32

MEMENTO

This pattern was previously described in GoF95.

DESCRIPTION

The state of an object can be defined as the values of its properties or attributes at any given point of time. The Memento pattern is useful for designing a mechanism to capture and store the state of an object so that subsequently, when needed, the object can be put back to this (previous) state. This is more like an undo operation. The Memento pattern can be used to accomplish this without exposing the object's internal structure. The object whose state needs to be captured is referred to as the *originator*. When a client wants to save the state of the originator, it requests the current state from the originator. The originator stores all those attributes that are required for restoring its state in a separate object referred to as a *Memento* and returns it to the client. Thus a Memento can be viewed as an object that contains the internal state of another object, at a given point of time. A `Memento` object must hide the originator variable values from all objects except the originator. In other words, it should protect its internal state against access by objects other than the originator. Towards this end, a `Memento` should be designed to provide restricted access to other objects while the originator is allowed to access its internal state.

When the client wants to restore the originator back to its previous state, it simply passes the memento back to the originator. The originator uses the state information contained in the memento and puts itself back to the state stored in the `Memento` object.

EXAMPLE

Data conversion is almost always an integral part of any application that involves converting a legacy system to newer technologies. Let us consider one such application where customer data needs to be moved from a flat file to a relational database. The process validates every customer record before sending it to the database.

In reality, a customer record would contain many attributes, but for simplicity, let us consider only three attributes — first name, last name and the credit card number. The validations are also kept very simple. A customer record is considered as valid if the last name is not blank and the credit card number is composed of only digits (0 through 9). Whenever an invalid customer record is found, the process stops and prompts the user to correct the data and restart the process. At this point, the state of the data conversion process is saved inside a `Memento` object. When the user restarts the process, the conversion process state is restored from the `Memento` object and the process resumes from where it stopped, instead of starting from the beginning of the source data file. In general, a `Memento` object can be stored either in the memory or to persistent media. In this application, the state needs to be saved even after the application has been terminated and needs to be restored when the application is run subsequently. Hence, storing the `Memento` in the memory is not an option in this case. The `Memento` needs to be stored to persistent media instead.

Instead of storing valid customer records in a relational database, the application generates a text file consisting of SQL insert statements, which can be executed to insert data into any relational database.

Let us design different components required for this process to work.

`DataConverter` (Originator)

The `DataConverter` class (Figure 32.1 and Listing 32.1) is the implementer of the data conversion process.

`ID`

The instance variable `ID` constitutes the state of the `DataConverter`. It represents the customer ID of the last successfully processed customer record.

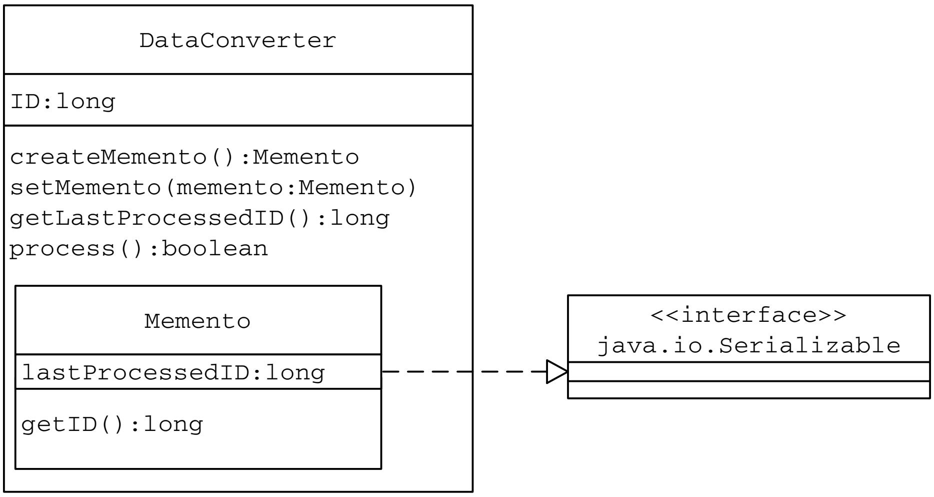

Figure 32.1 `DataConverter` Class: The Originator

Listing 32.1 `DataConverter` Class

```
public class DataConverter {
  public static final String DATA_FILE = "Data.txt";
  public static final String OUTPUT_FILE = "SQL.txt";
  private long ID = 0;
  public Memento createMemento() {
    return (new Memento(ID));
  }
  public void setMemento(Memento memento) {
    if (memento != null)
      ID = memento.getID();
  }
  public long getLastProcessedID() {
    return ID;
  }
  public void setLastProcessedID(long lastID) {
    ID = lastID;
  }
  public boolean process() {
    boolean success = true;
    String inputLine = "";
    long currID = 0;
    try {
      File inFile = new File(DATA_FILE);
      BufferedReader br = new BufferedReader(
                            new InputStreamReader(
                              new FileInputStream(inFile)));
      long lastID = getLastProcessedID();
      while ((inputLine = br.readLine()) != null) {
        StringTokenizer st =
          new StringTokenizer(inputLine, ″,");
        String strID = st.nextToken();
        currID = new Long(strID).longValue();
        if (lastID < currID) {
```

