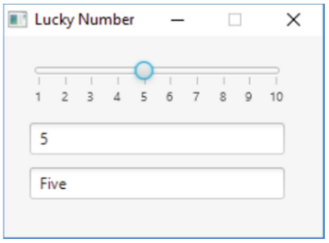
Chapter 8
Inversion of Control

This chapter covers:

Concepts and Principles: Inversion of control, Model–View–Controller (MVC) decomposition, callback method;
Programming Mechanisms: Application framework, event loop, Graphical User Interface (GUI) component graph;
Design Techniques: Adapter inheritance, event handling, GUI design;
Patterns and Antipatterns: PAIRWISE DEPENDENCIES†, OBSERVER, VISITOR.

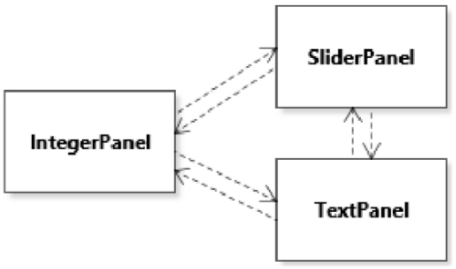
8.1 Motivating Inversion of Control

Synchronization:
keep different objects consistent with each other.



Naïve approach

PAIRWISE DEPENDENCIES†: whenever the user changes the number in a panel, this panel directly contacts all other panels and updates their view of the number.



Drawbacks

- High coupling: Each panel explicitly depends on many other panels.
- Low Extensibility: To add or remove a panel, it is necessary to modify all other panels.

The impact of these issues increases quadratically with the number of panels.

8.2 The Model-View-Controller Decomposition

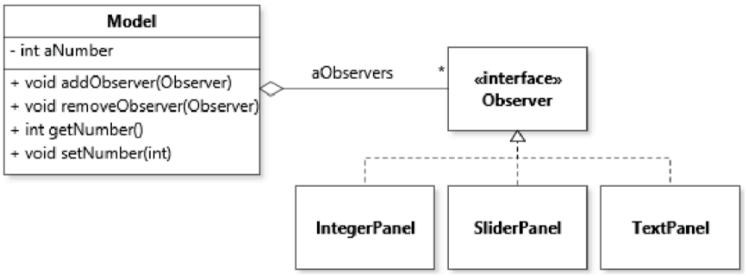
Separate abstractions responsible for *storing data* from abstractions responsible for *viewing data*, from abstractions responsible for *changing data*.

→ Model–View–Controller (MVC)

- The *Model* is the abstraction that keeps the unique copy of the data of interest.
- The *View* is, not surprisingly, the abstraction that represents one view of the data. Generally in a MVC decomposition there can be more than one view of the same model.
- The *Controller* is the abstraction of the functionality necessary to change the data stored in the model.

8.3 The OBSERVER Design Pattern

The central idea of the OBSERVER pattern is to *store data of interest in a specialized object, and to allow other objects to observe this data.* The object that stores the data of interest is called alternatively the subject, model, or observable, and it corresponds to the *Model* abstraction in the Model–View–Controller decomposition.



Linking Model and Observers

Model class also includes an aggregation to an Observer interface, with methods to add and remove Observer instances from its collection. This is also called registering (or deregistering) observers.

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

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

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

Classes that define objects that would be interested in observing the model must then declare to implement the `Observer` interface:

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

Through polymorphism, we thus achieve 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 run time.

Control Flow Between Model and Observers

Whenever there is a change in the model's state worth reporting to observers, the model should let the observers know by cycling through the list of observers and calling a certain method on

them. This method has to be defined on the Observer interface and is usually called a "callback" method because of the inversion of control that it implies. We talk of inversion of control because to find out information from the model the observers do not call a method on the model, they instead "wait" for the model to call them (back).

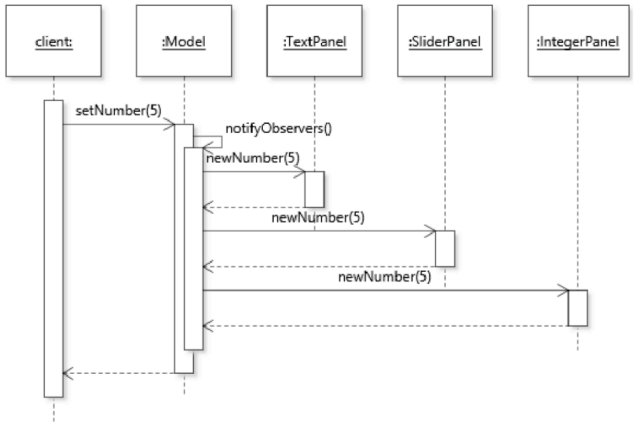
```
public interface Observer
{
  void newNumber(int pNumber);
}

someObserver.newNumber(5);
```

```
public class Model
{
  private void notifyObservers()
  {
    for(Observer observer : aObservers)
    { observer.newNumber(aNumber); }
  }
}
```

The idea of a callback is not to tell observers what to do, but rather to *inform* observers about some change in the model, and let them deal with it as they see fit (through the logic provided in the callback). Once we have a callback defined, within class Model, we can create a helper method, called a notification method that will notify all observers and provide them with the number they should know about.

To ensure that the model dutifully notifies observers whenever a state change occurs, two strategies are possible:

1. A call to the notification method must be inserted in every state-changing method; in this case the method can be declared `private`;



2. Clear documentation has to be provided to direct users of the model class to call the notification method whenever the model should inform observers. In this case the notification methods needs to be non-private.

[This strategy is preferred when flexibility is needed. If we had a model that could be initialized with a large collection of data items by adding each item one at a time, notifying observers after each individual addition may dramatically degrade the performance while providing no benefit. it may be better to change the model *silently* (without notifying the observers), and then trigger a notification once the batch operation is done.]

Data Flow Between Model and Observers

How do the observers access the information that they need to know about from the model.

1. **The push data-flow strategy**
   Make the information of interest available through one or more parameters of the callback.

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

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

  ...

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

We know in advance what type of data from the model the observers will be interested in.

