COMP 303

Lecture 19

Inversion of control III

Winter 2025

slides by Jonathan Campbell

Announcements

- Next team survey to be posted soon, due this Friday
- Tutorial / group coding
- Midterm grades
- Final exam questions
- Schedule of remaining lectures / final demo / awards
- Special trees

writing lyrics to all too well (10min ver.) (Taylor's version)

6. (1 point) What kind of question would you like to see on the final exam?

1

Draw your best garfield (I will win)

6. (1 point) What kind of question would you like to see on the final exam?

Q6 1

Designing a system with multiple interconnected design patterns, implenting part of each pattern, instead of just one isolation like here.

6. (1 point) What kind of question would you like to see on the final exam?

Q6 1

Punish us with multiple design patterns at once ☺

Inversion of control

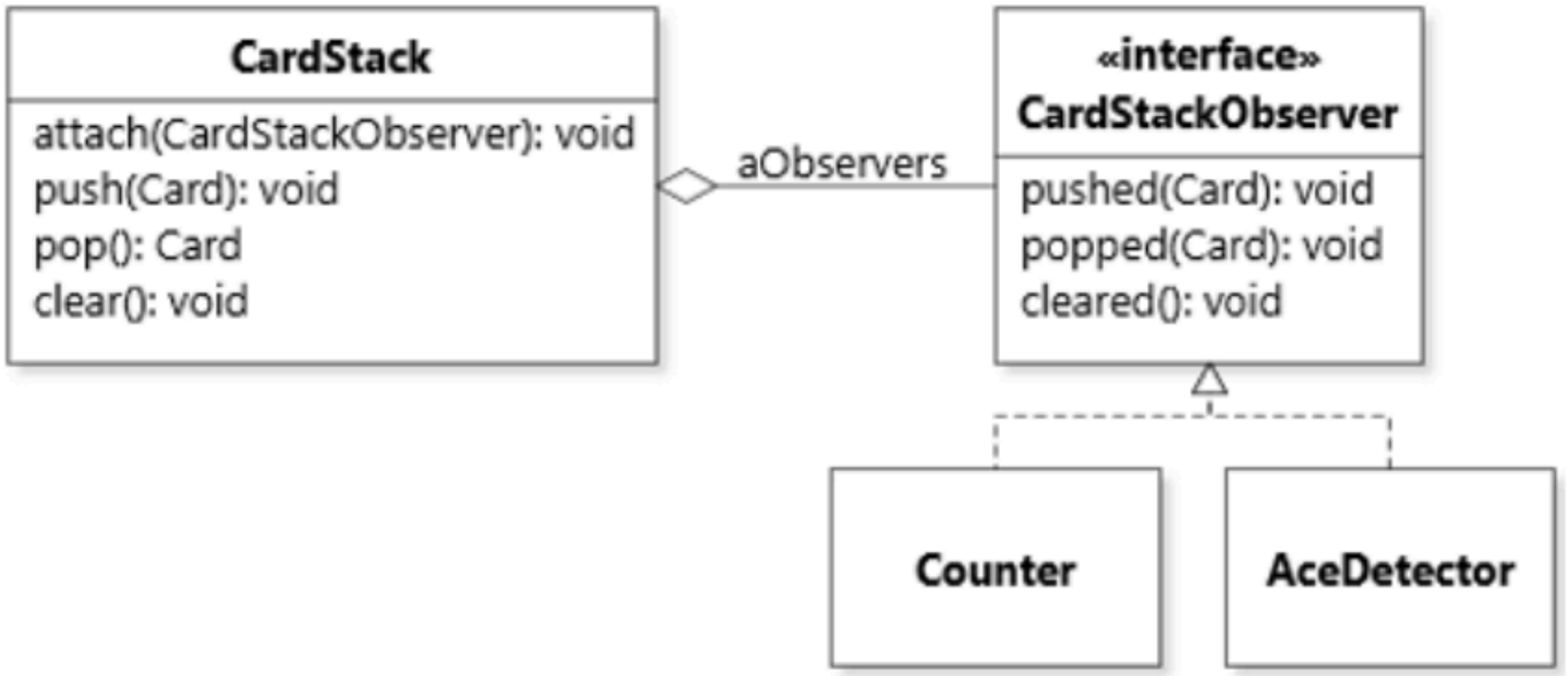
- Observer pattern
  - GUI
  - **Event handling (today)**
- **Visitor pattern (today)**

Recap

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).

