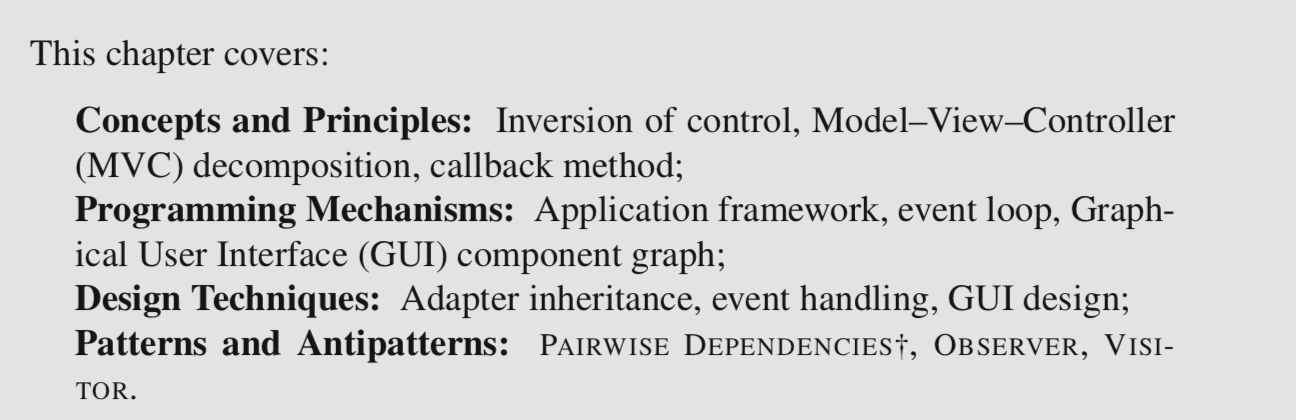
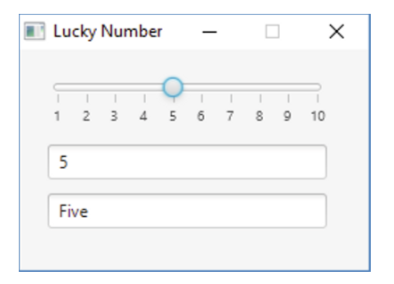
Chapter 8

Inversion of Control



8.1 Motivating Inversion of Control

Synchronization:

keep different objects consistent with each other.

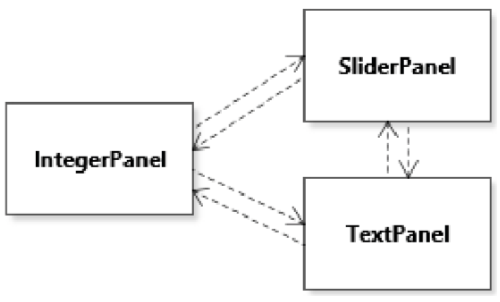
**Naïve approach**

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

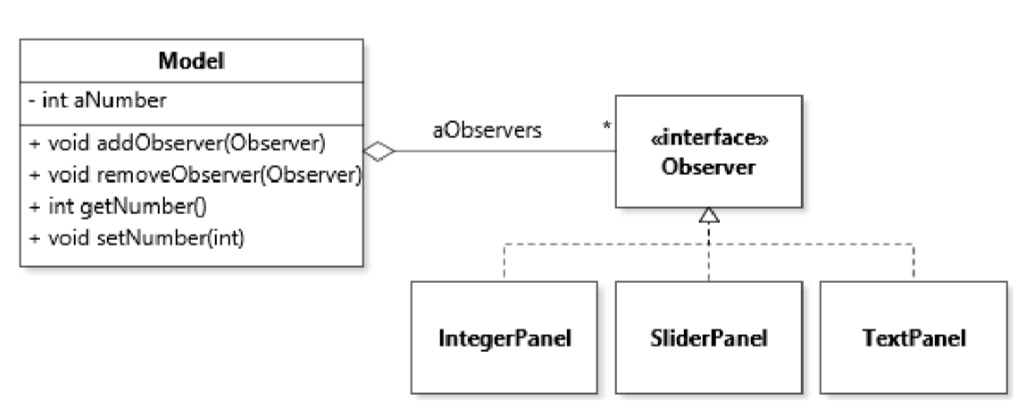
Separate abstractions responsible for *storing data* from abstractions responsible for *viewing data*, from abstractions responsible for *changing data*.

🡪 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.

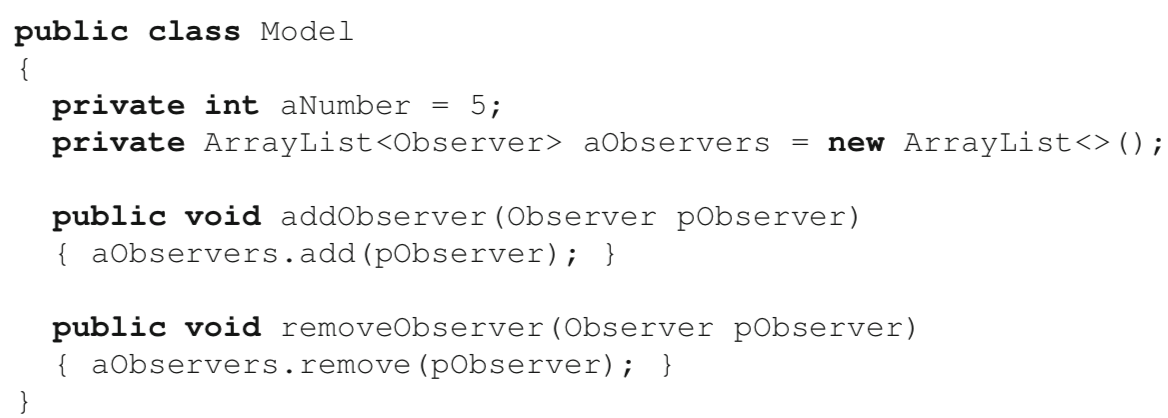
8.3 The OBSERVER Design Pattern

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.



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

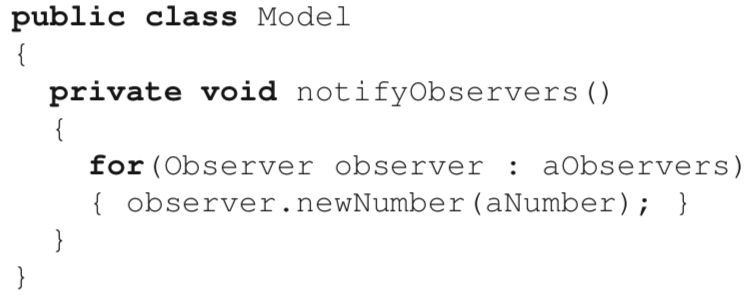


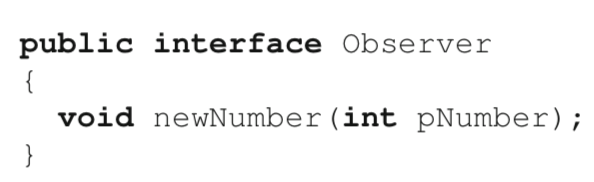
Through polymorphism, we thus achieve loose coupling between the model and its observers.

* 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).

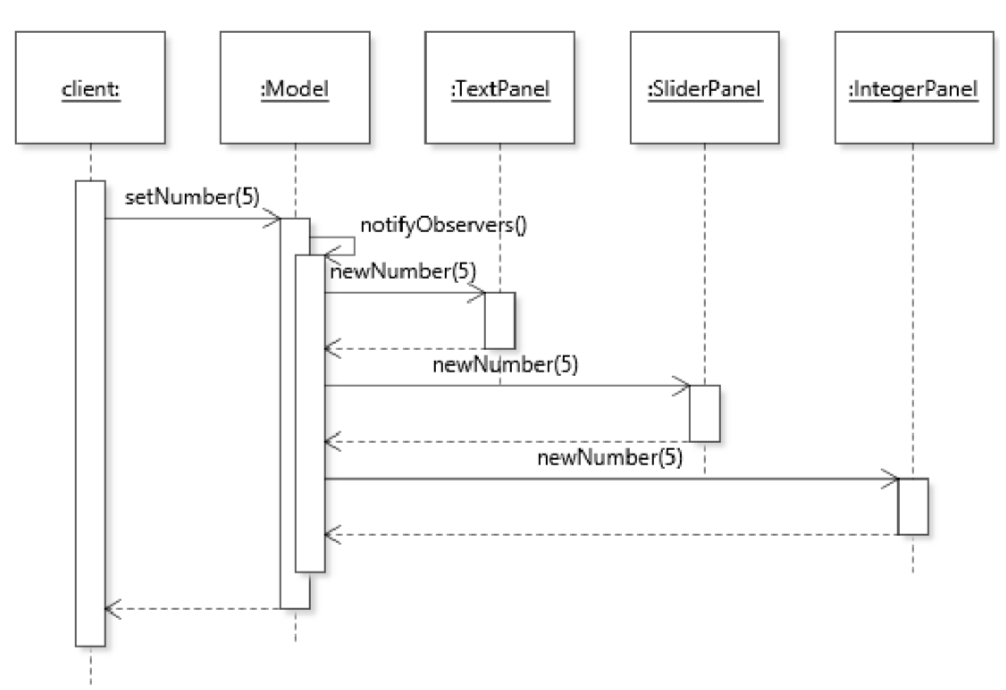






The idea of a callback is not to tell observers what to do, but rather to *inform* observers about some change in the model, and let them deal with it as they see fit (through the logic provided in the callback). Once we have a callback defined, within class Model, we can create a helper method, called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