2. The pull data-flow strategy

Let observers "pull" the data they want from the model using query methods defined on the model.

```
public interface Observer
{ void newNumber(Model pModel); }
```

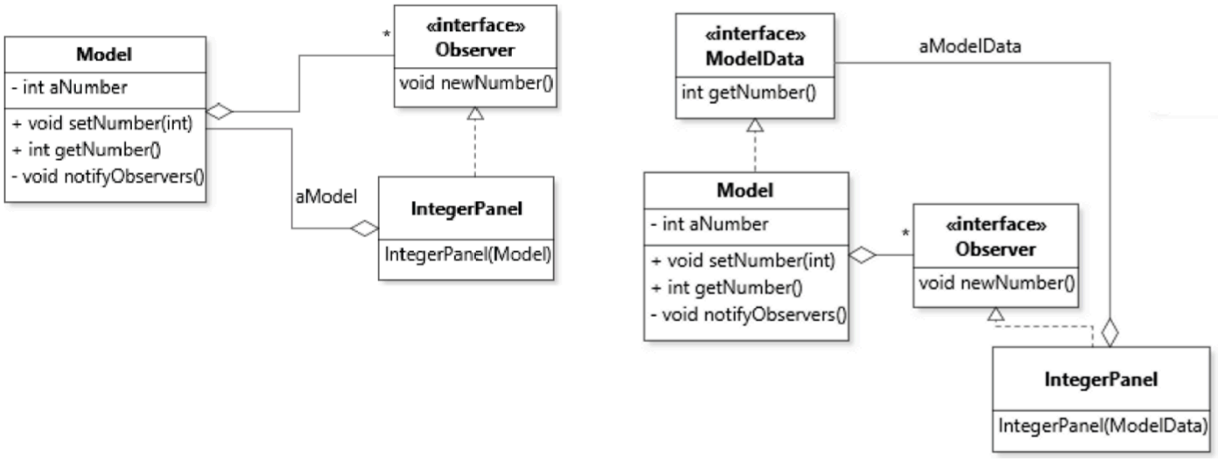
That way, the data to put in the text field must be obtained from the model:

```
class IntegerView implements Observer
{
  // User interface element that represents a text field
  private TextField aText = new TextField();

  ...

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

To implement the pull data flow strategy, observers must have a reference to the model

Solution 1. This reference is provided as an argument to the callback method.

Solution 2. Initialize observer objects with a reference to the model (stored as a field), and simply refer to that field as necessary.



At first glance, it may look like the pull data-flow strategy introduces a circular dependency between a model and its observers, given that both depend on each other. However, the crucial difference is that in this design, the model does not know the concrete type of its observers. Through interface segregation, the only slice of behavior that the model needs from observers is their callback methods. This being said, one of the main drawbacks of the pull data-flow strategy is that it does, indeed, increase the coupling between model and its observers. Observers can not only call getNumber(), they can also call setNumber(int)! In other words, by holding a reference to the model, observers have access to much more of the interface of the model than they need. Fortunately, we saw how to deal with this situation with the Interface Segregation Principle (ISP, see Section 3.2). To apply ISP to our design, we could create a new interface ModelData that only includes the getter methods for the model, and only refer to this type in the observers. Figure 8.7 illustrates this solution.

3. Combine pull strategy and push strategy

e.g. By specifying a callback that includes a parameter for both data from the model and a reference back to the model.

```
public interface Observer
{ void newNumber(int pNumber, Model pModel); }
```

4. Simple cases

For simple design contexts it may be the case that the only information that needs to flow between the model and the observers is the fact that a given callback method was invoked. In such cases, neither the push nor the pull strategy is required: receiving the callback invocation is enough information for the observers to do their job. An observer that serves as a counter of a type of event would be one example.

[Remark]
None of the callbacks return any value (i.e., they have return type void). This is not a design decision, but rather a constraint of the pattern. Because the model is supposed to ignore how many observers it has, it can be tricky for observers to attempt to manage the model by returning some value.

Event-Based Programming

One way to think about callback methods is as *events*, with the model being the *event source* and the observers being the *event handlers*. Within this paradigm, the model generates a series of events that correspond to different state changes of interest, and other objects are in charge of reacting to these events. What events correspond to in practice are simply method calls.

Imagine a situation where a Model might be used by observers that are sometimes interested only if the lucky number increased (or, conversely, decreased), or whether the number is set to its maximum or minimum value.

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

With this design, observers do not need to store a copy of the old number, and they can be notified of precisely the event they are interested in. In cases where an observer does not need to react to an event, the unused callbacks can be implemented as "do-nothing" methods. In the class below, it is assumed that the events are mutually exclusive, namely that the event increased means "increased but not to the maximum value", and similarly for decreased.

```
class IncreaseDetector implements Observer
{
  public void increased(int pNumber)
  { System.out.println("Increased to " + pNumber); }

  public void decreased(int pNumber) {}
  public void changedToMax(int pNumber) {}
  public void changedToMin(int pNumber) {}
}
```

If this "do nothing" situation occurs too often, it is possible to implement a "do nothing" class and inherit from it instead. Such "do nothing" classes are sometimes called *adapters*:

```
public class ObserverAdapter implements Observer
{
  public void increased(int pNumber) {}
  public void decreased(int pNumber) {}
  public void changedToMax(int pNumber) {}
  public void changedToMin(int pNumber) {}
}


class IncreaseDetector extends ObserverAdapter
{
  public void increased(int pNumber)
  { System.out.println("Increased to " + pNumber); }
}
```

In some cases, extensive use of "do nothing" code might point to a mismatch between the varied needs of observers and the design of the callbacks. Again, it is possible to rely on the interface segregation principle to clean things up. In our situation, we could define two observer interfaces that correspond to more specialized event handlers.

```
public interface ChangeObserver
{
  void increased(int pNumber) {}
  void decreased(int pNumber) {}
}

public interface BoundsReachedObserver
{
  void changedToMax(int pNumber) {}
  void changedToMin(int pNumber) {}
}
```

The trade-off for more flexibility is a slightly heavier interface for the Model class, because it now has to support two lists of observers with their corresponding registration methods.

[push data-flow strategy]

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

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

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

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

  private void notifyObservers()
  {
    for(Observer observer : aObservers)
    { observer.newNumber(aNumber); }
  }

  public void setNumber(int pNumber)
  {
    if( pNumber <= 0 ) { aNumber = 1; }
    else if( pNumber > 10 ) { aNumber = 10; }
    else { aNumber = pNumber; }
    notifyObservers();
  }
}
```

8.4 Applying the OBSERVER Design Pattern

We need an inventory system capable of keeping track of electronic equipment. An `Item` of equipment records a serial number and production year (both of type **`int`**). An `Inventory` object aggregates zero or more objects of type `Item`. Clients can add or remove `Items` from the `Inventory` at any time. Various entities are interested in changes to the state of the `Inventory`. For example, it should be possible to show the items in the `Inventory` in a `ListView`. It should also be possible to view a `PieChart` representing the proportion of `Items` in the `Inventory` for each production year (e.g., 2017 = 25%; 2018 = 30%, etc.). Views should be updated whenever items are added or removed from the `Inventory`.