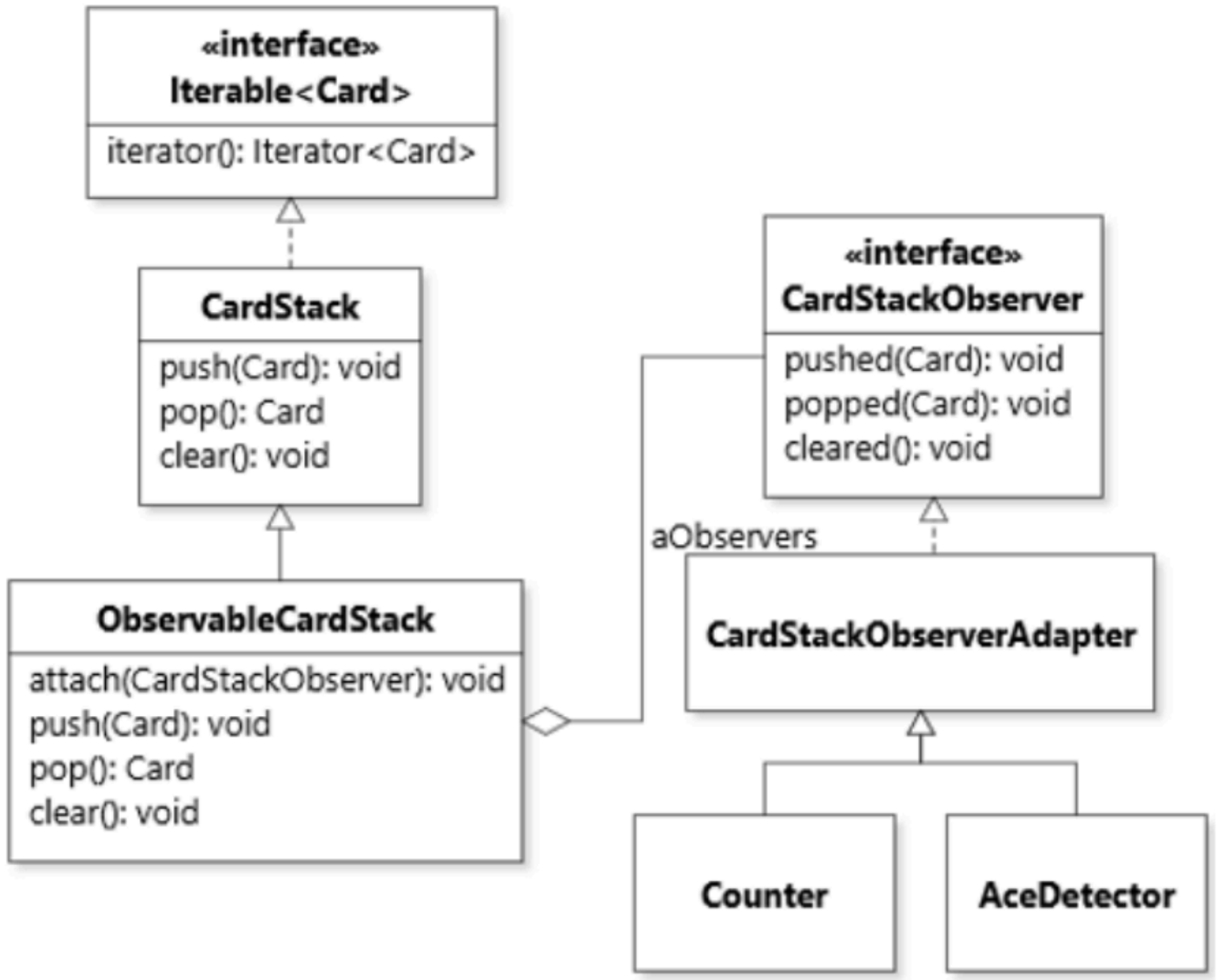
Example: observable CardStack



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

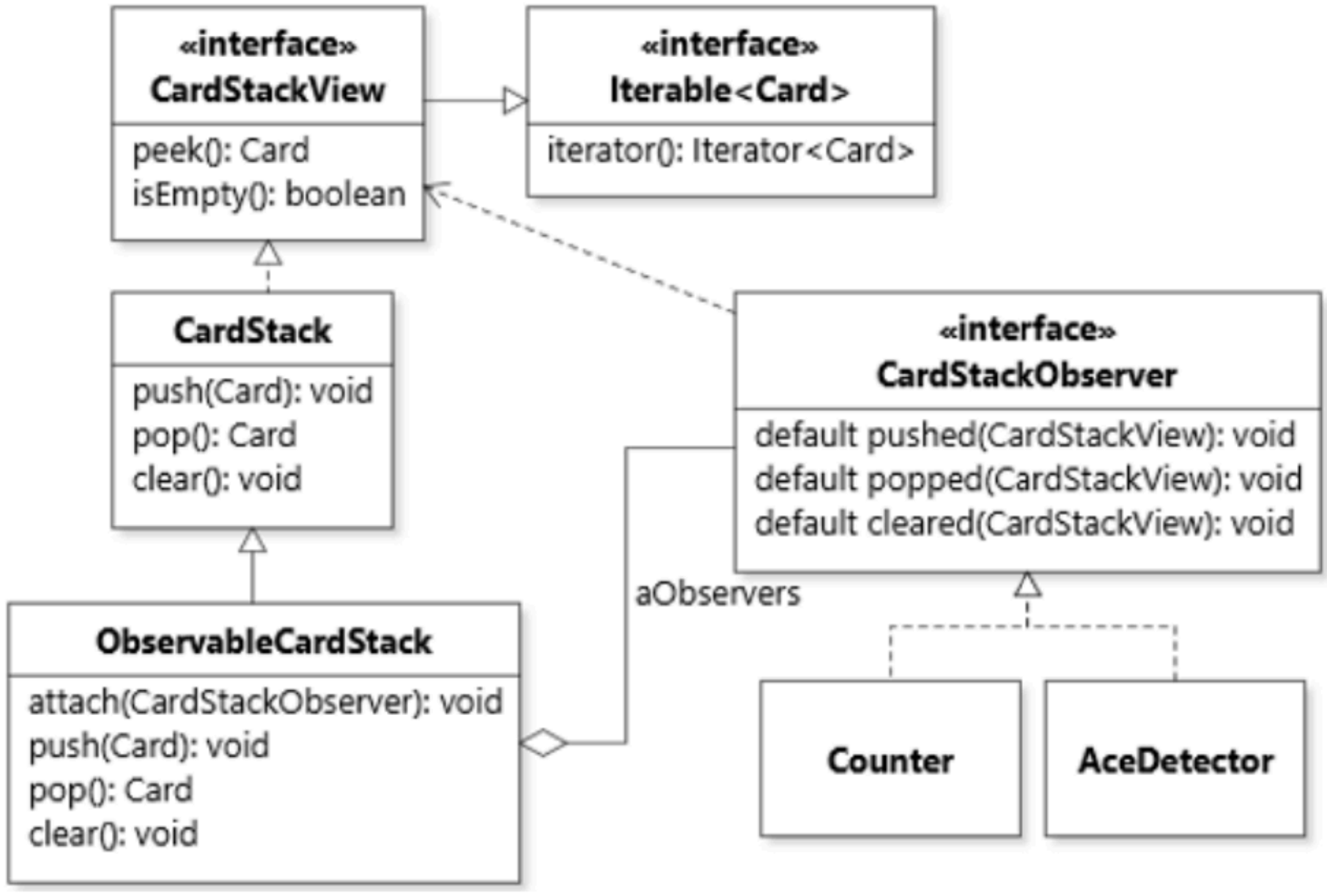
Ace Detector: detects whether an ace is added to the stack.