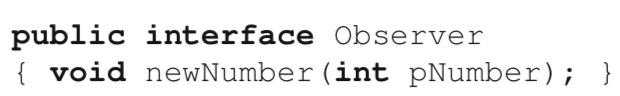
[ This strategy is preferred when flexibility is needed. If we had a model that could be initialized with a large collection of data items by adding each item one at a time, notifying observers after each individual addition may dramatically degrade the performance while providing no benefit. it may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

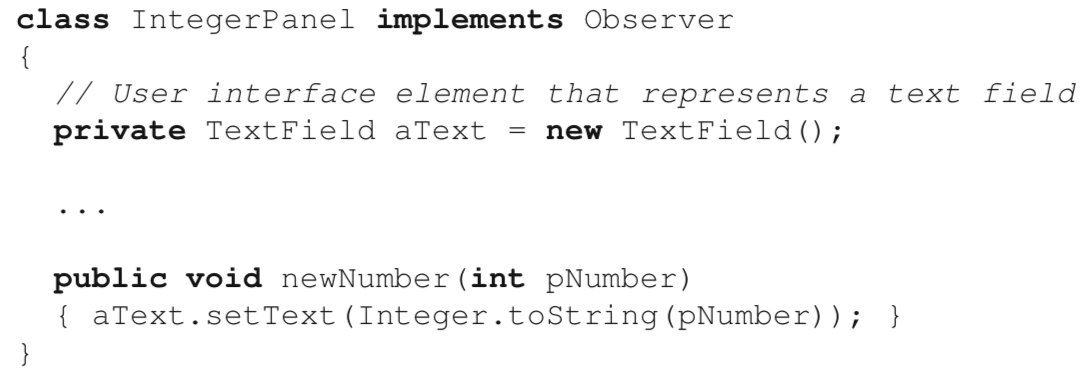
**Data Flow Between Model and Observers**

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.*

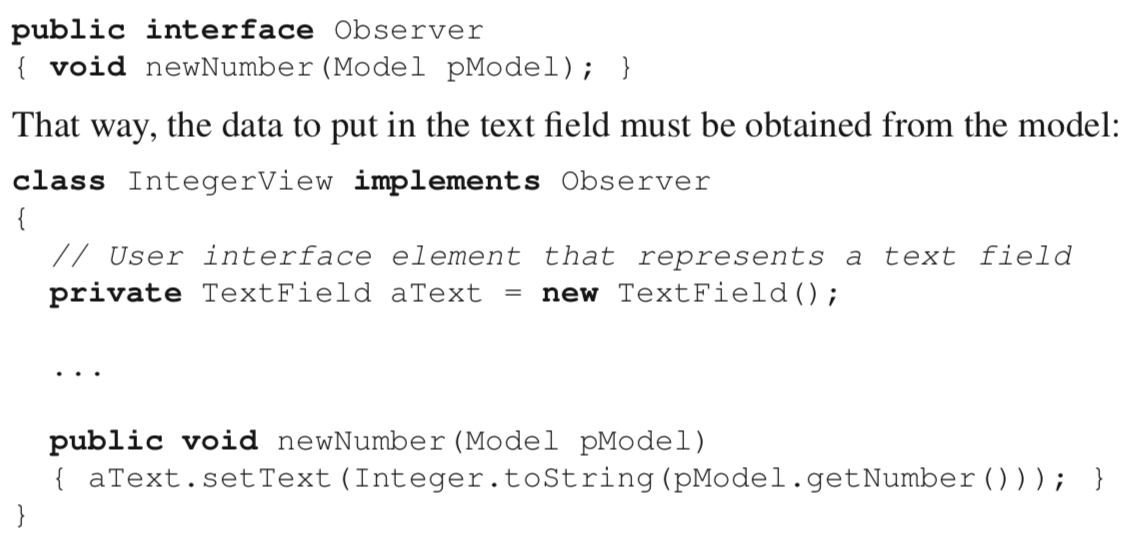
**



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

**2. The pull data-flow strategy**

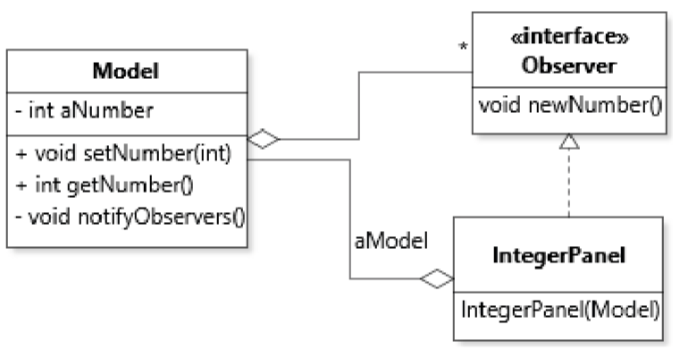
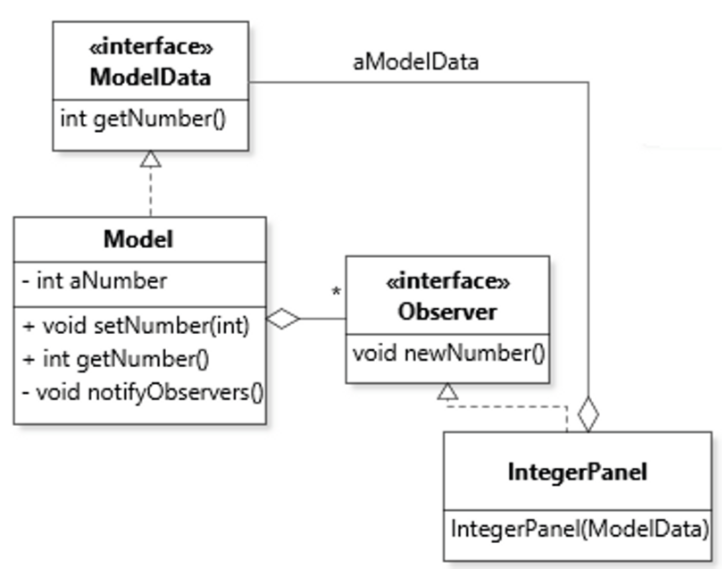
Let observers “pull” the data they want from the model using query methods defined on the model.



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.

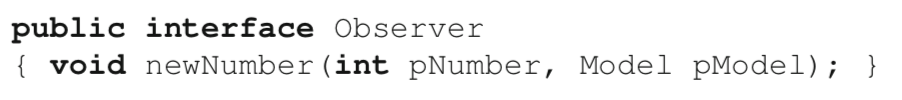
Solution 2. Initialize observer objects with a reference to the model (stored as a field), and simply refer to that field as necessary.



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.



**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.

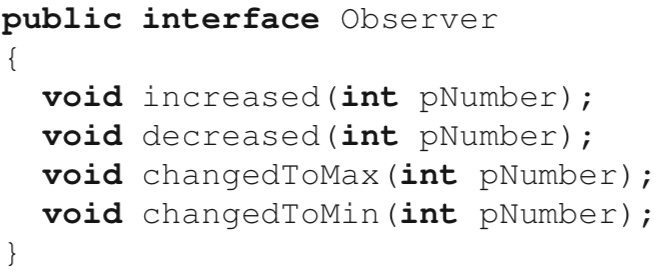
[Remark]

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.

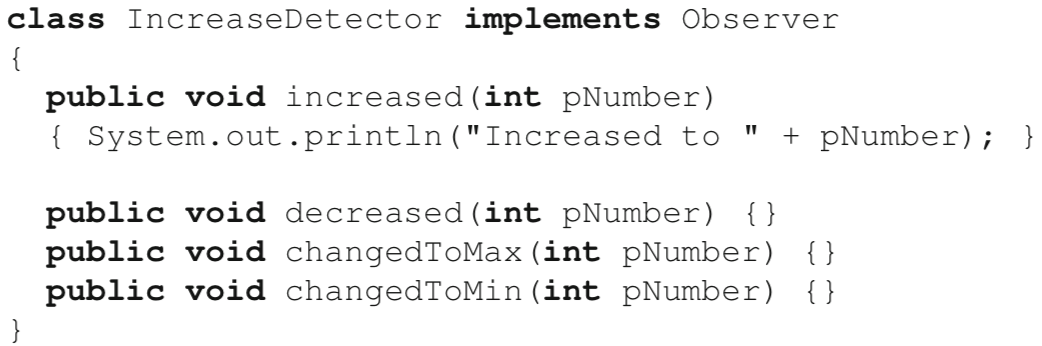
**Event-Based Programming**

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.