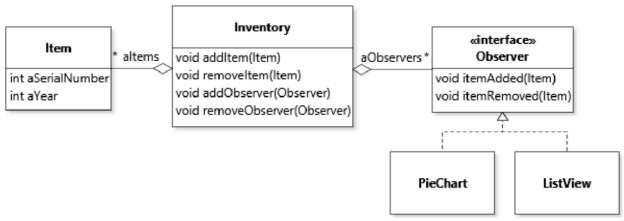


Fig. 8.11 Application of the OBSERVER to the inventory context with two callback methods using the push data-flow strategy

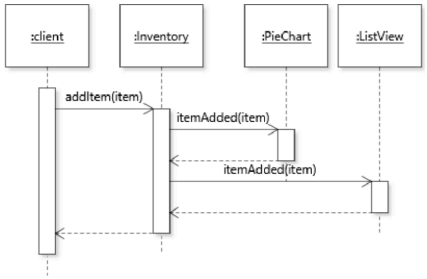


Fig. 8.12 Sequence diagram modeling an invocation of `addItem` on an inventory

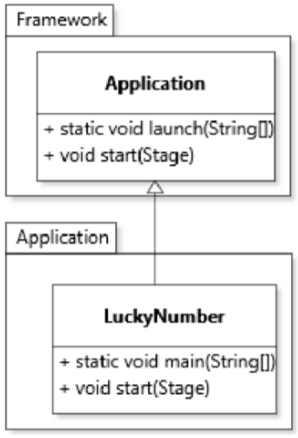
8.5 Introduction to Graphical User Interface Development

Conceptually, the code that makes up a GUI application is split into two parts:

- The framework code consists of a component library and an application skeleton. The component library is a collection of reusable types and interfaces that implement typical GUI functionality: buttons, windows, etc. The application skeleton is a GUI application that takes care of all the inevitable low-level aspects of GUI applications, and in particular monitoring events triggered by input devices and displaying objects on the screen. By itself, the application skeleton does not do anything visible: it must be extended and customized with application code.
- Application code consists of the code written by GUI developers to extend and customize the application skeleton so that it provides the required user interface functionality.

A GUI application does not execute the same way as the script-like applications we write when learning to program. In such programs, the code executes sequentially from the first statement of the application entry point (the main method in Java) and the flow of control is entirely dictated by the application code.
With GUI frameworks, the application must be started by launching the framework using a special library method. The framework's skeleton application then starts an event loop that continually monitors the system for input from user interface devices. Throughout the execution of the GUI application, the framework remains in control of calling the application code. The application code, written by the GUI developers, only get executed at specific points, in response to calls by the framework. This process is thus a clear example of inversion of control. Application code does not tell the framework what to do: it waits for the framework to call it.

The class diagram shows how the application code defines a `LuckyNumber` class that inherits from the framework's `Application` class. To launch the framework, the following code is used:




```
public class LuckyNumber extends Application
{
  public static void main(String[] pArgs)
  { launch(pArgs); }

  @Override
  public void start(Stage pPrimaryStage)
  { ... }
}
```

Method launch uses metaprogramming to discover which application class to instantiate, a bit like Object.clone() detects which class to clone.

This code calls the static method `Application.launch`, which launches the GUI framework, instantiates class `LuckyNumber` and then executes method `start()` on this instance. With this setup, class `LuckyNumber` is effectively used as the connection point between the application code used to extend the GUI and the framework code in charge of running the show.\

Conceptually, the *application code* for a GUI application can be split into two categories: the component graph, and the event handling code.
The component graph is the "actual" interface and is comprised of a number of objects that represent both visible (e.g., buttons) and invisible (e.g., regions) elements of the application. These objects are organized as a tree, with the root of the tree being the main window or area of the GUI. In modern GUI frameworks, constructing a component graph can be done by *writing code*, but also through *configuration files that can be generated by GUI building tools*. Ultimately, the two approaches are equivalent, because once the code runs the outcome is the same: a tree of plain Java objects that form the user interface. The design of the library classes that support the construction of a component graph makes heavy use of polymorphism and the COMPOSITE and DECORATOR patterns. In JavaFX, the component graph for a user interface is typically instantiated in the application's start(Stage) method.
Once the framework is launched and displaying the desired component graph, its event loop will automatically map low-level system events to specific interactions with components in the graph (e.g., placing the mouse over a text box, clicking a button). In common GUI programming terminology, such interactions are called events. Unless specific application code is provided to react to events, nothing will happen as a result of the framework detecting an event. For example, clicking on a button will graphically show the button to be clicked using some user interface cue, but then the code will simply continue executing without having any impact on the application logic. To build interactive GUI applications, it is necessary to handle events like button clicks and other user interactions. Event handling in GUI frameworks is an application of the OBSERVER pattern, where the model is a GUI component (e.g., a button). Handling a button click, or any similar event, then just becomes a question of defining an observer and registering it with the button. The next two sections detail how to design component graphs and handle events on GUI components.

8.6 Graphical User Interface Component Graphs

The component graph is the collection of objects that forms what we usually think of as the interface: windows, textboxes, buttons, etc. At different stages in the development of a graphical user interface, it can be useful to think about this user interface from four different point of views, or perspectives: user experience, logical, source code, run-time.

The Logical Perspective

The logical perspective on the graphical user interface is a way to think about it from the point of view of the hierarchical organization of UI components, independently of their definition in code. Figure 8.15 shows the logical perspective on the LuckyNumber application. This model of the component graph shows that the user interface consists of one `Slider` instance and two `TextField` instances, each wrapped in a `Parent` region component, which are themselves children of an outer region of type `GridPane`, which is the child of the top-level element of the component graph, the `Scene`.[8]

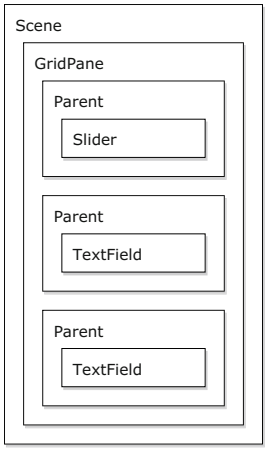


Fig. 8.15 Logical perspective on the LuckyNumber application

The logical perspective is often a useful complement to the other perspectives, because it shows how components relate to each other hierarchically without getting cluttered by the visual rendering of the components or details of the source code. In simpler applications, where there is a minimum of visual overlap between components, the logical view can also serve as a sketch of the physical arrangement of objects in a layout.

The Source Code Perspective

The source code perspective shows the kind of information about the component graph that is readily available from the declarations of the classes of the objects that form the component graph. This information is best summarized by a class diagram. Figure 8.16 models the source code perspective on the component graph of the LuckyNumber application. Despite the application being tiny, the diagram

[8] Technically in JavaFX the `Scene` needs to be set as the child of a further parent called a `Stage`, but one could argue that the `Stage` is not really part of the component graph.

shows that a lot of code is required to instantiate its component graph. Let us walk through this diagram.