Design with inheritance

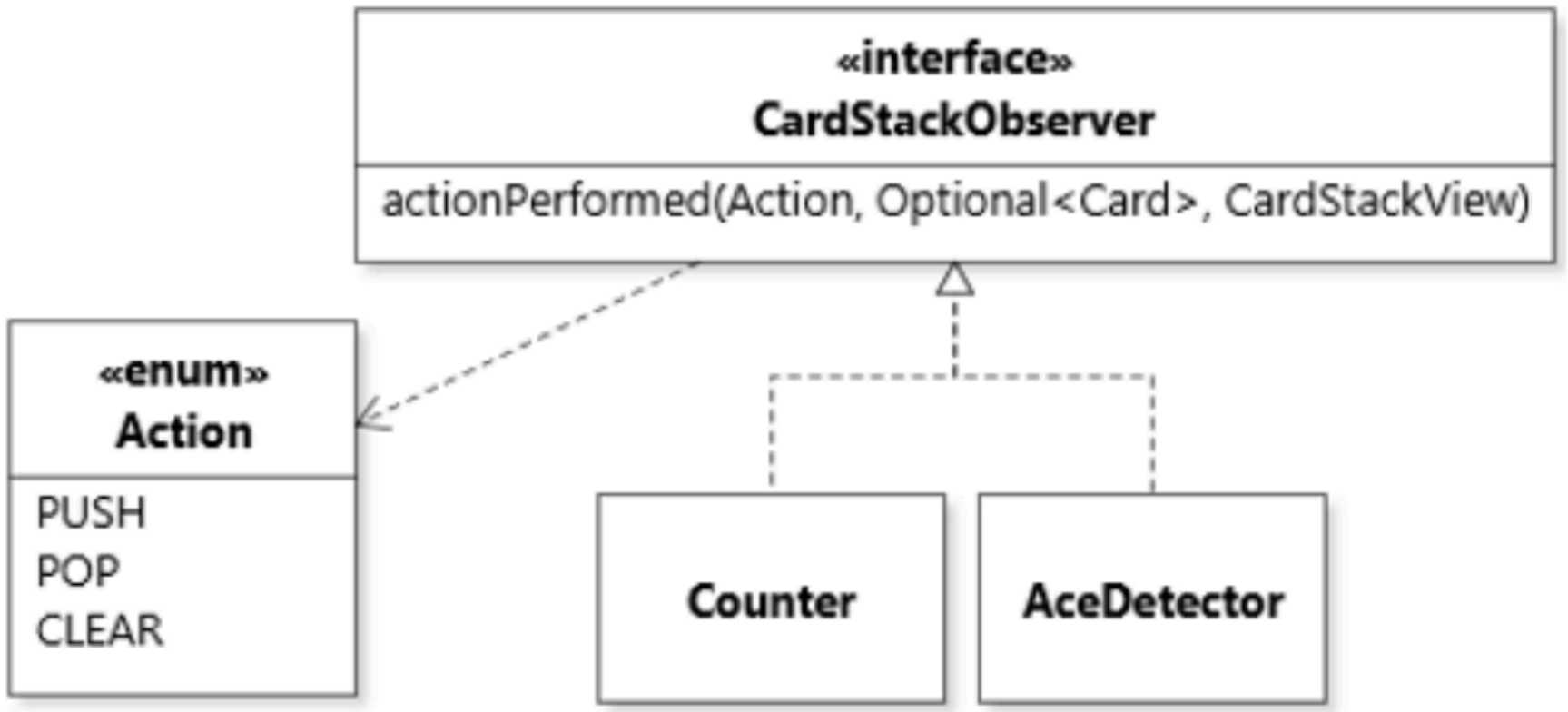


Design with pull data flow

CardStackView: like CardStack, but with only getter methods.



Single callback, push+pull



Example on project server

- Keybinds are callback methods.

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.