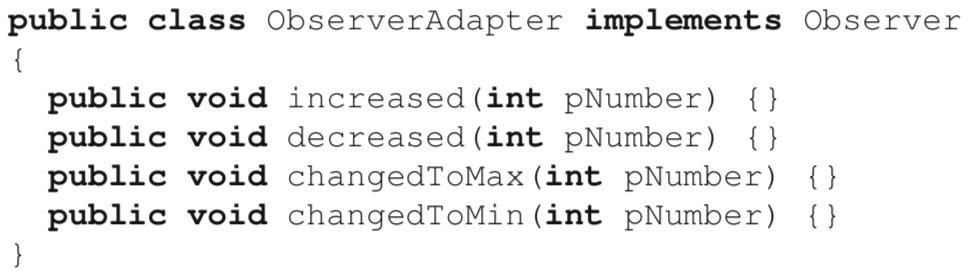
Imagine a situation where a Model might be used by observers that are sometimes interested only if the lucky number increased (or, conversely, decreased), or whether the number is set to its maximum or minimum value.

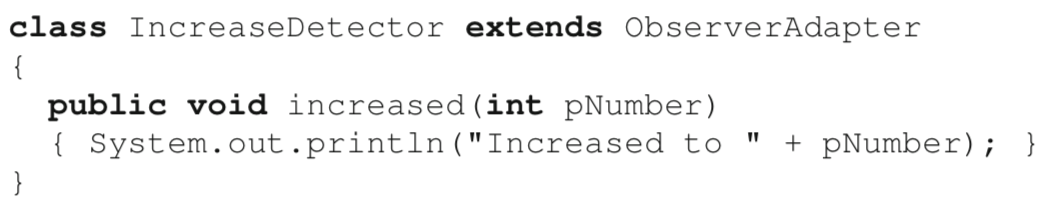


With this design, observers do not need to store a copy of the old number, and they can be notified of precisely the event they are interested in. In cases where an observer does not need to react to an event, the unused callbacks can be implemented as “do-nothing” methods. In the class below, it is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

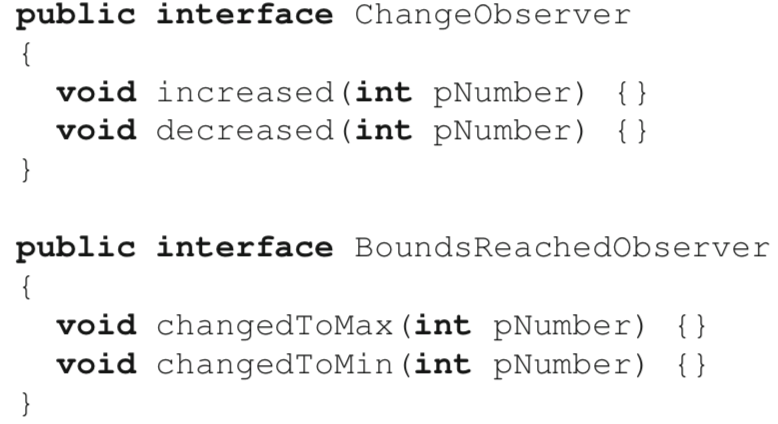


If this “do nothing” situation occurs too often, it is possible to implement a “do nothing” class and inherit from it instead. Such “do nothing” classes are sometimes called *adapters*:



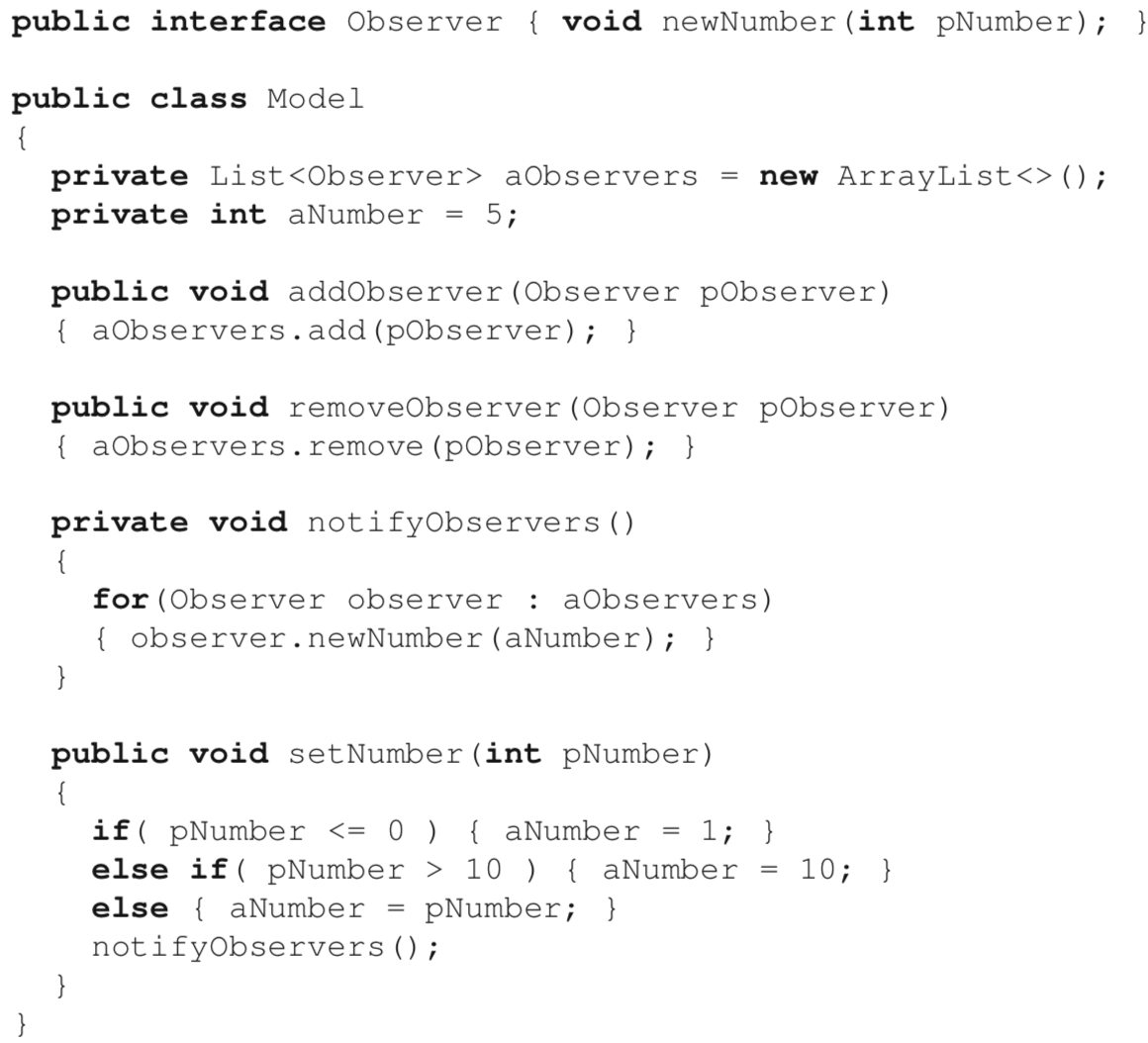


In some cases, extensive use of “do nothing” code might point to a mismatch between the varied needs of observers and the design of the callbacks. Again, it is possible to rely on the interface segregation principle to clean things up. In our situation, we could define two observer interfaces that correspond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