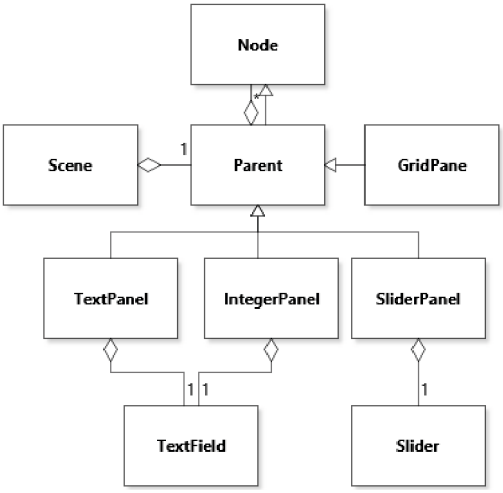


Fig. 8.16 Source code perspective on the LuckyNumber application

The `Scene` holds a reference to the root node of the component graph, something we can deduce from the fact that it is not a subtype of `Node`, and no class in the diagram aggregates it. The `Scene` class aggregates class `Parent`. This may be puzzling at first, because in my logical model of the component graph, I indicated that the `Scene` contains a `GridPane`. This is an example of polymorphism in use. To allow users to build any kind of application, the `Scene` library class accepts any subtype of type `Parent` as its child object. In turn, `Parent` is a subtype of the general `Node` type that adds functionality to handle children nodes. In JavaFX, all objects that can be part of a component graph need to be a subtype of `Node`, either directly or, more generally, indirectly by inheriting from other subtypes of `Node`. The fact that `Parent` nodes, which can contain children nodes, are themselves of type `Node` shows that the design of the GUI component hierarchy is an application of the COMPOSITE pattern.

By continuing our investigation of the diagram, we find class `GridPane` as a subtype of `Parent`. This is the reason it is possible to add a `GridPane` to a scene. A `GridPane` is a type of user interface `Node` that specializes in organizing its children into a grid. I used it for LuckyNumber to easily lay out the number views vertically on top of each other.

In the LuckyNumber application, a `GridPane` contains a set of `Parent` components. In the general case, a `GridPane` can contain any subtype of `Node`. However, in my design of the application, I created three classes that inherit from `Parent`: `TextPanel`, `IntegerPanel`, and `SliderPanel`. These classes represent the three views of the number in the Model–View–Controller decomposition. By defining these classes as subclasses of `Parent`, I achieve two useful properties:

- I *reuse* the "parenting" functionality of `Parent` to easily add a widget (e.g., a slider) to a `Node`;
- By defining my view classes as subtypes of `Node`, I make it possible to add them as children of a `GridPane` through polymorphism.

The remainder of the diagram shows how the tree would generate its leaves: the `SliderPanel` aggregates a `Slider` instance, and both the `TextPanel` and the `IntegerPanel` aggregate a `TextField` instance. Note that because this is a class diagram and not an object diagram, the fact that both `TextPanel` and `IntegerPanel` have an association to the `TextField` model element does *not* mean that their instance would refer to the same `TextField` instance!

The diagram of Figure 8.16, already somewhat involved, actually omits, for clarity, many intermediate types in the inheritance hierarchy for nodes. For example, the diagram shows `GridPane` to be a direct subclass of `Parent`. In reality, `GridPane` is a subclass of `Pane`, which itself is a subclass of `Region`, which is a subclass of `Parent`. Figure 8.17, while still an incomplete model, shows a bigger picture of the class hierarchy that can be leveraged to define component graphs in JavaFX.

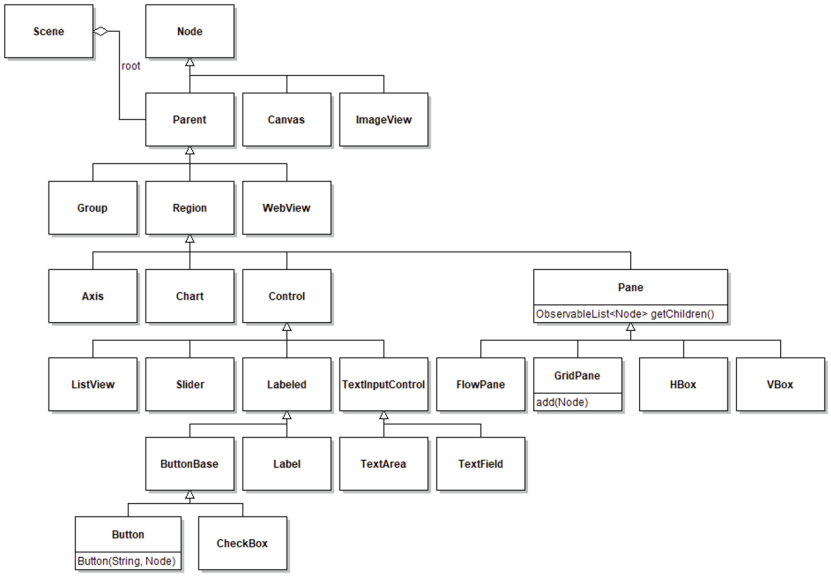


Fig. 8.17 Partial `Node` class hierarchy in JavaFX

The Run-time Perspective

The run-time perspective is the instantiated component graph for a graphical interface. This perspective can best be represented as an object diagram. Figure 8.18 shows the instantiated component graph for LuckyNumber. Conceptually close to

the logical view, it differs by making object identities explicit while losing, in this case, the relative positioning of graphical components.

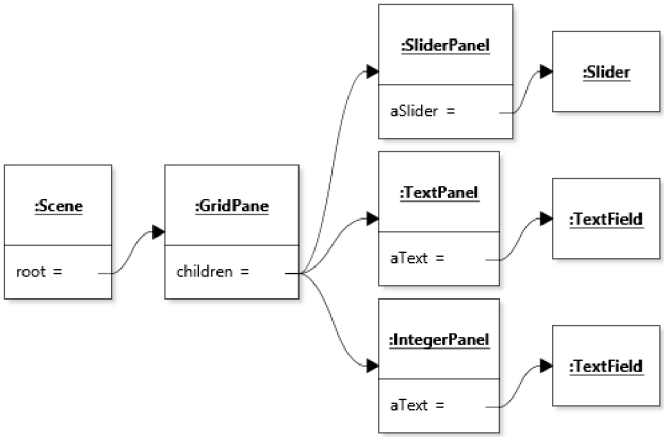


Fig. 8.18 Run-time perspective of the LuckyNumber user interface

Defining the Object Graph

In Section 8.5 I mentioned how after the framework starts it calls the `start` method of the main application class (`LuckyNumber` in our case). This `start` method is the natural integration point for extending the framework, and this is where we put the code that builds the component graph. The code below is the minimum required to get the application to create the LuckyNumber component graph. In practice, this kind of code would typically be extended with additional configuration code and organized using helper methods. The additional configuration code can be used to beautify the application, for example by adding margins around components, a title to the window, etc. The JavaFX functionality to generate component graphs from configuration files is outside the scope of this book.

