16

ITERATOR

This pattern was previously described in GoF95.

DESCRIPTION

The Iterator pattern allows a client object to access the contents of a container in a sequential manner, without having any knowledge about the internal representation of its contents.

The term *container*, used above, can simply be defined as *a collection of data or objects*. The objects within the container could in turn be collections, making it a collection of collections. The Iterator pattern enables a client object to traverse through this collection of objects (or the container) without having the container to reveal how the data is stored internally.

To accomplish this, the Iterator pattern suggests that a `Container` object should be designed to provide a public interface in the form of an *Iterator* object for different client objects to access its contents. An `Iterator` object contains public methods to allow a client object to navigate through the list of objects within the container.

ITERATORS IN JAVA

One of the simplest iterators available in Java is the `java.sql.ResultSet` class, which is used to hold database records. This class offers a method `next()` for navigating along rows and a set of `getter` methods for column positioning.

Java also offers an interface `Enumeration` as part of the `java.util` package, which declares the methods listed in Table 16.1.

Table 16.1 Enumeration Methods

Method	*Return*	*Description*
`hasMoreElements()`	boolean	Checks if there are more elements in the collection
`nextElement()`	Object	Returns the next element in the collection

Table 16.2 Iterator Interface Methods

Method	*Return*	*Description*
`hasNext()`	boolean	Checks if there are more elements in the collection.
`next()`	Object	Returns the next element in the collection.
`remove()`	void	Removes from the collection, the last element returned by the iterator.

Concrete iterators can be built as implementers of the `Enumeration` interface by providing implementation for its methods.

In addition, the `java.util.Vector` class offers a method:

```
public final synchronized Enumeration elements()
```

that returns an enumeration of elements or objects. The returned `Enumeration` object works as an iterator for the `Vector` object. The Java `Enumeration` interface methods listed in Table 16.1 can be used on the returned `Enumeration` object to sequentially fetch elements stored in the `Vector` object.

Besides the `Enumeration` interface, Java also offers the `java.util.Iterator` interface. The `Iterator` interface declares three methods as in Table 16.2.

Similar to the `Enumeration` interface, concrete iterators can be built as implementers of the `java.util.Iterator` interface.

Though it is considered useful to employ existing Java iterator interfaces such as `Iterator` or `Enumeration`, it is not necessary to utilize one of these built-in Java interfaces to implement an iterator. One can design a custom iterator interface that is more suitable for an application need.

FILTERED ITERATORS

In the case of the `java.util.Vector` class, its iterator simply returns the next element in the collection. In addition to this basic behavior, an iterator may be implemented to do more than simply returning the next object in line. For instance, an iterator object can return a selected set of objects (instead of all objects) in a sequential order. This filtering can be based on some form of input from the client. These types of iterators are referred to as *filtered iterators*.

INTERNAL VERSUS EXTERNAL ITERATORS

An iterator can be designed either as an internal iterator or as an external iterator.

- **Internal iterators**
  – The collection itself offers methods to allow a client to visit different objects within the collection. For example, the `java.util.ResultSet` class contains the data and also offers methods such as `next()` to navigate through the item list.
  – There can be only one iterator on a collection at any given time.
  – The collection has to maintain or save the state of iteration.

- **External iterators**
  - The iteration functionality is separated from the collection and kept inside a different object referred to as an *iterator*. Usually, the collection itself returns an appropriate iterator object to the client depending on the client input. For example, the `java.util.Vector` class has its iterator defined in the form of a separate object of type `Enumeration`. This object is returned to a client object in response to the `elements()` method call.
  - There can be multiple iterators on a given collection at any given time.
  - The overhead involved in storing the state of iteration is not associated with the collection. It lies with the exclusive `Iterator` object.

EXAMPLE: INTERNAL ITERATOR

Let us build an application to display data from a file `Candidates.txt` containing details of different IT professionals who have offered their candidature for a job opening. For simplicity, let us consider only three attributes — name, current working location and certification type. As discussed in the preceding section "Internal versus External Iterators", in case of an internal iterator, the container (or the collection) is responsible for providing the interface for a client object to navigate through the container's contents.

Let us define a container class `AllCandidates` (Listing 16.1) that:

- Reads data from the data file as part of its constructor and stores the data in the form of a group of `Candidate` objects inside of an instance variable of `Vector` type.
- Implements the built-in `java.util.Iterator` interface and provides implementation for its methods as follows:
  - `hasNext()` – Checks to see if there are any more candidates in the collection.
  - `next()` – Returns the next candidate object, if any, from the collection. If there is none, it throws a `NoSuchElementException` exception.
  - `remove()` – Because the application does not deal with the candidate data deletion, this method implementation does nothing.