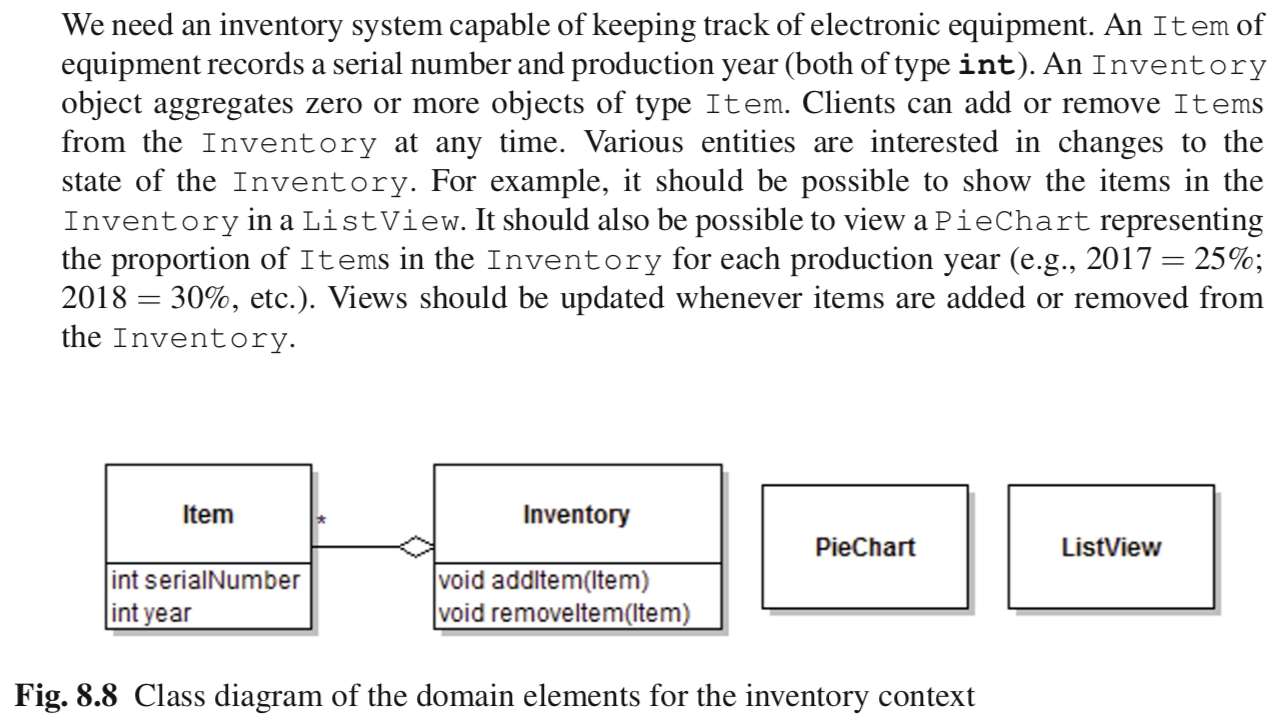
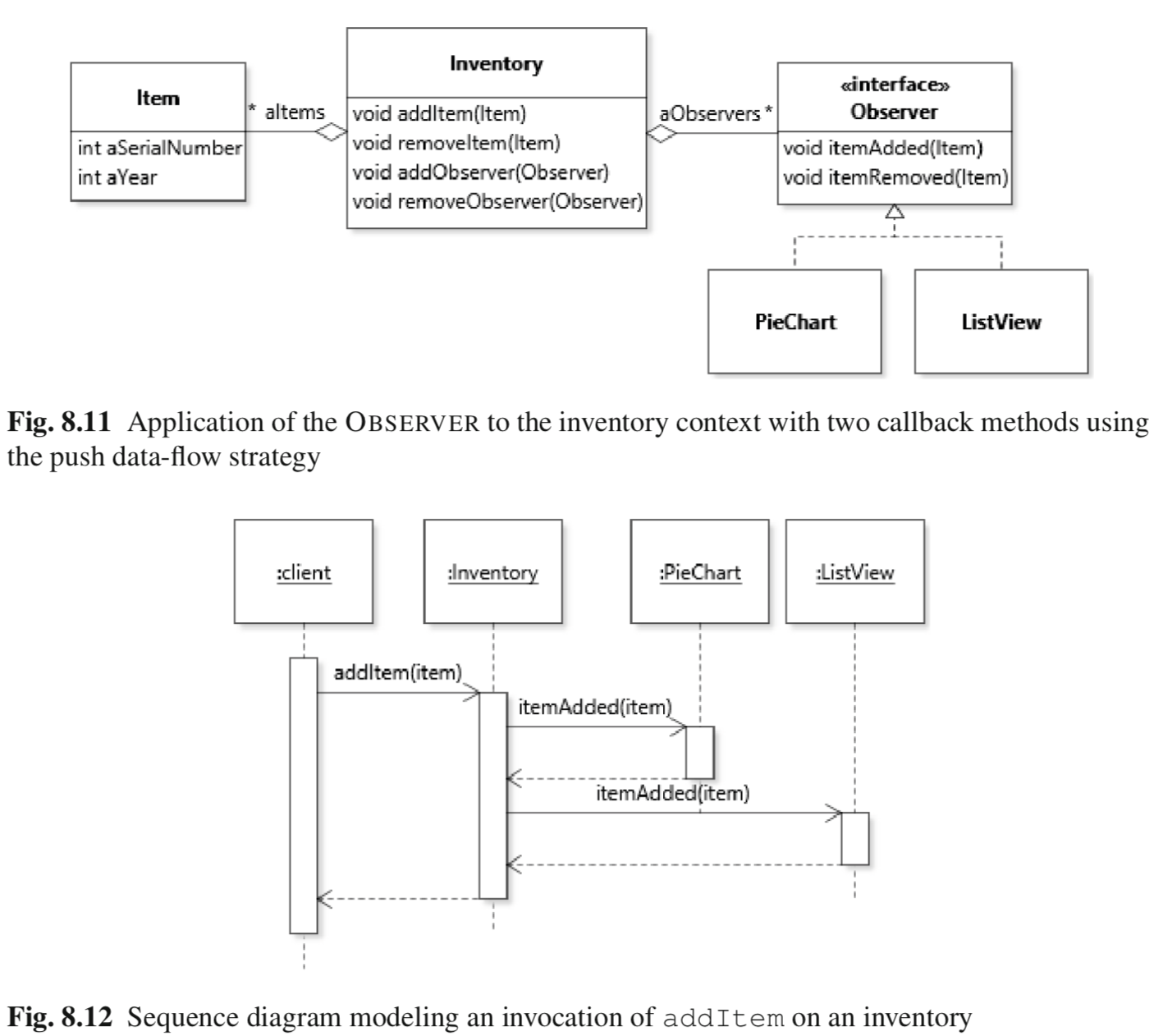


The trade-off for more flexibility is a slightly heavier interface for the Model class, because it now has to support two lists of observers with their corresponding registration methods.

[push data-flow strategy]



8.4 Applying the OBSERVER Design Pattern



8.5 Introduction to Graphical User Interface Development

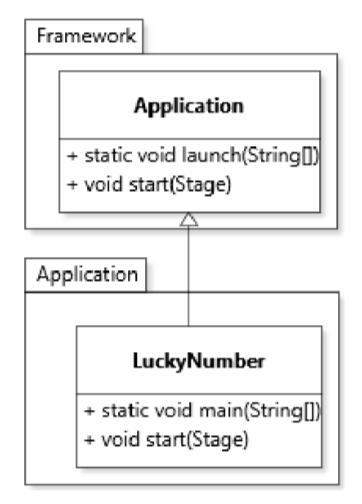
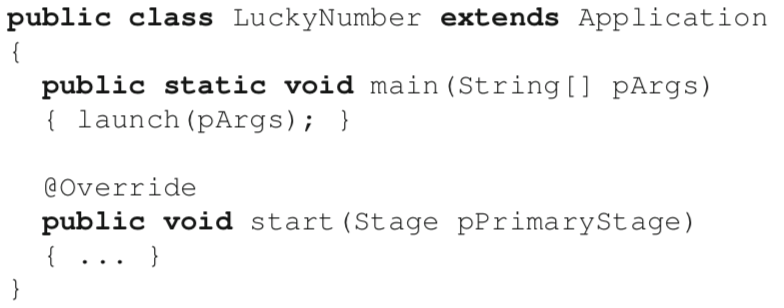
Conceptually, the code that makes up a GUI application is split into two parts:

* 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:



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.

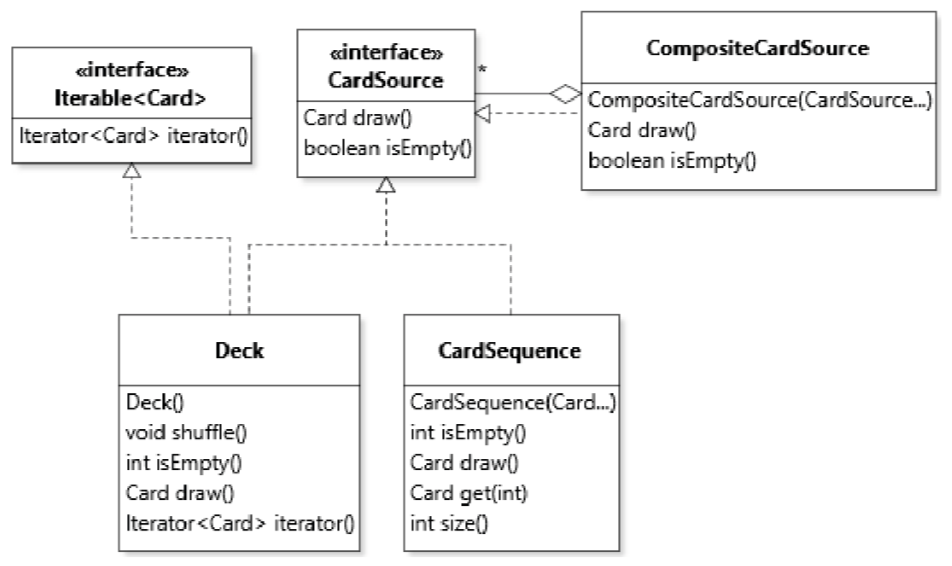
Once the framework is launched and displaying the desired component graph, its event loop will automatically map low-level system events to specific interactions with components in the graph (e.g., placing the mouse over a text box, clicking a button). In common GUI programming terminology, such interactions 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.

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.*



The constructor uses the vararg construct to accept an unspecified number of cards as argument. The mechanism is also employed by the constructor of CompositeCardSource.