Application code

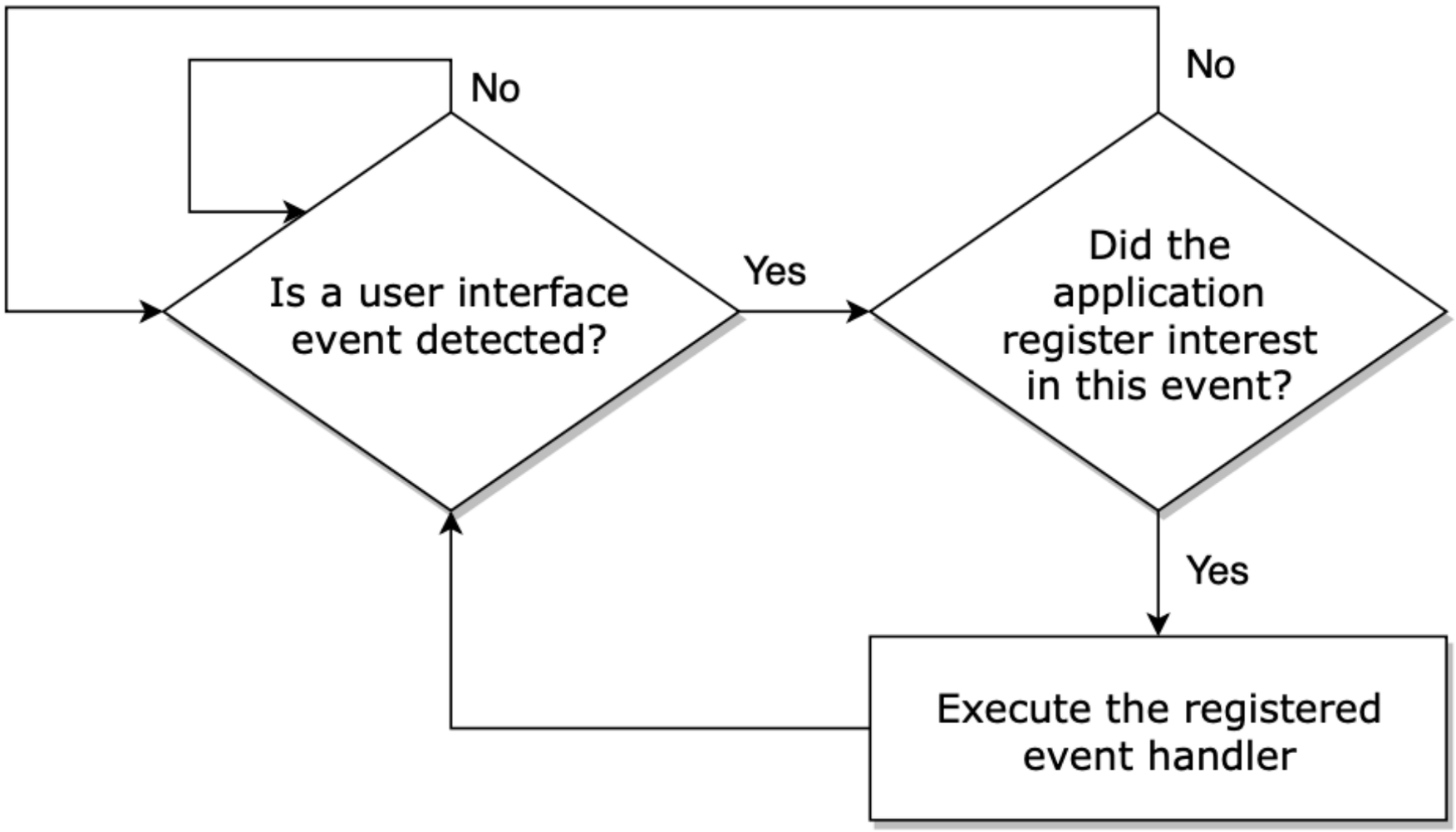
- Application code can be split into two parts:
    - the **component graph**: the actual UI - buttons, etc. Organized as a tree (buttons go on windows, etc.).
        - Heavy use of the Composite and Decorator patterns.
    - the **event handling code**: the code to execute when a button is clicked,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

Event handling

Event handling

- Once the framework is launched, an event loop begins, monitoring input events and checking whether they map to events that can be observed by application code.

Event handling



Events

- Events are typically defined by the component library.
  - E.g., TextField defines an event that occurs when the user types the [enter] key.

- After we instantiate a component, we must create and register an **event handler**: the code that will execute when this event occurs.

Event handlers

```
public class IntegerPanel extends Parent implements Observer {
  private TextField aText = new TextField();
  private Model aModel;
  public IntegerPanel(Model pModel) {
    aModel = pModel;
    aModel.addObserver(this);
    aText.setText(new Integer(aModel.getNumber()).toString());
    getChildren().add(aText);
    aText.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent pEvent) {
        int number;
        try {
           number = Integer.parseInt(aText.getText());
        } catch(NumberFormatException pException ) {
           number = 1;
        }
        aModel.setNumber(number);
      }
    });
  }
}
```

setOnAction: registering a new event handler

defining an anonymous class, subtype of EventHandler

handle method will be called when the event occurs.

Event handlers

```
class IntegerPanel(tk.Frame):
  def __init__(self, parent, model):
    super().__init__(parent)
    self.__aModel = model
    self.__aModel.add_observer(self)
    self.__aText = tk.Entry(self)
    self.__aText.insert(0, str(self.__aModel.get_number()))
    self.__aText.pack()
    self.__aText.bind("<Return>", self.on_enter)

def on_enter(self, event):
  try:
    number = int(self.__aText.get())
  except ValueError:
    number = 1
  self.__aModel.set_number(number)
```

