    self.__model = model
    self.__model.add_observer(self)

    # passing self automatically adds to the current frame
    self.__aText = tk.Entry(self)
    self.__aText.insert(0, str(self.__model.get_number()))
    self.__aText.pack(padx=10, pady=10)

  def numberChanged(self, pNumber):
    self.__aText.delete(0, tk.END)
    self.__aText.insert(0, str(pNumber))
```

Defining the object graph

```
class LuckyNumber:
  def start():
    root = tk.Tk()
    root.title("Lucky Number")

    model = Model()

    slider_panel = SliderPanel(root, model)
    slider_panel.pack(fill='x')

    integer_panel = IntegerPanel(root, model)
    integer_panel.pack(fill='x')

    text_panel = TextPanel(root, model)
    text_panel.pack(fill='x')

    root.mainloop()
```

References

- Robillard ch. 8.4-8.6, p.208-224
  - Exercises #6-10: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter8.md

Coming up

- Next lecture:
    - Visitor pattern