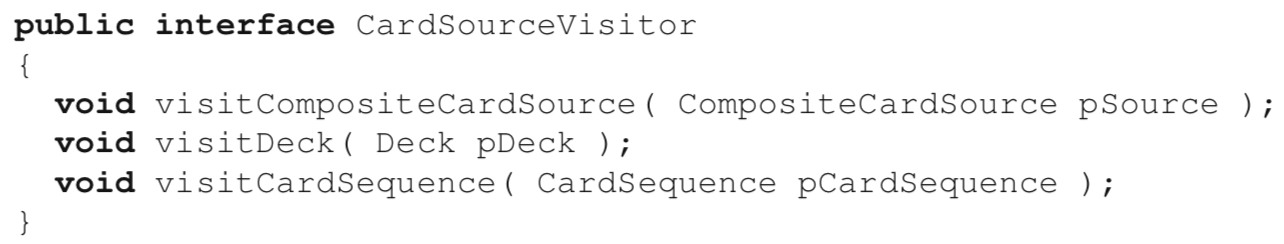
[drawbacks of adding methods to interfaces]

* 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:



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:



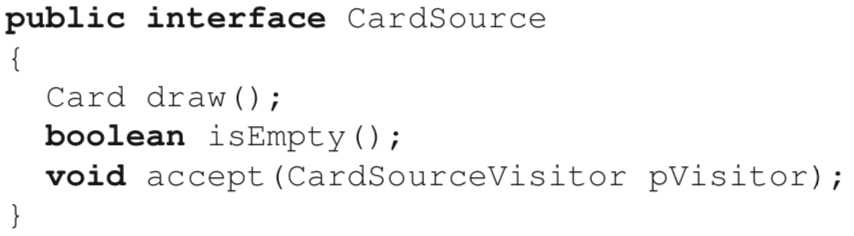
don’t require Deck and CardSequence to have the same interface. They can use whatever methods are available on the concrete type to implement the required behavior

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**

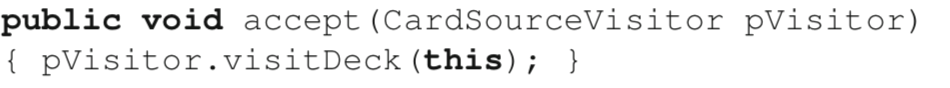
Although a concrete visitor separates a well-defined operation into its own class, it still needs to be integrated with the class hierarchy that defines the object graph on which the operation will be applied (henceforth simply referred to as the “class hierarchy”).

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:

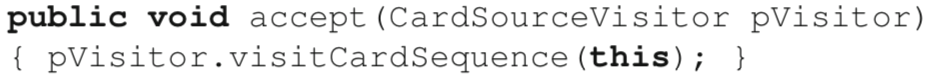


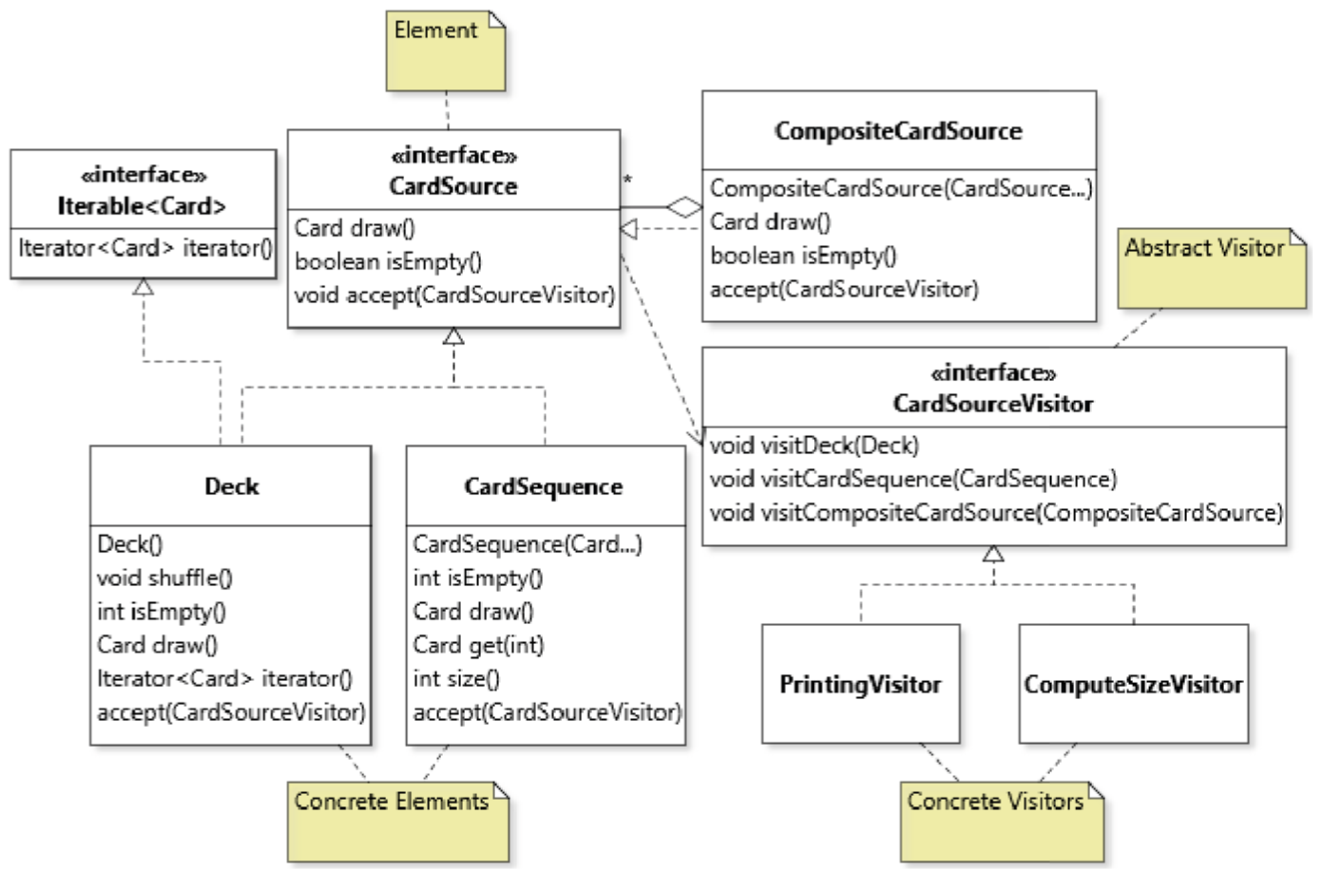
The implementation of accept by concrete types is where the integration really happens. This implementation follows a prescribed formula: to call the visit method for the type of the class that defines the accept method.

For example, the implementation of accept for class Deck is:

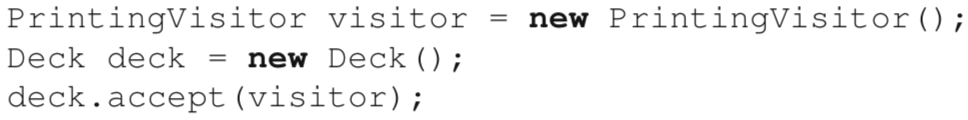


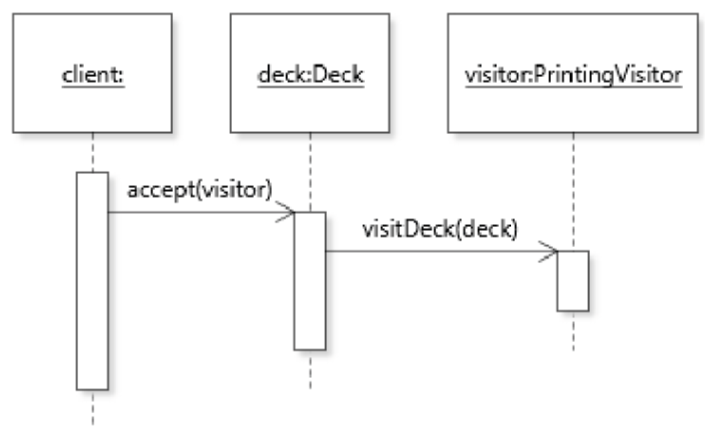
and the one for class CardSequence is:





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:



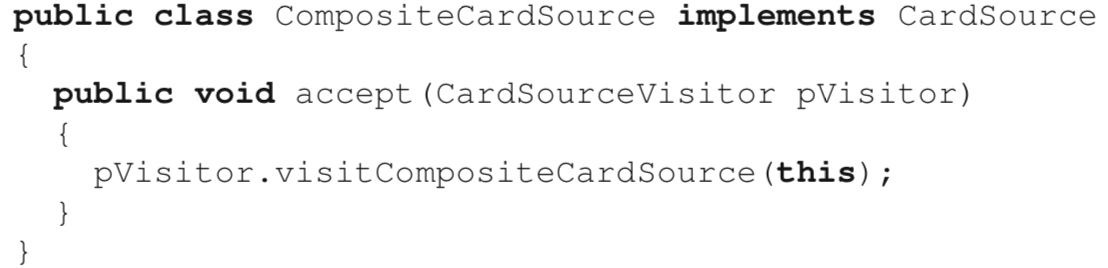


The accept method then calls back the appropriate method on the visitor. In this sequence, the visitDeck method qualifies as a callback method.

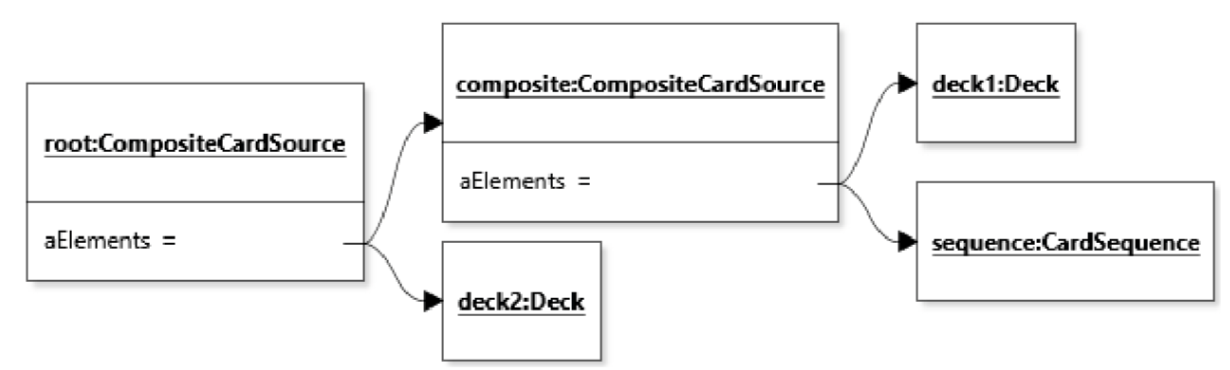
**Traversing the Object Graph**

Any object graph with more than one element will have an aggregate node as its root.