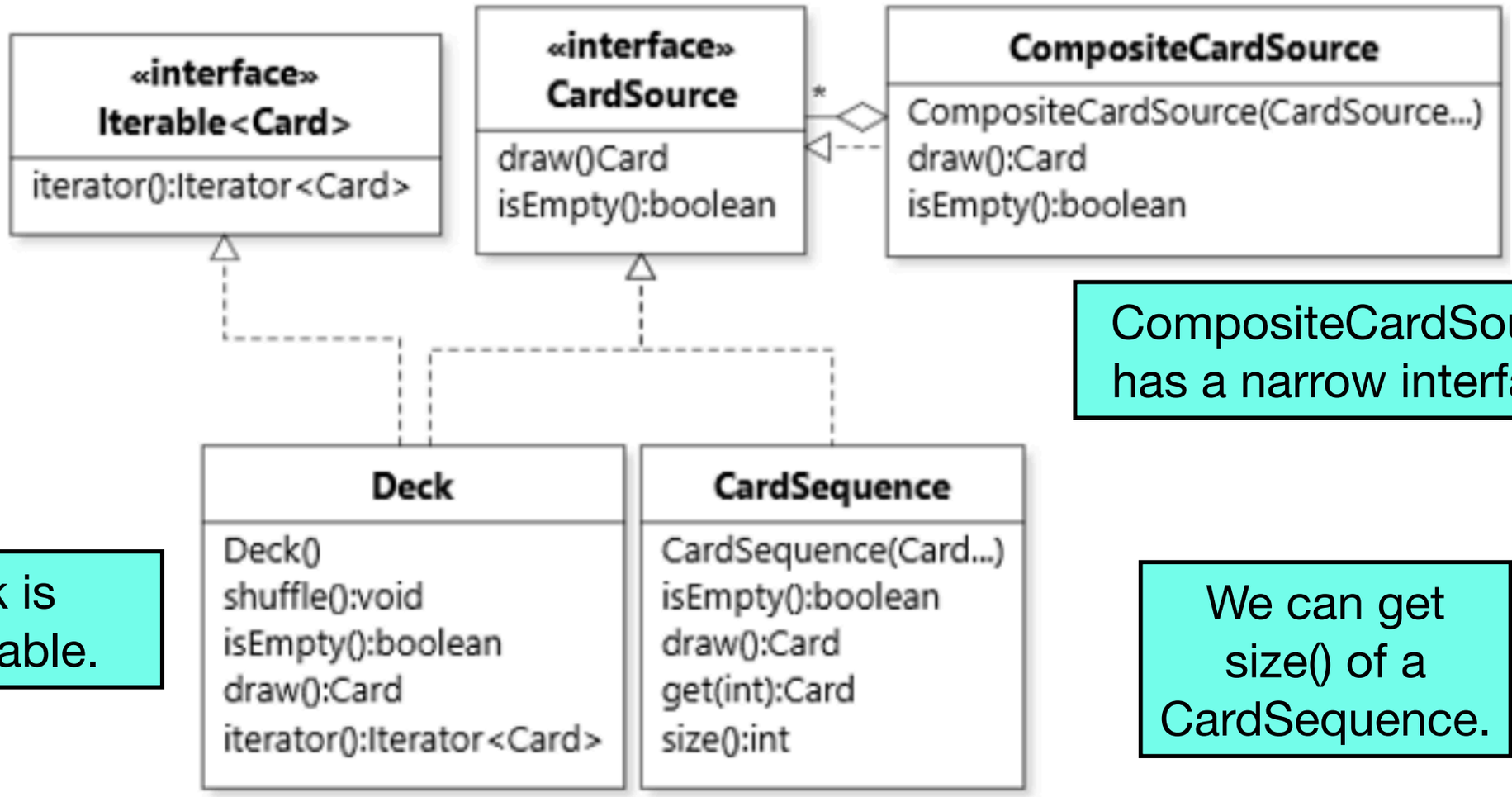
bind: registering a new event handler

defining a method taking an event parameter

method will be called when the event occurs.

Visitor pattern

CardSource



CompositeCardSource has a narrow interface.

Deck is shuffleable.

We can get size() of a CardSequence.

Three different types of CardSources, each having slightly different methods.

Adding methods to CardSource

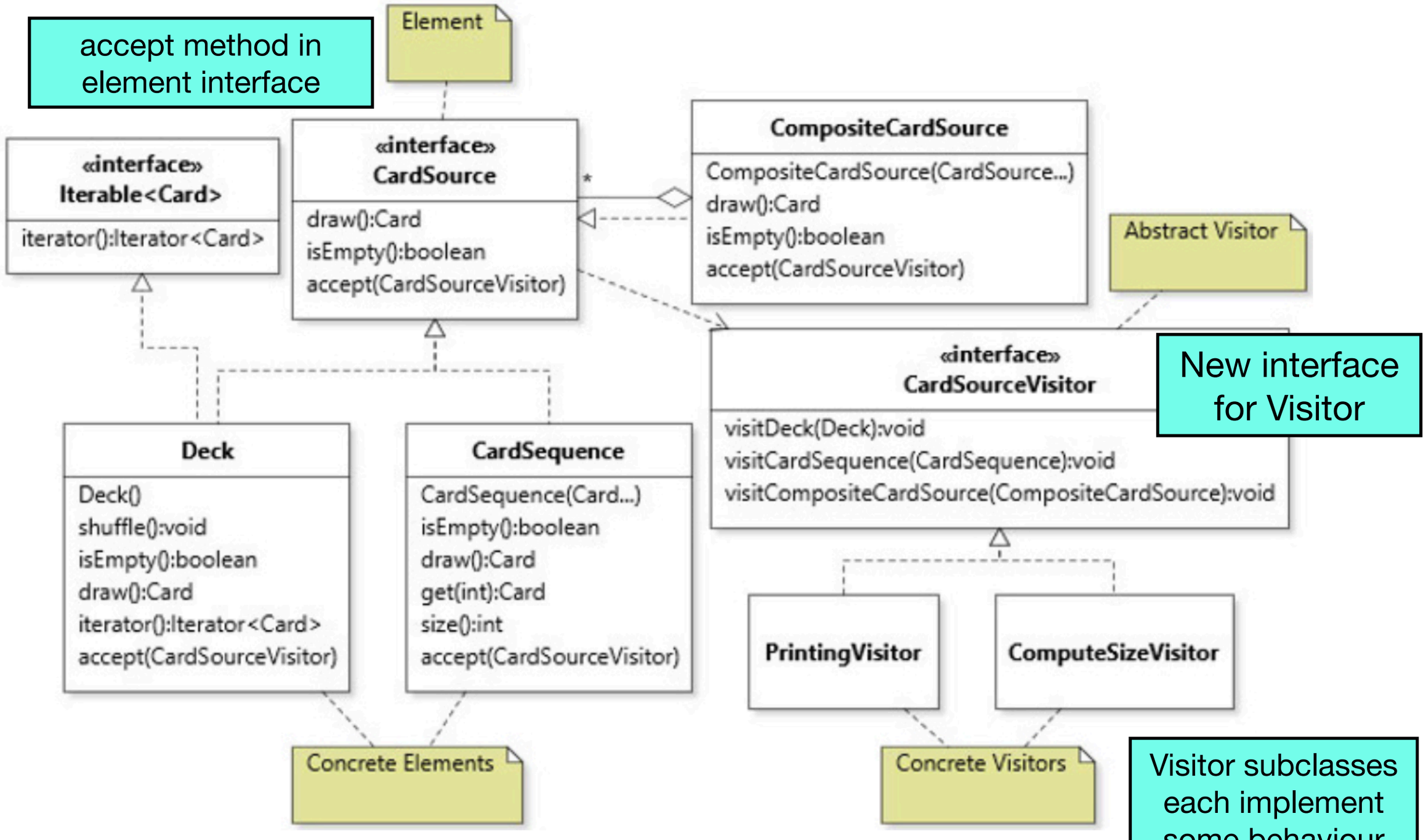
- What if we want all CardSources to have more methods, such as print, size, remove(Card) and contains(Card)?