Figure 16.1 shows the class association of the example application using an internal iterator.

A client `SearchManager` can be designed to make use of the `AllCandidates` container to display different candidates data.

CLIENT/CONTAINER INTERACTION

The client `SearchManager` creates the necessary user interface for the display of the data (Figure 16.2). When a user clicks on the `Show All` button, it creates an instance of the container `AllCandidates`. As part of its constructor, the `AllCandidates` object reads the data file and stores it inside a `Vector` in the form of a group of `Candidate` objects. The client does not have to be aware of how the data is stored, in which form and other details. In other words, the client `SearchManager` only needs to know that the container `AllCandidates`

Listing 16.1 `AllCandidates` Class

```
public class AllCandidates implements Iterator {
  private Vector data;
  Enumeration ec;
  Candidate nextCandidate;
  public AllCandidates() {
    initialize();
    ec = data.elements();
  }
  private void initialize() {
    /*
      Get data from db.
    */
    data = new Vector();
    FileUtil util = new FileUtil();
    Vector dataLines = util.fileToVector("Candidates.txt");
    for (int i = 0; i < dataLines.size(); i++) {
      String str = (String) dataLines.elementAt(i);
      StringTokenizer st = new StringTokenizer(str, ”,");
      data.add(
        new Candidate(st.nextToken(), st.nextToken(),
                      st.nextToken()));
    }
  }
  public boolean hasNext() {
    boolean matchFound = false;
    nextCandidate = null;
    while (ec.hasMoreElements()) {
      Candidate tempObj = (Candidate) ec.nextElement();
      nextCandidate = tempObj;
      break;
    }
    return (nextCandidate != null);
  }
```

(continued)

Listing 16.1 `AllCandidates` Class (Continued)

```
      public Object next() {
        if (nextCandidate == null) {
          throw new NoSuchElementException();
        } else {
          return nextCandidate;
        }
      }
      public void remove() {};
    }
```

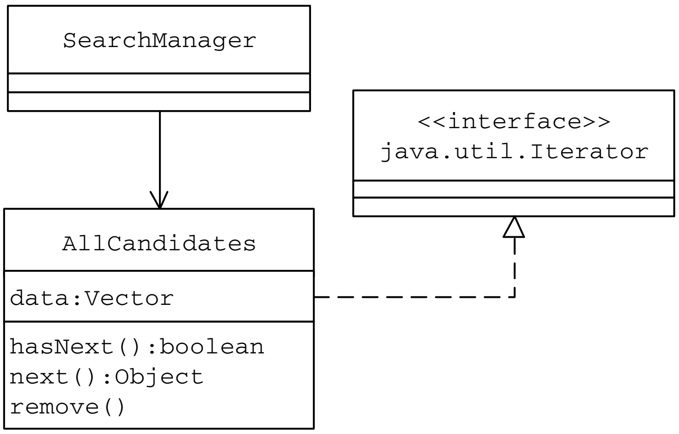


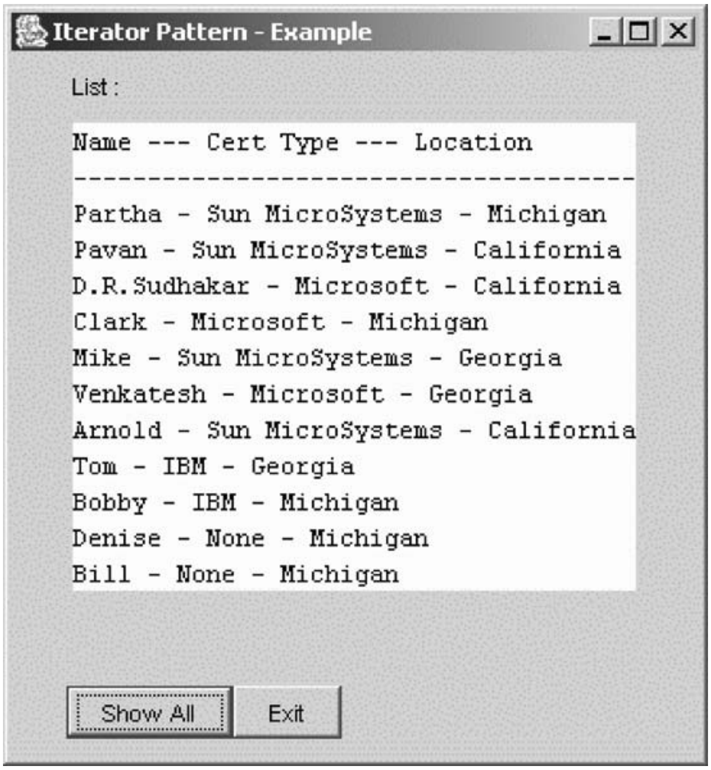
Figure 16.1 Internal Iterator: Class Association