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

    GridPane root = new GridPane();
    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();
  }
}
```

The first statement of method `start` is to create an instance of `Model`. This instance will play the role of the model in the OBSERVER pattern. It is related to the construction of the component graph because, as detailed later, some of the components in the graph need access to the model. The second statement creates a `GridPane`, which is an invisible component used for assisting with the layout of children components. The local variable that holds a reference to this component is helpfully named `root` to indicate that it is the root of the component graph. Then, three application-defined components are added to the grid. The parameters to the `add` method indicate the column and row index and span. For example, the statement:

```
root.add(new SliderPanel(model), 0, 0, 1, 1);
```

specifies to add an instance of the `SliderPanel` in the top-left cell in the grid, and span only one column and one row. Because `SliderPanel` is a subtype of `Parent`, and thus a subtype of `Node`, it can be added to the grid. Another important thing to note about the instantiation of the panel components is that their constructor takes as argument a reference to the model.

The last two statements of the method are not really related to the construction of the component graph, but are nevertheless crucial steps in the creation of the GUI. The statement with the call to `setScene` creates a `Scene` from the component graph and assigns it to the framework's `Stage`. Finally, the last statement requests that the framework display the `Stage` onto the user's display.

For additional insights on the creation of the component graph, the code below shows the relevant part of the constructor of the `IntegerPanel` (the other panels are very similar).

```
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);
    ...
  }

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

This code illustrates a number of insights about the design of the component graph. First, as already mentioned, the application-defined `IntegerPanel` class extends the framework-defined `Parent` class so that it can become part of the component graph. Second, an instance of `IntegerPanel` aggregates a framework-defined `TextField` component. However, the mere fact of defining an instance

variable of type `TextField` inside the class does not add the `TextField` to the component graph. To do this, it is necessary for the `IntegerPanel` to add the instance of `TextField` to itself, something that is done with the call `getChildren().add(aText)`. Method `getChildren()` is inherited from class `Parent`, and used to obtain the list of children of the parent user interface `Node`, to which the `TextField` instance can then be added.

The `IntegerPanel` instance also maintains a reference to the `Model`. The reason for this is that the `IntegerPanel` needs to act as a controller for the `Model`, something that will be explained in more detail in the next section. Also, it is worth noticing how the `IntegerPanel` is an observer of the `Model` instance: it declares to implement `Observer`, it registers itself as an observer upon construction (second statement of the constructor), and it supplies an implementation for the `newNumber` callback. As expected, the behavior of the callback is to set the value of the `TextField` user interface component with the most recent value in the model, obtained from the callback parameter.

As a final insight on the design of the component graph, we can note how the instance of `Model` created in method `start` (see preceding code fragment) is stored in a *local variable*. In other words, the application class `LuckyNumber` does *not* manage an instance of the model: this is only done within each panel. This design decision is to respect the guideline provided in Chapter 4, to keep the number of fields to a minimum. Without care, application-defined user interface components can easily become a GOD CLASS† bloated with numerous references to stateful objects, which makes a design much harder to understand.

8.7 Event Handling

In GUI frameworks, objects in the component graph act as models in the OBSERVER. Once the framework is launched, it continually goes through a loop that monitors input events and checks whether they map to events that can be observed by the application code. This process is illustrated in Figure 8.19.

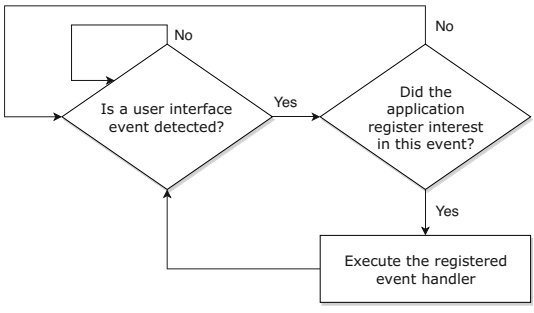


Fig. 8.19 The event loop in a GUI framework

Typically, events are defined by the component library supplied by the framework. For example, the `TextField` user interface component defines an "action" event. According to its class documentation "The action handler is normally called when the user types the ENTER key". This means that an instance of `TextField` can play the role of the model in the OBSERVER. Figure 8.20 shows the correspondence between the code elements and the roles in the OBSERVER pattern.

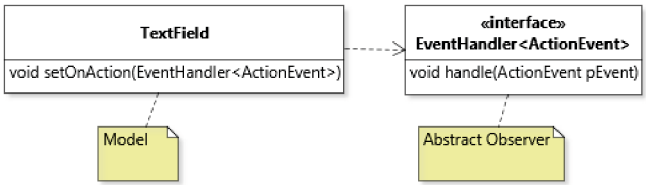


Fig. 8.20 Correspondence between `TextField` and the roles in the OBSERVER

Handling the action event on a text field is thus pretty straightforward. All we need to do is to:

- **Define a handler for the event.** This means defining a class that is a subtype of `EventHandler<ActionEvent>`. The class will be our event handler class.
- **Instantiate a handler.** This means creating an instance of the class we defined in the previous step. The instance will be our event handler instance, or more simply event handler, or even just "handler".
- **Register the handler.** This means calling the registration method on the model and passing the handler as an argument. In the case of `TextField`, we need to call `setOnAction(handler)`. It is worth noticing an interesting design choice for this application of OBSERVER: it is only possible to have a single observer for a `TextField`.

Although the basic mechanism for specifying and registering event handlers is always the same, one design choice that must be resolved is where to place the definition of the handling code. For this, two main strategies are possible:

- **To define the handler as a function object** using an anonymous class or a lambda expression (see Section 3.4). This is a good choice if the code of the handler is simple and does not require storing data that is specific to the handler;
- **To delegate the handling to an element of the component graph** by declaring to implement the observer interface. This is a good choice if the code of the handler is more complex or requires knowing about many different aspects of the internal structure of the target component.

Let us see how these two options can be realized in the context of the LuckyNumber application. Using the function object strategy, we could complete the code of the constructor of `IntegerPanel` as follows:

```
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 = 1;
          try{ number = Integer.parseInt(aText.getText()); }
          catch(NumberFormatException pException )
          { /* Just ignore. We use 1 instead. */ }
          aModel.setNumber(number);
        }
      });
  }
}
```

With this strategy, the constructor of `IntegerPanel` creates a function object using an anonymous class and, at the same time, registers this object to become the handler of the action event on the text field. The behavior of the handler is to serve as the controller for the model.