- We could add all these methods to the CardSource interface. Then all subclasses would need to implement them. But:

  - The interface of CardSource would get much larger.

  - We may not use all the methods in all subclasses, which would violate the ISP. If we just think we may use it in future, then it is a case of SPECULATIVE GENERALITY.

Solution: VISITOR pattern



VISITOR pattern

- Solution: **VISITOR** pattern.
- Idea: Define functionality (like `contains(Card)`) in its own class.

- Three parts:
  - Abstract Visitor interface
  - Concrete Visitors (one for each behaviour)
  - accept methods in element subtypes

Abstract visitor

```
public interface CardSourceVisitor {
   void visitCompositeCardSource(CompositeCardSource pSource);
   void visitDeck(Deck pDeck);
   void visitCardSequence(CardSequence pCardSequence);
   // ...
}
```

A visit method for every different element subclass.

Concrete visitor

```
public class PrintingVisitor implements CardSourceVisitor {
  public void visitCompositeCardSource(CompositeCardSource pSource)
  {}

  public void visitDeck(Deck pDeck) {
    for (Card card : pDeck) {
      System.out.println(card);
    }
  }

  public void visitCardSequence(CardSequence pCardSequence) {
    for (int i = 0; i < pCardSequence.size(); i++) {
      System.out.println(pCardSequence.get(i));
    }
  }
}
```

A concrete visitor for every different behaviour.

Visitors

- In a classic design, code to implement behaviours like printing, getting size, etc., would be scattered throughout the three CardSource classes.

- Here, all the code for a specific behaviour is located in a single class.

  - An organization of code in terms of functionality instead of data.

Integrating the visitors

- We've defined a class that encapsulates a well-defined operation. But we still have to somehow integrate it with our actual element interface (e.g., CardSource) and subclasses.

- We will define a method accept, that takes an object of the abstract visitor type.

Integrating the visitors

```
public interface CardSource {
   Card draw();
   boolean isEmpty();
   void accept(CardSourceVisitor pVisitor);
}
```

Integrating the visitors

- The accept method will simply call the relevant visit method on the visitor.

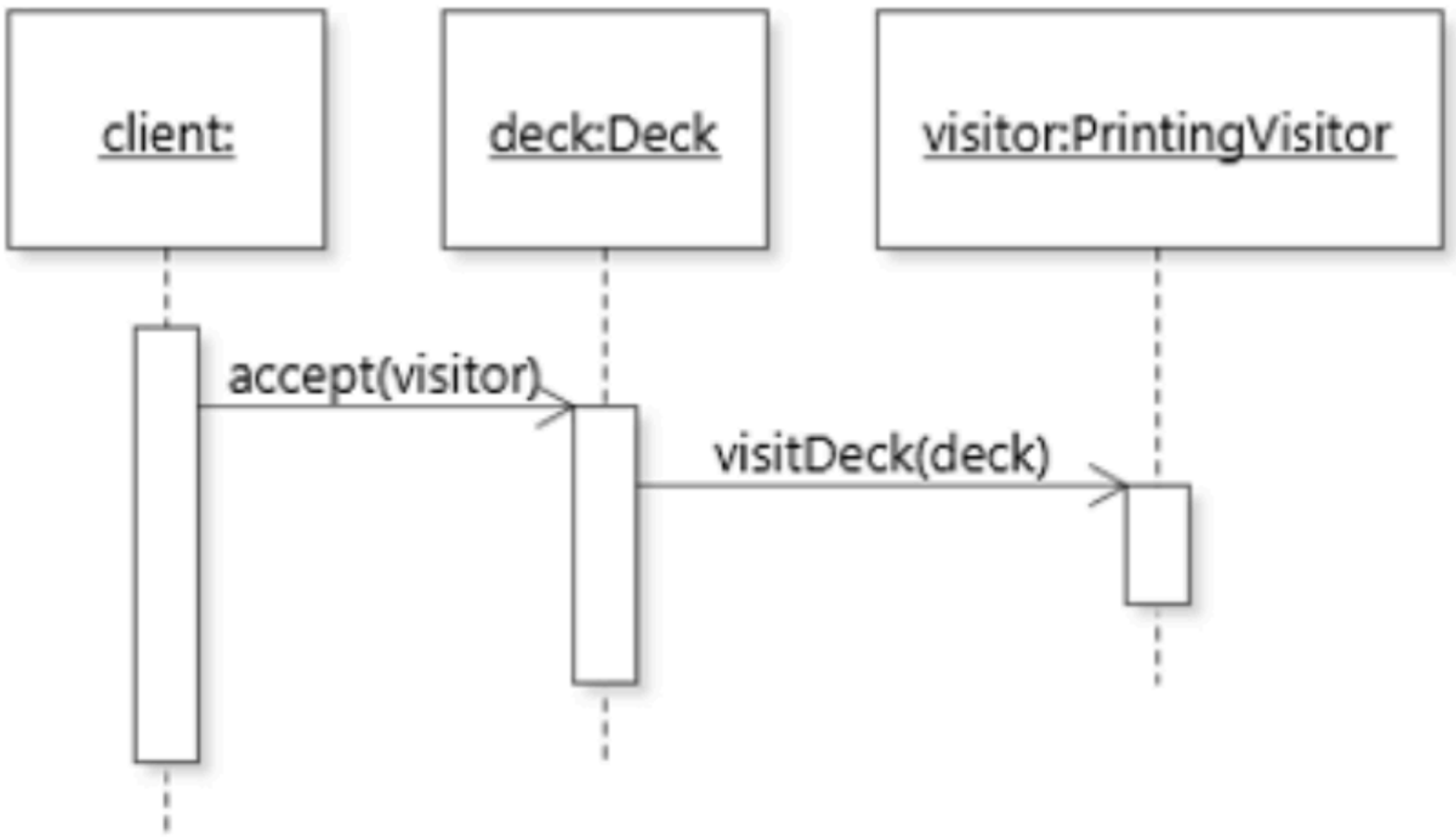
```
public class Deck {
   public void accept(CardSourceVisitor pVisitor) {
      pVisitor.visitDeck(this);
   }
}
public class CardSequence {
   public void accept(CardSourceVisitor pVisitor) {
      pVisitor.visitCardSequence(this);
   }
}
```