functions as a data store for the candidate data in the form of a group of `Candidate` objects. It does not need to know how these objects are stored (in a `Vector` or `Hashmap`, etc.) inside the container.

For the client, the `AllCandidates` object functions both as a container and an iterator. It makes use of the `hasNext()` and the `next()` methods to retrieve different `Candidate` objects and displays them in the user interface (Figure 16.2).

```
                …
                …
      public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals(SearchManager.EXIT)) {
          System.exit(1);
        }
        if (e.getActionCommand().equals(SearchManager.SHOW_ALL)) {
```

```
      AllCandidates ac = new AllCandidates();
      String selectedCandidates =
        "Name — - Cert Type — - Location" + "\n" +
          " — — — — — — — — — — — — — — — — — — — ";
      while (ac.hasNext()) {
        Candidate c = (Candidate) ac.next();
        selectedCandidates = selectedCandidates + "\n" +
          c.getName() + " - " +
          c.getCertificationType() + " - " +
          c.getLocation();
      }
      manager.setSelectedCandidates(selectedCandidates);
    }
  }
          …
          …
```



```
Name --- Cert Type --- Location
---------------------------------------
Partha - Sun MicroSystems - Michigan
Pavan - Sun MicroSystems - California
D.R.Sudhakar - Microsoft - California
Clark - Microsoft - Michigan
Mike - Sun MicroSystems - Georgia
Venkatesh - Microsoft - Georgia
Arnold - Sun MicroSystems - California
Tom - IBM - Georgia
Bobby - IBM - Michigan
Denise - None - Michigan
Bill - None - Michigan
```

Figure 16.2 Client User Interface: Results Display

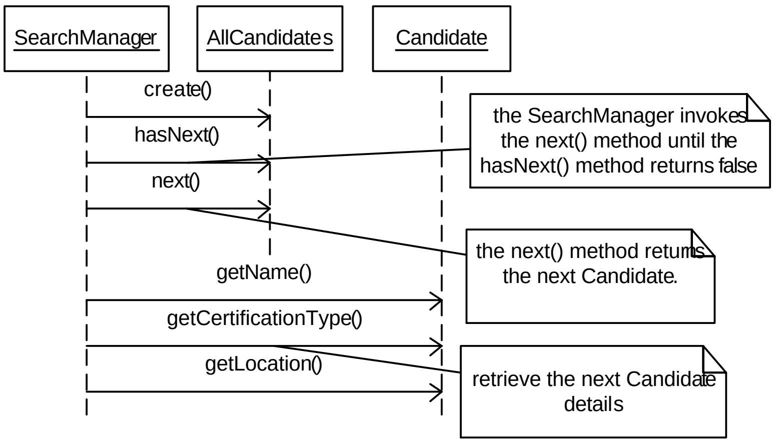

SearchManager
AllCandidates
Candidate
create()
hasNext()
next()
getName()
getCertificationType()
getLocation()
the SearchManager invokes the next() method until the hasNext() method returns false
the next() method returns the next Candidate.
retrieve the next Candidate details

Figure 16.3 Internal Iterator: Message Flow

Figure 16.3 shows the application message flow when the user clicks on the `Show All` button.

EXAMPLE: EXTERNAL FILTERED ITERATOR

Let us enhance the example application to allow a user to filter candidates by the type of certification they have. This enhancement can be designed using an external filtered iterator.

In the case of an external iterator, the implementation is decoupled from the container and is kept inside a separate iterator class.