In our case this is CompositeCardSource



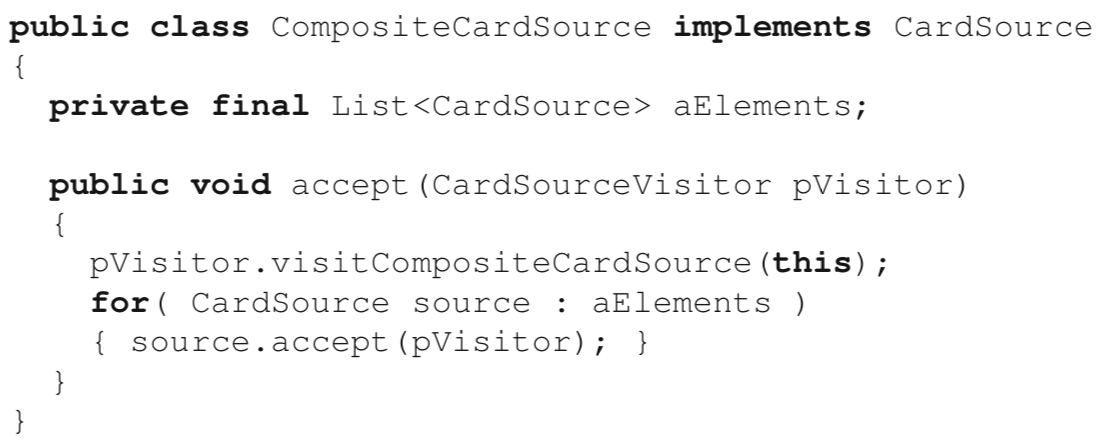
The two core ideas of the VISITOR pattern are to (1) enable the integration of an open-ended set of operations that (2) can be applied by traversing an object graph, often a recursive one. For the traversal aspect of the pattern to be applicable, at least one element type in the target hierarchy needs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



[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.

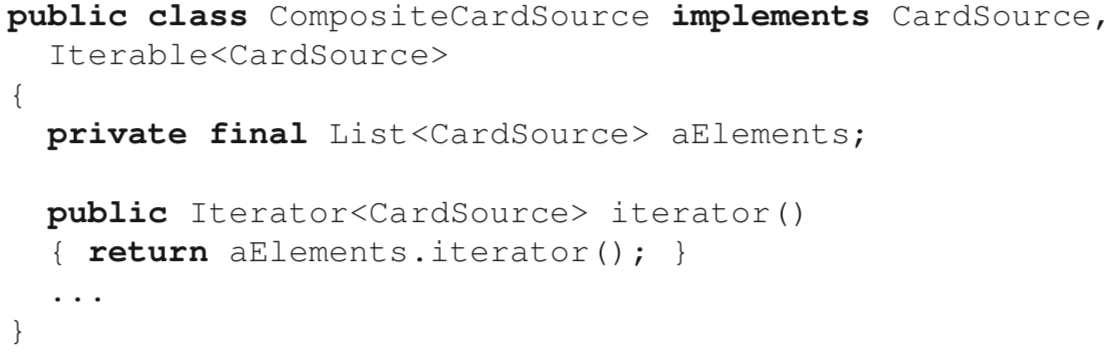
[disadvantage]

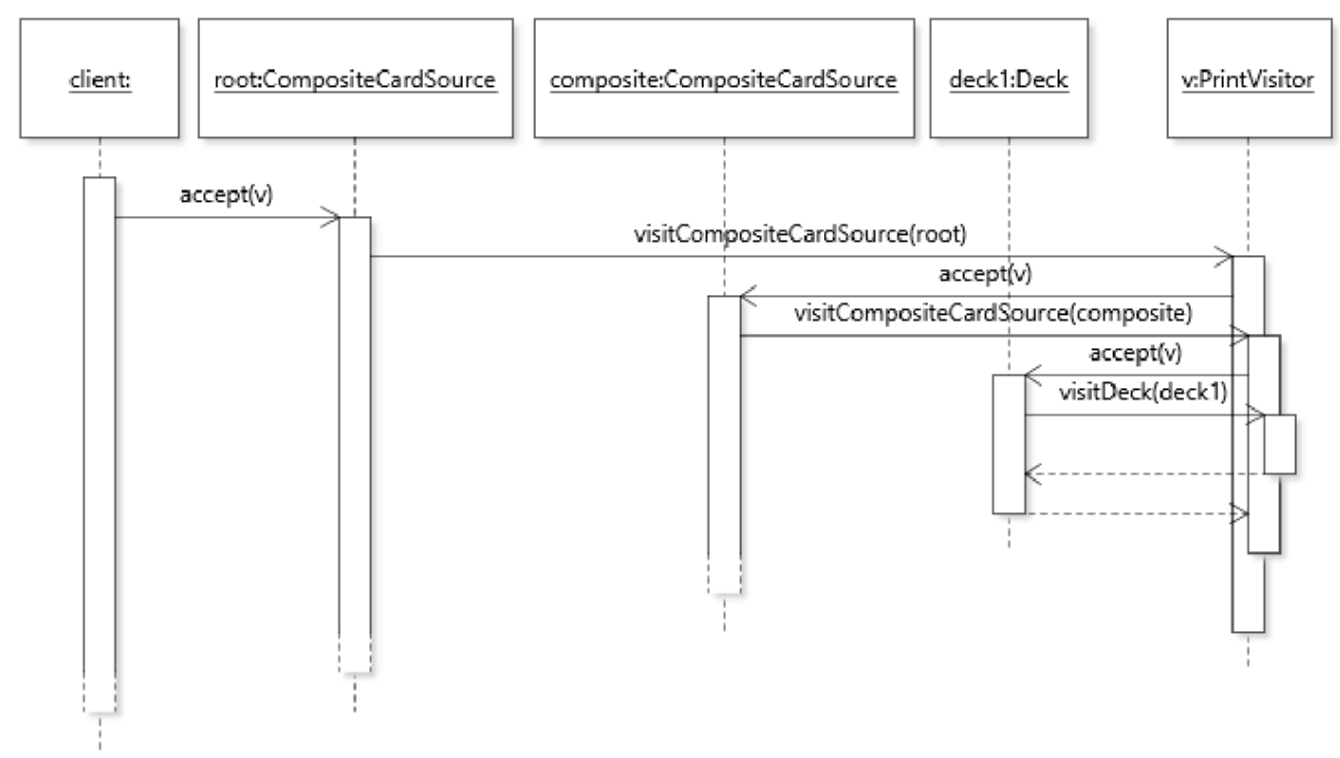
The traversal order is fixed in the sense that it cannot be adapted by different visitors.

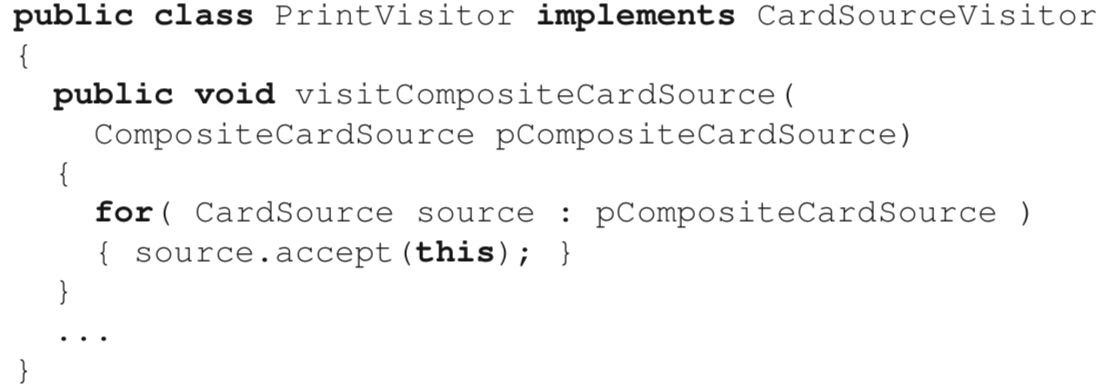


Second option Place this code in the visit methods that serve as callbacks for aggregate types.

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.