Invoking the behaviour

```
// in client code
Deck deck = new Deck();

PrintingVisitor visitor = new PrintingVisitor();
deck.accept(visitor);
```

Invoking the behaviour



We can think of visitDeck as a sort of a callback method -- the elements call the visitors at the appropriate time.

CompositeCardSource

- We have to do some extra work to make the Visitor pattern work with CompositeCardSource.

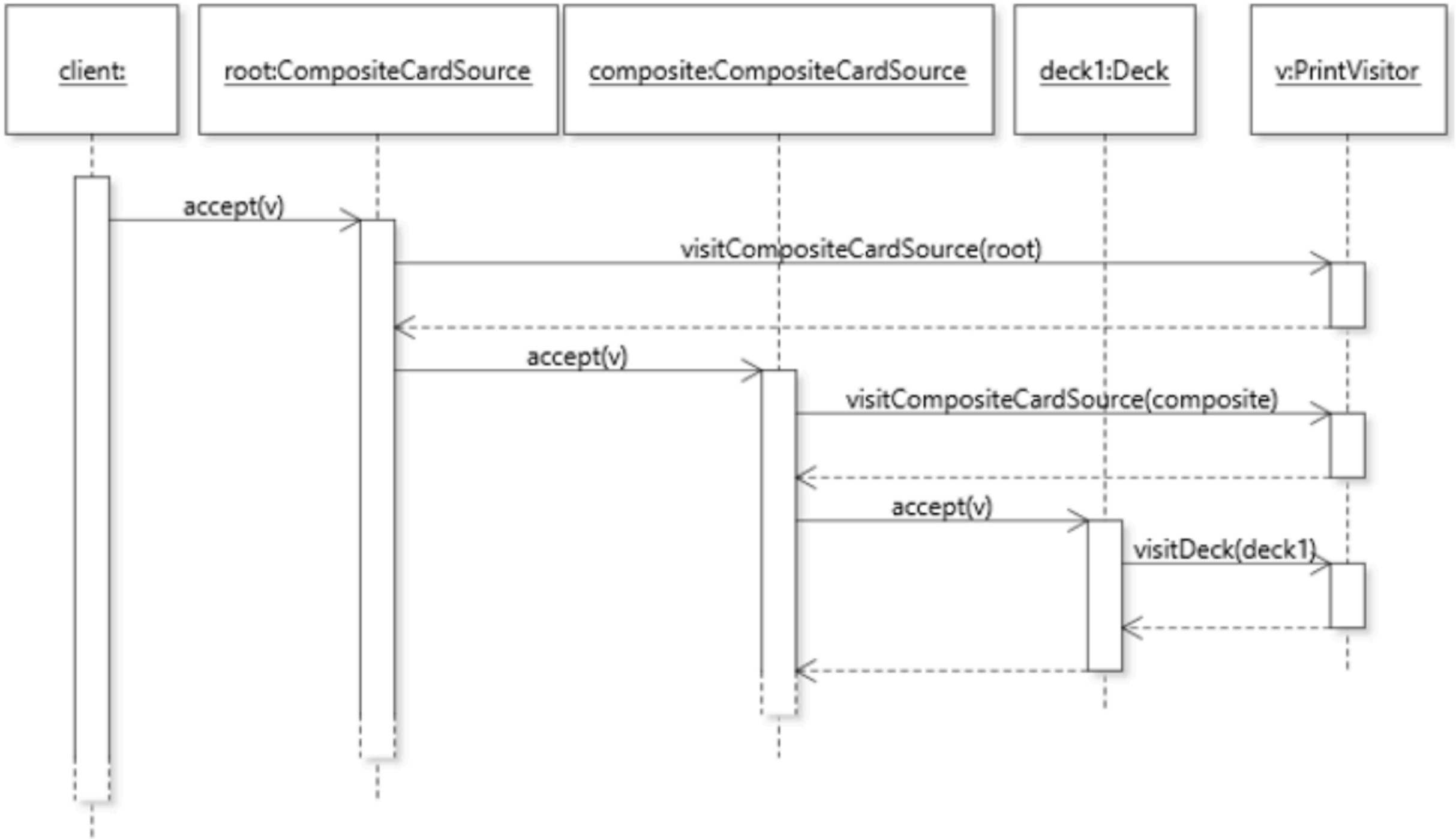
- If we apply an operation (like print, size, etc.) to a CompositeCardSource, we really want the operation to be applied to all of its aggregated elements.