Let us design an external iterator class `CertifiedCandidates` (Listing 16.2) as an implementer of the built-in `java.util.Iterator` interface. As part of its constructor, the `CertifiedCandidates` iterator accepts a certification type and an instance of `AllCandidates` as input. It works as a filtered iterator for the candidate data contained in the `AllCandidates` container and returns the group of candidates with the specified certification type in a sequential manner. It implements the `java.util.Iterator` methods as follows:

- `hasNext()` – Checks to see if there are any more candidates with the specified certification type.
- `next()` – Returns the next candidate, if any, with the specified certification type. If there is none, it throws a `NoSuchElementException` exception. Ideally, a client would invoke the `next()` method only if a prior call to the `hasNext()` method returns true.
- `remove()` – Because the scope of the example application does not deal with deleting the profile of a candidate, this method implementation does nothing.

Besides the external iterator definition, as part of the new design, the container `AllCandidates` class needs to be modified (Listing 16.3) so that:

Listing 16.2 `CertifiedCandidates` Class

```
public class CertifiedCandidates implements Iterator {
  private Vector v;
  AllCandidates ac;
  String certificationType;
  Candidate nextCandidate;
  Enumeration ec;
  public CertifiedCandidates(AllCandidates inp_ac,
      String certType) {
    ac = inp_ac;
    certificationType = certType;
    ec = inp_ac.getAllCandidates();
  }
  public boolean hasNext() {
    boolean matchFound = false;
    while (ec.hasMoreElements()) {
      Candidate tempObj = (Candidate) ec.nextElement();
      if (tempObj.getCertificationType().equals(
            certificationType)) {
        matchFound = true;
        nextCandidate = tempObj;
        break;
      }
    }
    if (matchFound == true) {
    } else {
      nextCandidate = null;
    }
    return matchFound;
  }
  public Object next() {
    if (nextCandidate == null) {
      throw new NoSuchElementException();
    } else {
      return nextCandidate;
    }
  }
  public void remove() {};
}
```

Listing 16.3 `AllCandidates` Class: Modified

```
public class AllCandidates {
  private Vector data;
  public AllCandidates() {
    initialize();
  }
  private void initialize() {
    /*
      Get data from db.
    */
    data = new Vector();
    FileUtil util = new FileUtil();
    Vector dataLines = util.fileToVector("Candidates.txt");
    for (int i = 0; i < dataLines.size(); i++) {
      String str = (String) dataLines.elementAt(i);
      StringTokenizer st = new StringTokenizer(str, ”,");
      data.add(
        new Candidate(st.nextToken(), st.nextToken(),
                      st.nextToken()));
    }
  }
  public Enumeration getAllCandidates() {
    return data.elements();
  }
  public Iterator getCertifiedCandidates(String type) {
    return new CertifiedCandidates(this, type);
  }
}
```

- It is still responsible for reading the data file and carrying the data inside it, in the form of `Candidate` objects.
- It offers a public method `getCertifiedCandidates(String type)`. This method creates and returns an iterator as an object of type `java.util.Iterator`. The client `SearchManager` can use this method when it wants to filter the candidate data by a specific certification type.

While creating an instance of the iterator `CertifiedCandidates`, the container `AllCandidates` sends itself as an argument to the iterator. The iterator uses this instance to access the data stored inside the `AllCandidates` container.