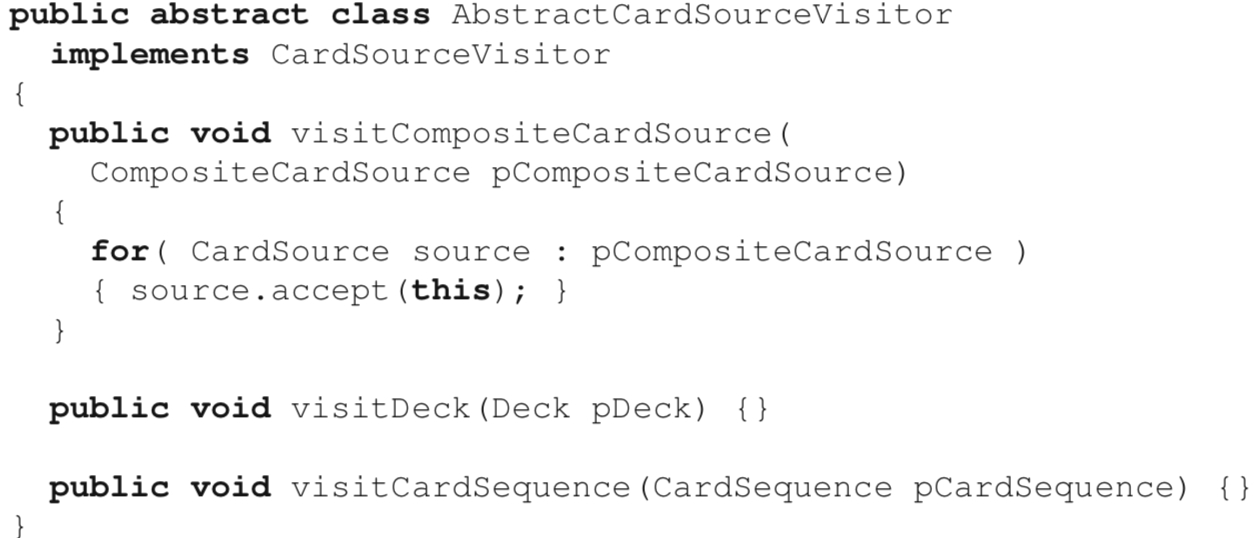


Conclusion

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**

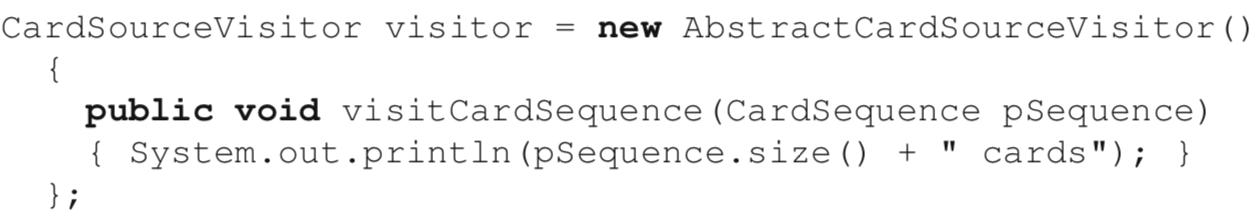
The question of where to place the traversal code brings up the issue of code DUPLI- CATED CODE† again. If we place the traversal code in the visit methods, and have more than one concrete visitor class, every class is bound to repeat the traversal code in its visit method. A common solution to alleviate 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:



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

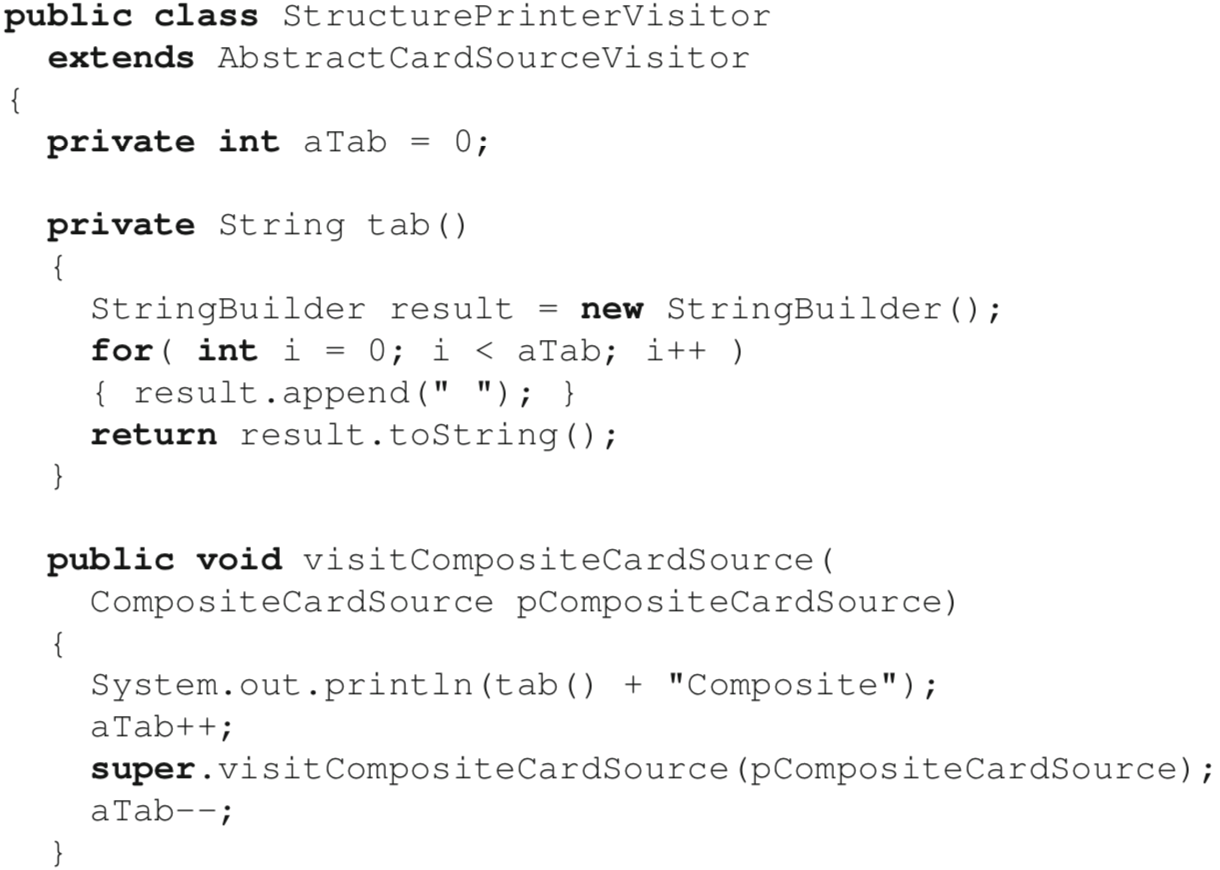
* First, I retained the interface. Because most concrete visitors would be implemented as sub-classes of AbstractCardSourceVisitor, one can wonder, why not simply use this abstract class to serve in the role of abstract visitor, and get rid of the interface? The general reason is that interfaces promote 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.

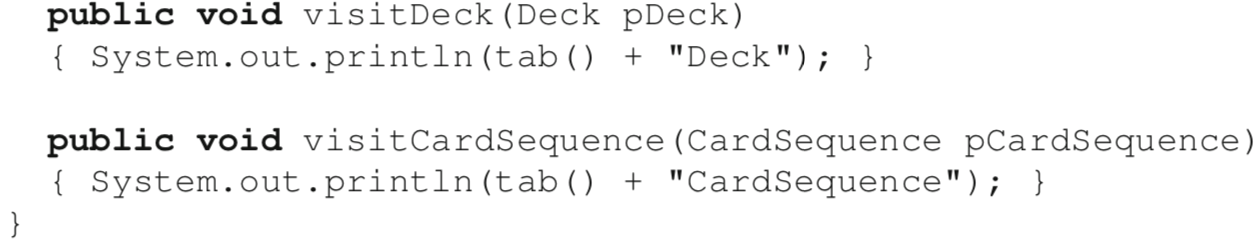
e.g. Print the number of cards in every CardSequence in a card source structure, and ignores the rest.

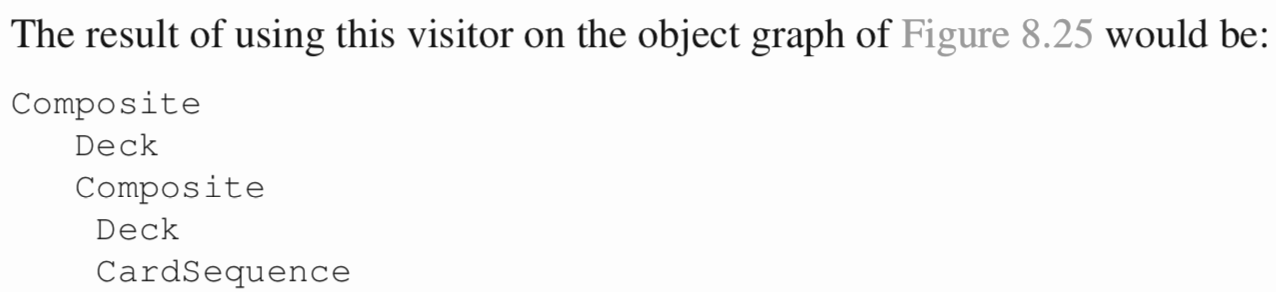


Because the class inherits the traversal code, card sequences aggregated within composite card sources will also be reached.

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







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.