CompositeCardSource

```
public class CompositeCardSource implements CardSource {
  private final List<CardSource> aElements;

  public void accept(CardSourceVisitor pVisitor) {
    pVisitor.visitCompositeCardSource(this);
    for (CardSource source : aElements) {
      source.accept(pVisitor);
    }
  }
}
```

CompositeCardSource



CompositeCardSource

- We could have instead placed this same code inside the visitCompositeCardSource method, instead of accept:

```
public class PrintVisitor implements CardSourceVisitor {
  public void visitCompositeCardSource(CompositeCardSource pCompCardSrc) {
    for (CardSource source : pCompositeCardSource) {
      source.accept(this);
    }
  }
  ...
}
```

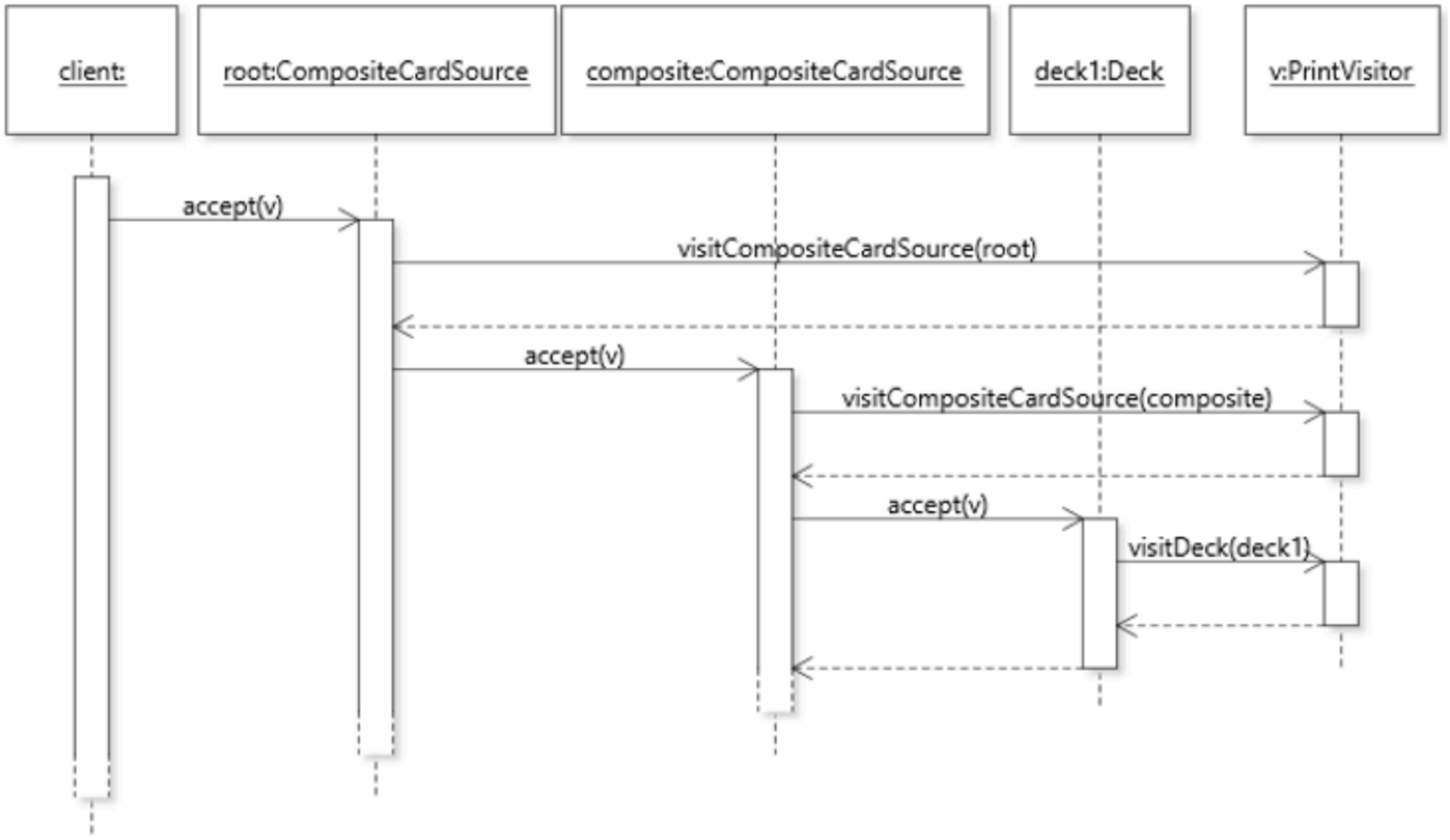
- (Since this class can't access the private aElements field of the composite, we'd have to make the composite iterable.)

CompositeCardSource

```
public class CompositeCardSource implements CardSource,
                                             Iterable<CardSource> {
  private final List<CardSource> aElements;

  public Iterator<CardSource> iterator() {
    return aElements.iterator();
  }
  ...
}
```

CompositeCardSource



CompositeCardSource

- The advantage to placing the traversal code in the visit method is that, depending on the behaviour, we can change the order of traversal, if we wanted.

- The downside is that we have to make the composite class iterable, possibly making its encapsulation weaker.

  - Another downside is that the traversal code would be repeated in every concrete visitor (DUPLICATED CODE).

Visitor with inheritance

```
public abstract class AbstractCardSourceVisitor implements CardSourceVisitor {
  public void visitCompositeCardSource(CompositeCardSource pCompositeCardSrc) {
    for (CardSource source : pCompositeCardSource) {
      source.accept(this);
    }
  }

  public void visitDeck(Deck pDeck) {}
  public void visitCardSequence(CardSequence pCardSequence) {}
}
```

Avoids duplicated code problem.

Visitor with data flow

- All of our visit methods have been void so far.
- But we may want to return information from them. E.g., a size visitor should return the size.
  - But, all visit methods must return void, or else they wouldn't implement the abstract visitor interface.
  - Instead, we will store the computed data into the visitor object.

Visitor with data flow

```
public class CountingVisitor extends AbstractCardSourceVisitor {
  private int aCount = 0;

  public void visitDeck(Deck pDeck) {
    for (Card card : pDeck) {
      aCount++;
    }
  }

  public void visitCardSequence(CardSequence pCardSequence) {
    aCount += pCardSequence.size();
  }

  public int getCount() {
    return aCount;
  }
}
```

Visitor with data flow

```
// in client code
CountingVisitor visitor = new CountingVisitor();
root.accept(visitor);
int result = visitor.getCount();
```

References

- Robillard ch. 8.7-8.8, p.224-242

  - Exercises #11-13: https://github.com/prmr/DesignBook/blob/master/exercises/e-chapter8.md

Coming up

- Next lecture:
  - New topic!