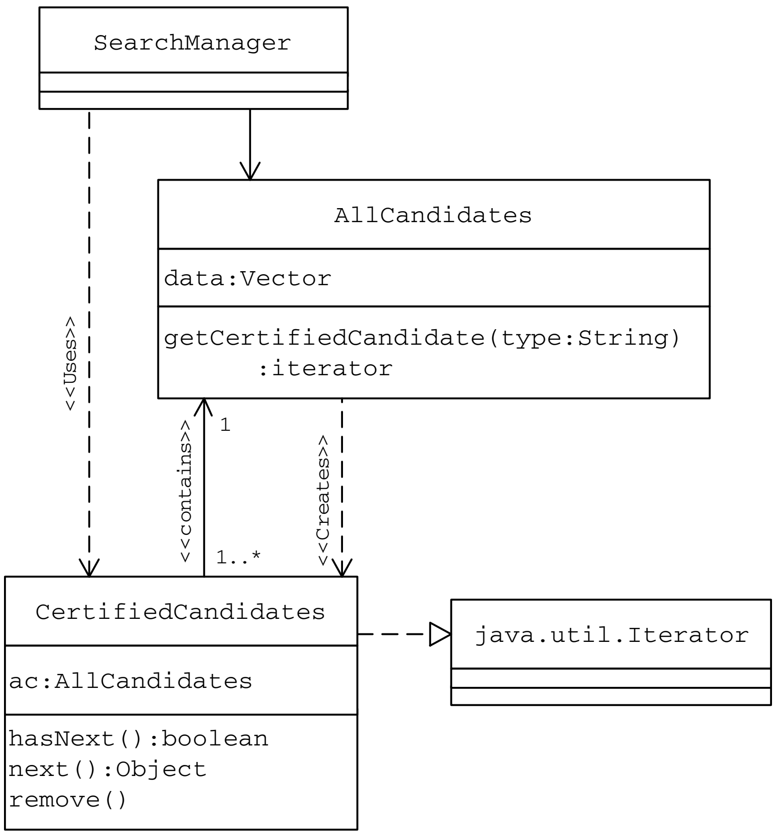
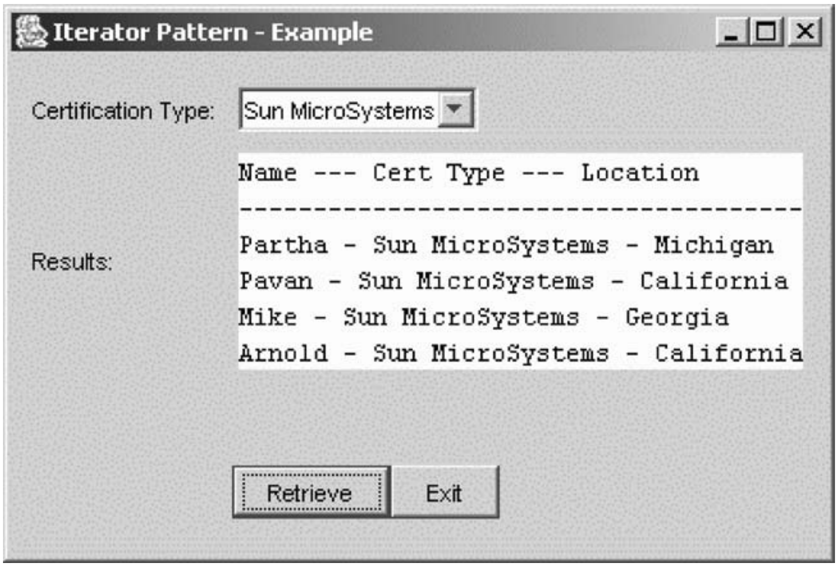


Figure 16.4 External Iterator: Class Association

The class diagram in Figure 16.4 shows the class association in the example application using an external filtered iterator.

The client `SearchManager` creates the necessary user interface to allow a user to select a certification type and to display the data (Figure 16.5). When a user selects a certification type and clicks on the `Retrieve` button, the `Search-Manager`:

- Creates an instance of the container `AllCandidates`. As part of its constructor, the `AllCandidates` object reads the data file and stores the data inside an instance variable data of type `Vector`. The client does not have to be aware of the data format or how the data is stored.
- Invokes the `getCertifiedCandidates(String type)` method on the `AllCandidates` container object by passing the selected certification type as an argument. The `getCertifiedCandidates` method creates an instance of the `CertifiedCandidates` class and returns it as an object of type `java.util.Iterator`.



```
Name --- Cert Type --- Location
--------------------------------------
Partha - Sun MicroSystems - Michigan
Pavan - Sun MicroSystems - California
Mike - Sun MicroSystems - Georgia
Arnold - Sun MicroSystems - California
```

Figure 16.5 Client User Interface: Results Display

Once the `Iterator` object is received, the client `SearchManager` makes use of the `hasNext()` and the `next()` methods to retrieve the matching `Candidate` objects and displays them in the user interface (Figure 16.5).

```
              …
              …
  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals(SearchManager.EXIT)) {
      System.exit(1);
    }
    if (e.getActionCommand().equals(
          SearchManager.GET_CANDIDATES)) {
      String selection = manager.getCertificationType();
      AllCandidates ac = new AllCandidates();
      Iterator certCandidates =
        ac.getCertifiedCandidates(selection);
      String selectedCandidates =
        "Name — - Cert Type — - Location" + "\n" +
          " — — — — — — — — — — — — — — — — — — — — ";
      while (certCandidates.hasNext()) {
        Candidate c = (Candidate) certCandidates.next();
        selectedCandidates = selectedCandidates + "\n" +
        c.getName() + " - " + c.getCertificationType() +
          " - " + c.getLocation();
      }
      manager.setSelectedCandidates(selectedCandidates);
    }
  }
              …
              …
```

The sequence diagram in Figure 16.6 shows the message flow when the user clicks on the `Retrieve` button.