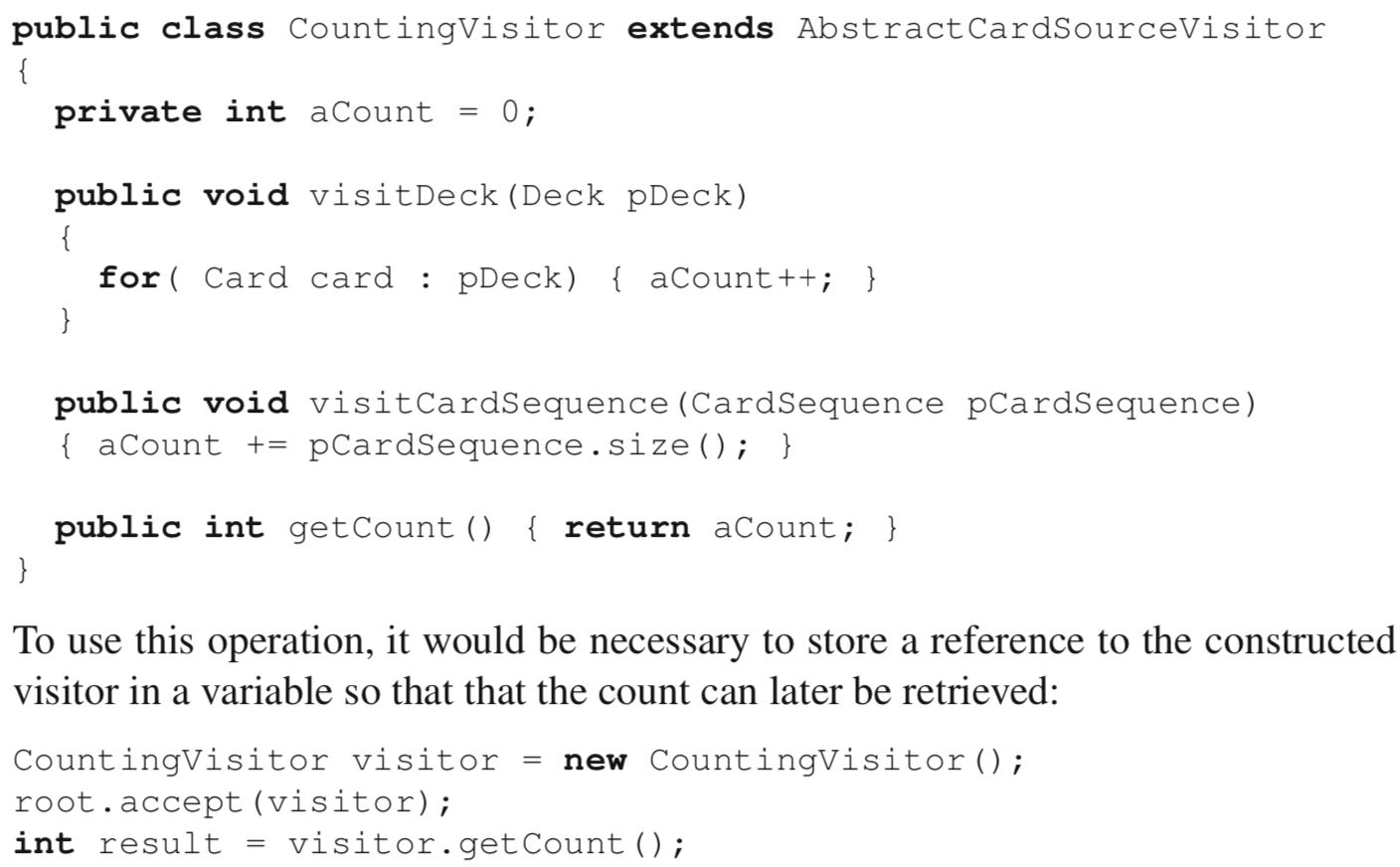
Second, the reuse of the traversal code through a super call. Here, the pre-order traversal implemented in the abstract visitor class is what we need. However, additional code is required when visiting a composite card source. To make this possible, visitCompositeCardSource is overridden to manage the indentation level, and a super call is made to trigger the traversal code at the appropriate point.

**Supporting Data Flow in Visitor Structures**

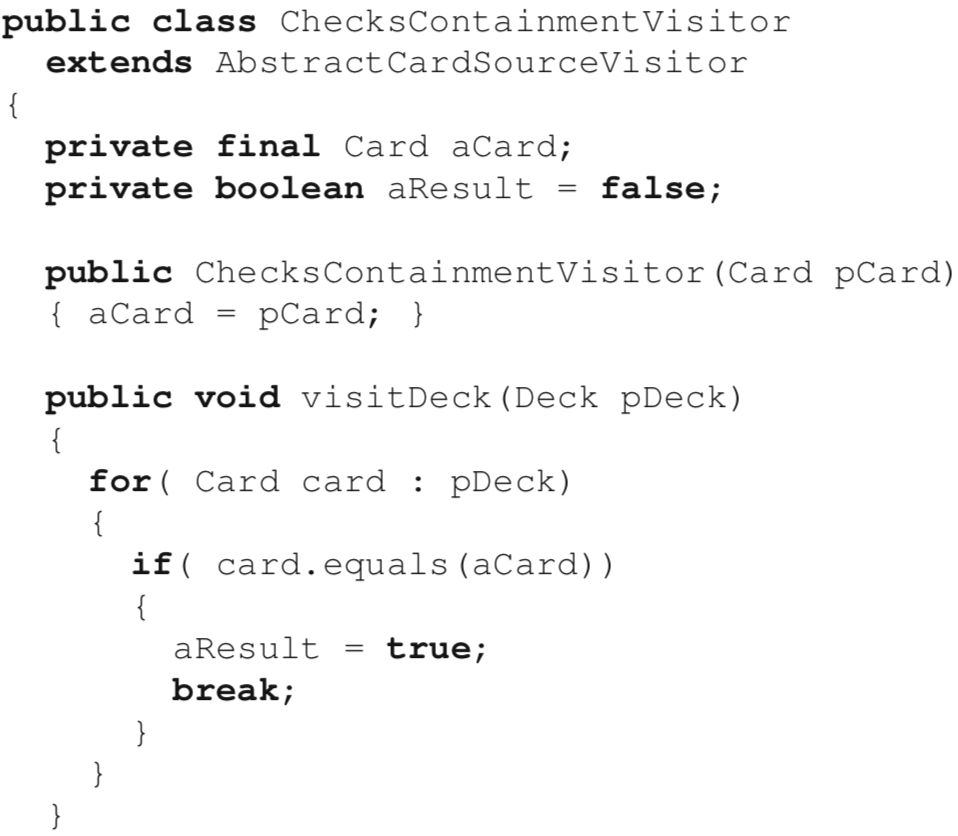
*To support a general and extensible mechanism for defining operations on an object graph, the pattern requires that no assumption be made about the nature of the input and output of operations.*

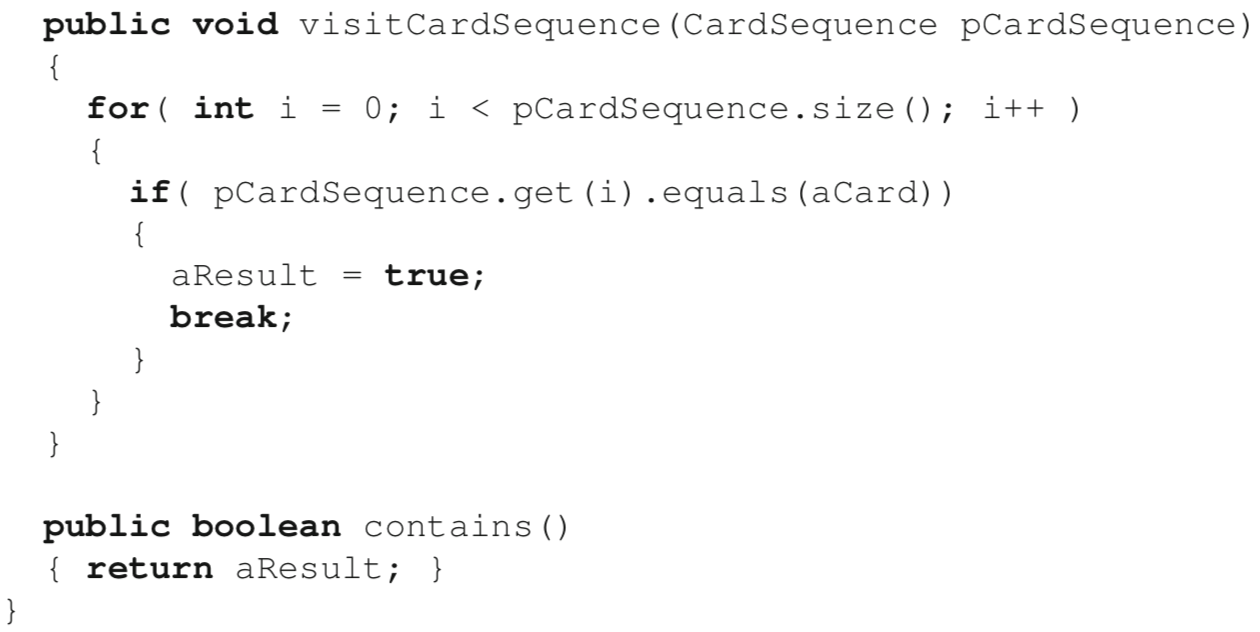
Data flow for VISITOR-based operations is thus implemented differently, by storing data within a visitor object. Input values can be provided when constructing a new visitor object and made accessible to the visit methods. Output values can be stored internally by visit methods during the traversal of 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:



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





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.