At this point in the design of the application, we now have *two* applications of the OBSERVER at play. One subject is the `Model` being observed by all three panels, and another subject is the `IntegerPanel`'s `TextField` that is observed by the anonymous function object. Figure 8.21 captures the design. Naturally, in the finished application, we would also need an event handler for the text panel and the slider panel, which would bring the total number of applications of the OBSERVER to four.

In the case of the LuckyNumber application, one alternative to using function objects for defining handlers is to delegate the handling of GUI events to the panels themselves. In our case, this would mean declaring `IntegerPanel` to implement both `Observer` and `EventHandler<ActionEvent>`. The `Observer` interface is the same one as before, used to receive callbacks when the model (the number) is changed. The difference in this case is the addition of the `EventHandler` interface, which allows the `IntegerPanel` to respond to the event that corresponds to the "Enter" key being pressed in the panel's text field.

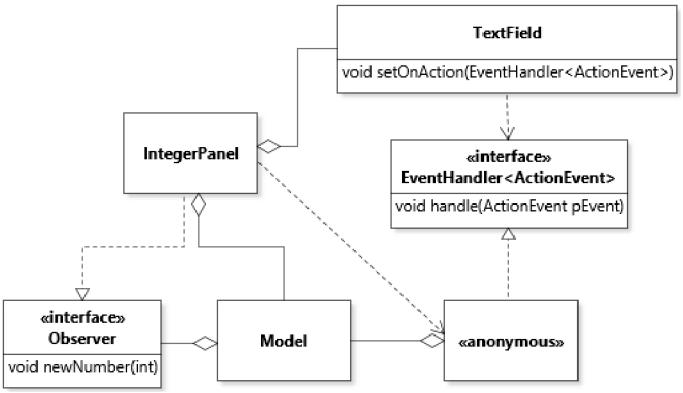


Fig. 8.21 Two applications of the OBSERVER pattern

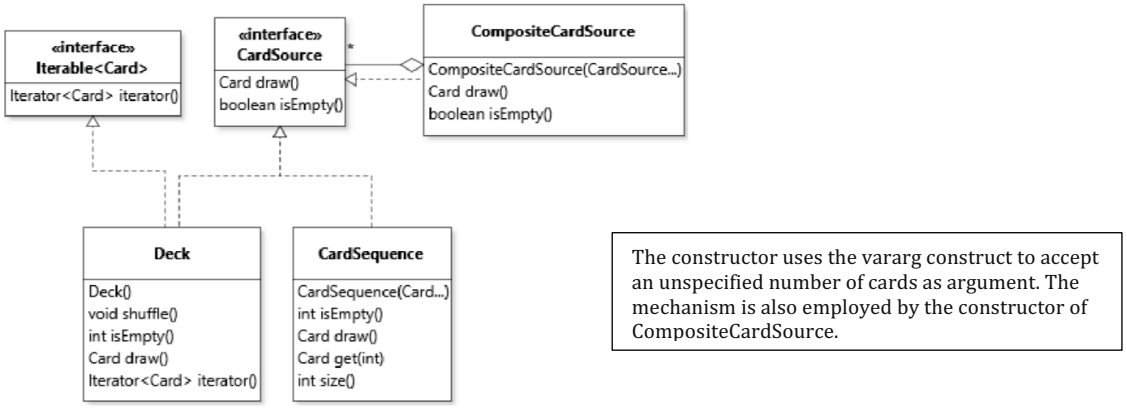
```
public class IntegerPanel extends Parent implements Observer,
  EventHandler<ActionEvent>
{
  private TextField aText = new TextField();
  private Model aModel;

  public IntegerPanel(Model pModel)
  {
    aModel = pModel;
    aModel.addObserver(this);
    aText.setText(new Integer(aModel.getNumber()).toString());
    getChildren().add(aText);
    aText.setOnAction(this);
  }

  public void handle(ActionEvent pEvent)
  {
    int number = 1;
    try { number = Integer.parseInt(aText.getText()); }
    catch(NumberFormatException pException )
    { /* Just ignore. We'll use 1 instead. */ }
    aModel.setNumber(number);
  }

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

There are two main implications of this choice on the code. First, the `handle` method needs to be declared directly in class `IntegerPanel`. Second, the argument passed to `aText.setOnAction` is now **`this`**, because it is the `IntegerPanel` instance itself that is now responsible for handling the event.

Although both design options for locating the handler code are workable, for the LuckyNumber application I prefer the function object alternative. The handler code

8.8 The VISITOR Design Pattern

Support an open-ended number of operations that can be applied to an object graph, but without impacting the interface of the objects in the graph.



[drawbacks of adding methods to interfaces]

- The interface will get much bigger. Not all methods might be used in all usage contexts. As mentioned above, there is a risk of violating the interface segregation principle;
- For a versatile data structure that can be used as a library, it may be hard to anticipate which operations are going to be necessary in the future. Adding operations that end up unused is a clear case of SPECULATIVE GENERALITY†. In fact, if the code is indeed distributed as a library, future users may not be able to, or want to, change the code to add additional operations.

The VISITOR provides a solution in such a context by supporting a mechanism whereby it is possible to define an operation of interest in a separate class and "inject" it into the class hierarchy that needs to support it.

Abstract and Concrete Visitors

The cornerstone of the VISITOR pattern is an interface that describes objects that can "visit" objects of all classes of interest in an object graph. This interface is called the abstract visitor. An abstract visitor follows a prescribed structure: it contains one method with signature `visitElementX(ElementX pElementX)` for each different type of concrete class `ElementX` in the object structure. In our case, the abstract visitor would be defined as follows:

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

A concrete visitor is an implementation of this interface. In the VISITOR pattern, we implement one concrete visitor for each operation of interest. In a concrete visitor, each `visitElementX`

method provides the behavior of the operation as applied to a given class. For example, a simple visitor that prints all the cards in a card source to the console would be defined as such:

```
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)); }
 }
}
```

don't require Deck and CardSequence to have the same interface. They can use whatever methods are available on the concrete type to implement the required behavior

Concrete visitor provides a way to organize code in terms of *functionality* as opposed to *data*. In a classic design, the code to implement the printing operation would be scattered throughout the three card source classes. In this design, all this code in located in a single class. One of the benefits of the VISITOR is thus to allow a different style of assignment of responsibilities to classes, and thus a separation of concerns along a different criterion (functionality-centric vs. data-centric).

Integrating Operations into a Class Hierarchy

Although a concrete visitor separates a well-defined operation into its own class, it still needs to be integrated with the class hierarchy that defines the object graph on which the operation will be applied (henceforth simply referred to as the "class hierarchy").
This integration is accomplished by way of a method, usually called `accept`, that acts as a gateway into the object graph for visitor objects. An accept method takes as single argument an object of the abstract visitor type (CardSourceVisitor in our case). Unless there is good reason not to, we normally define the accept method on the common supertype of the class hierarchy:

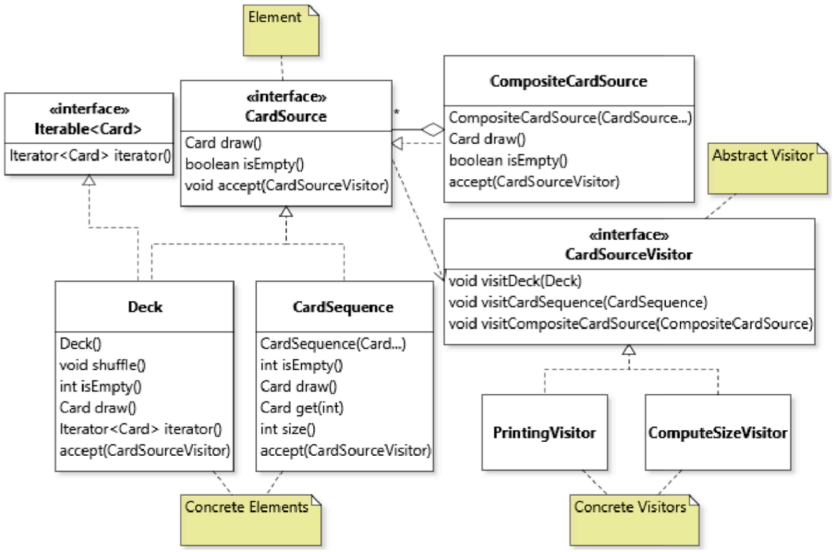
```
public interface CardSource
{
  Card draw();
  boolean isEmpty();
  void accept(CardSourceVisitor pVisitor);
}
```

The implementation of `accept` by concrete types is where the integration really happens. This implementation follows a prescribed formula: to call the visit method for the type of the class that defines the accept method.
For example, the implementation of accept for class `Deck` is:

```
public void accept(CardSourceVisitor pVisitor)
{ pVisitor.visitDeck(this); }
```

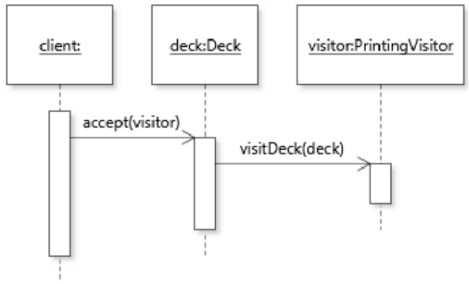
and the one for class CardSequence is:

```
public void accept(CardSourceVisitor pVisitor)
{ pVisitor.visitCardSequence(this); }
```



With the accept method in place, executing an operation on the object graph is now a matter of creating the concrete visitor object that represents the operation, and passing this object as argument to method accept on the target element:

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



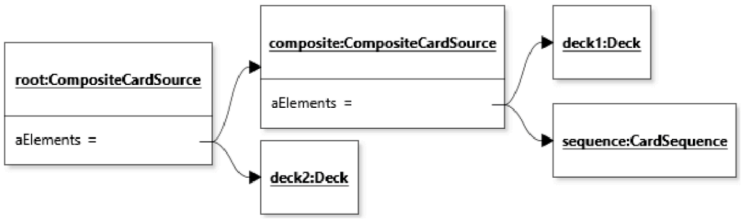
The accept method then calls back the appropriate method on the visitor. In this sequence, the visitDeck method qualifies as a callback method.

Traversing the Object Graph

Any object graph with more than one element will have an aggregate node as its root.
In our case this is CompositeCardSource

```
public class CompositeCardSource implements CardSource
{
  public void accept(CardSourceVisitor pVisitor)
  {
    pVisitor.visitCompositeCardSource(this);
  }
}
```

The two core ideas of the VISITOR pattern are to (1) enable the integration of an open-ended set of operations that (2) can be applied by traversing an object graph, often a recursive one. For the traversal aspect of the pattern to be applicable, at least one element type in the target hierarchy needs to serve as an aggregate for other types.



There are two main ways to implement the traversal of the object graph in the VISITOR.

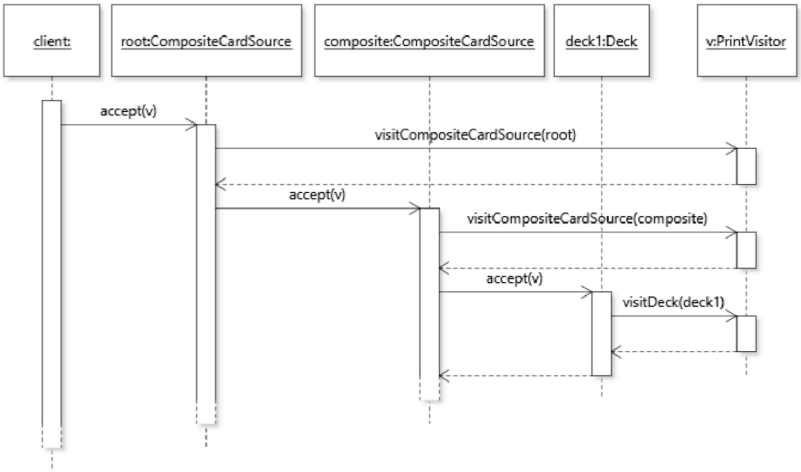
First option Place the traversal code in the accept method of aggregate types.

In our case, placing the traversal code in the accept method is relatively straightforward:

```
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); }
  }
}
```

[advantage] Stronger encapsulation
Because the traversal code is implemented within the class of the aggregate, it can refer to the private field that stores the aggregation (`aElements`). This access to private structures is one major motivation for implementing the traversal code within the accept method.

[disadvantage]
The traversal order is fixed in the sense that it cannot be adapted by different visitors.

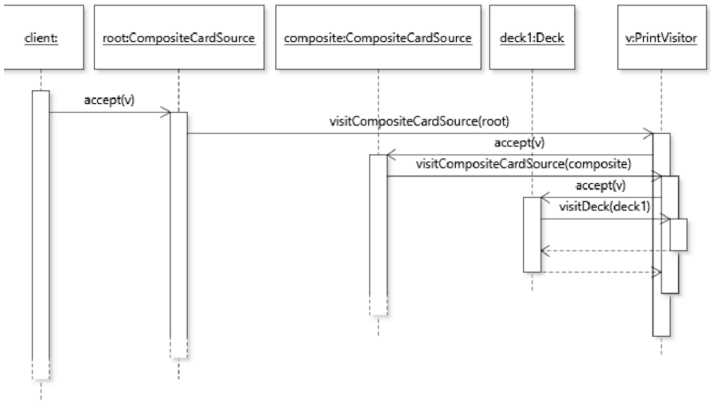


Second option Place this code in the visit methods that serve as callbacks for aggregate types.

Unfortunately, in our context it is not possible to implement this option directly, because the aggregate class offers no public access to the `CardSource` objects it aggregates. Because the code of the visit methods is in a separate class, we need a way to access the objects stored by the private field `aElements`. To make this work we make `CompositeCardSource` iterable over the CardSource instances it aggregates. However, this requirement to decrease the level of encapsulation of the class is a disadvantage of this design decision.

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

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

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



Conclusion

If encapsulation of target elements is more important, it is better to place the traversal code in the accept method. If the ability to change the traversal order is more important, then it is better to place the traversal code in the visit method.

Using Inheritance in the Pattern

The question of where to place the traversal code brings up the issue of code DUPLI- CATED CODE† again. If we place the traversal code in the visit methods, and have more than one concrete visitor class, every class is bound to repeat the traversal code in its visit method. A common solution to alleviate this issue is to define an abstract visitor class to hold default traversal code. In our case, the following would be a good implementation of an abstract visitor class:

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

  public void visitDeck(Deck pDeck) {}

  public void visitCardSequence(CardSequence pCardSequence) {}
}
```

Here it is important to distinguish between an **abstract** visitor *class* and an abstract *visitor*, which is usually an interface.

- First, I retained the interface. Because most concrete visitors would be implemented as sub-classes of `AbstractCardSourceVisitor`, one can wonder, why not simply use this abstract class to serve in the role of abstract visitor, and get rid of the interface? The general reason is that interfaces promote more flexibility in a design. For example, one concrete drawback of using an abstract class is that, because Java only supports single inheritance, defining the abstract visitor as an abstract class prevents classes that already inherit from another class to serve as concrete visitors.
- The second notable detail in the above code is that the visit methods for classes `Deck` and `CardSequence` are implemented as empty placeholders. Given that `AbstractCardSourceVisitor` is declared abstract, we do not need these declarations. However, providing empty implementations for visit methods allows the abstract visitor class to serve as an adapter (Section 8.3: *adapter classes*). In more realistic applications of the pattern, the element type hierarchy can have dozens of different types, with a corresponding high number of visit methods. With empty implementations, concrete observers only need to override the methods that correspond to types they are interested in visiting.

e.g. Print the number of cards in every `CardSequence` in a card source structure, and ignores the rest.

```
CardSourceVisitor visitor = new AbstractCardSourceVisitor()
  {
    public void visitCardSequence(CardSequence pSequence)
    { System.out.println(pSequence.size() + " cards"); }
  };
```

Because the class inherits the traversal code, card sequences aggregated within composite card sources will also be reached.

e.g. Print a representation of the object graph that includes the nesting depth of a card source type:

```
public class StructurePrinterVisitor
  extends AbstractCardSourceVisitor
{
  private int aTab = 0;

  private String tab()
  {
    StringBuilder result = new StringBuilder();
    for( int i = 0; i < aTab; i++ )
    { result.append(" "); }
    return result.toString();
  }

  public void visitCompositeCardSource(
    CompositeCardSource pCompositeCardSource)
  {
    System.out.println(tab() + "Composite");
    aTab++;
    super.visitCompositeCardSource(pCompositeCardSource);
    aTab--;
  }
  public void visitDeck(Deck pDeck)
  { System.out.println(tab() + "Deck"); }

  public void visitCardSequence(CardSequence pCardSequence)
  { System.out.println(tab() + "CardSequence"); }
}
```

The result of using this visitor on the object graph of Figure 8.25 would be:

```
Composite
   Deck
   Composite
    Deck
    CardSequence
```

This example introduces two new aspects to our discussion so far.
First, the visitor is *stateful*, i.e., it stores data. Specifically, the class defines a field `aTab` that stores the depth of the element currently being visited. Depth increases when visiting the elements aggregated by a composite card source.
Second, the reuse of the traversal code through a super call. Here, the pre-order traversal implemented in the abstract visitor class is what we need. However, additional code is required when visiting a composite card source. To make this possible, `visitCompositeCardSource` is overridden to manage the indentation level, and a super call is made to trigger the traversal code at the appropriate point.

Supporting Data Flow in Visitor Structures

To support a general and extensible mechanism for defining operations on an object graph, the pattern requires that no assumption be made about the nature of the input and output of operations.
Data flow for VISITOR-based operations is thus implemented differently, by storing data within a visitor object. Input values can be provided when constructing a new visitor object and made accessible to the visit methods. Output values can be stored internally by visit methods during the traversal of the object graph, and made accessible to client code through a getter method. Let us consider each case in turn, starting with output values.

e.g. count the total number of cards in the source. This version assumes that the abstract visitor class defined above is available:

```
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; }
}
```

To use this operation, it would be necessary to store a reference to the constructed visitor in a variable so that that the count can later be retrieved:

```
CountingVisitor visitor = new CountingVisitor();
root.accept(visitor);
int result = visitor.getCount();
```

e.g. check whether a card source structure contains a certain card. Such an operation requires both input and output.

```
public class ChecksContainmentVisitor
  extends AbstractCardSourceVisitor
{
  private final Card aCard;
  private boolean aResult = false;

  public ChecksContainmentVisitor(Card pCard)
  { aCard = pCard; }

  public void visitDeck(Deck pDeck)
  {
    for( Card card : pDeck)
    {
      if( card.equals(aCard))
      {
        aResult = true;
        break;
      }
    }
  }

  public void visitCardSequence(CardSequence pCardSequence)
  {
    for( int i = 0; i < pCardSequence.size(); i++ )
    {
      if( pCardSequence.get(i).equals(aCard))
      {
        aResult = true;
        break;
      }
    }
  }

  public boolean contains()
  { return aResult; }
}
```

Although this implementation works, it is not as efficient as it should be because aggregate nodes are traversed even when a card has already been found. Fortunately, the structure of the VISITOR allows us to eliminate this source of inefficiency with very little impact on the overall design: all we need to do is to provide an implementation for `visitCompositeCardSource` that only triggers the traversal if the card has not already been found.

```
public void visitCompositeCardSource(
  CompositeCardSource pCompositeCardSource)
{
  if( !aResult )
  { super.visitCompositeCardSource(pCompositeCardSource); }
}
```