In the case of both iterators, the client `SearchManager` does not contain any implementation that is tied to the way data is stored inside the container. All such implementation is completely moved out of the client to either the container (internal iterator) or to the iterator object (external iterator). The resulting design protects the client `SearchManager` from any changes to the way the data is maintained inside the container. For instance, if the internal representation of the data is changed so that the data is stored in an `array` or in some other form instead of a `Vector`, no changes are required to the client `SearchManager` implementation.

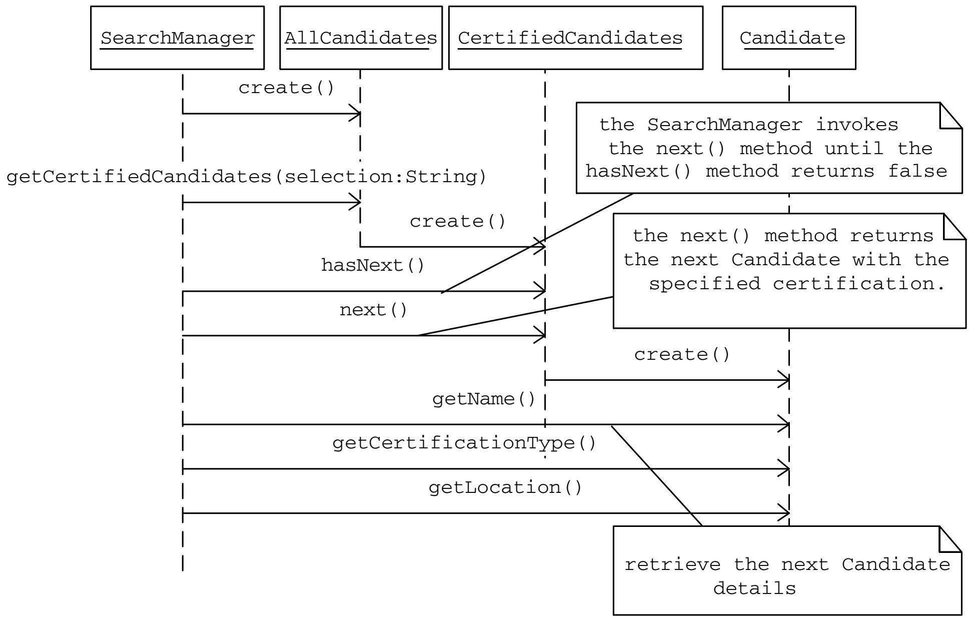

SearchManager
AllCandidates
CertifiedCandidates
Candidate
create()
the SearchManager invokes the next() method until the hasNext() method returns false
getCertifiedCandidates(selection:String)
create()
the next() method returns the next Candidate with the specified certification.
hasNext()
next()
create()
getName()
getCertificationType()
getLocation()
retrieve the next Candidate details

Figure 16.6 Message Flow When the External Iterator Is in Use

PRACTICE QUESTIONS

1. Design and implement a new filtered external iterator to filter the list of candidates by location and integrate it into the example application.
2. Consider the following author details XML data file contents.

```
<Authors>
  <Author>
    <Name>Auth_1</Name>
    <Books>
      <Book>
        <Title>B1</Title>
      </Book>
      <Book>
        <Title>B2</Title>
      </Book>
    <Books>
  </Author>
  <Author>
    <Name>Auth_2</Name>
      <Books>
        <Book>
          <Title>B3</Title>
        </Book>
        <Book>
          <Title>B4</Title>
        </Book>
    <Books>
  </Author>
</Authors>
```

Design an application to go through this list of authors and their books using the components listed in Table 16.3.
Design a client to use these two iterators together to access each author, the author's books and display them in a desired format.

Table 16.3 Application Components

Name	*Role*	*Responsibility*
`AuthorCollection`	Container	This class is responsible for reading from the physical XML file and holding the data. Offers methods to create two external iterators — `AuthorIterator` and `BookIterator` — on its data.
`AuthorIterator`	Iterator	An external iterator that returns all authors one by one in response to its `next()` method call.
`BookIterator`	Iterator	An external filtered iterator that returns all books written by a specified author one by one in response to its `next()` method call.