(continued)

Listing 32.1 `DataConverter` Class (Continued)

```
          Customer c =
            new Customer(strID, st.nextToken(),
                            st.nextToken(), st.nextToken());
          if (!(c.isValid())) {
            success = false;
            break;
          }
          ID = new Long(strID).longValue();
          FileUtil util = new FileUtil();
          util.writeToFile(OUTPUT_FILE, c.getSQL(),
                              true, true);
          }
        }
        br.close();
      }//Try
      catch (Exception ex) {
        System.out.println(" An error has occurred " +
                              ex.getMessage());
        System.exit(1);
      }
      if (success == false) {
        System.out.println("An error has occurred at ID=" +
                              currID);
        System.out.println("Data Record=" + inputLine);
        return false;
      }
      return true;
    }
    class Memento implements java.io.Serializable {
      private long lastProcessedID;
      private Memento(long ID) {
        lastProcessedID = ID;
      }
      private long getID() {
        return lastProcessedID;
      }
    }//end of class
  }//end of class
```

Memento

The Memento class is defined as an inner class within the DataConverter. The Memento is defined with its constructor and other methods as private.

In Java, a class can access the private members of its inner classes.

The DataConverter will be able to access these methods while they remain inaccessible to other objects. Because the state of the DataConverter needs to be preserved even after the application ends, the Memento object needs to be serialized to a file. Hence the Memento is designed to implement the java.io.Serializable interface to identify itself as a Serializable class.

In Java, a Serializable class must:

- Explicitly specify nonserializable attributes using the *transient* keyword
- Implement the java.io.Serializable interface
- Have access to the first zero argument constructor of its first non-Serializable super class

---

process

The process method reads from the source data file, validates the customer data using a Customer helper class. For every valid customer record, a corresponding SQL insert statement is written to the output file. When a customer record with invalid data is encountered, the data conversion process stops.

createMemento

As the method name suggests, this method is responsible for the creation of the Memento object. It stores the DataConverter current state inside a Memento instance and returns it.

setMemento

Retrieves the state information from the input Memento object and resets the DataConverter back to this state.

DCClient (Client)

The client DCClient (Listing 32.2) first instantiates the DataConverter and starts the data conversion process by invoking the process method on this DataConverter instance. If the process method returns without processing the entire source data file due to invalid customer data, it invokes the create-Memento method on the DataConverter to capture its current state. The createMemento method returns a Memento object (See createMemento

Listing 32.2 `DCClient` Class

```
public class DCClient {
  public static void main(String[] args) {
    MementoHandler objMementoHandler = new MementoHandler();
    DataConverter objConverter = new DataConverter();
    objConverter.setMemento(objMementoHandler.getMemento());
    if (!(objConverter.process())) {
      System.out.println("Description: Invalid data - " +
                         "Process Stopped");
      System.out.println("Please correct the Data and " +
                         "Run the Application Again");
      objMementoHandler.setMemento(
        objConverter.createMemento());
    }
  }
}
```

method description above). The client `DCClient` uses a helper `MementoHandler` object to serialize this `Memento` instance to a file.

Once the data is corrected and the client `DCClient` is run again:

- The client `DCClient` invokes the `getMemento` method on the `MementoHandler` requesting it for the stored `Memento` object.
- The `MementoHandler` deserializes the previously serialized `Memento` object from the file and returns it to the client.
- The client passes it to the `DataConverter` as an argument to its `setMemento` method. The `DataConverter` puts itself back to the state stored in the memento and resumes with the data conversion process from where it stopped during the previous run.

MementoHandler

The `MementoHandler` (Listing 32.3) contains an object reference of `Memento` type. It is passed as a `Memento` instance by the client `DCClient`.

As discussed above, whenever the data conversion process returns without processing the entire source data file, the client captures the `DataConverter` state in a Memento and the application ends. For this Memento to be available during the next run of the application, it must be saved to persistent media. This involves object serialization. Also during the subsequent run if the `DataConverter` is to be put back to its previous state, this Memento needs to be reconstructed. This involves object deserialization. These details of Memento handling are maintained inside the `MementoHandler` class, freeing all clients

Listing 32.3 `MementoHandler` Class

```
public class MementoHandler {
  public static final String ID_FILE = "ID.txt";
  private DataConverter.Memento objMemento = null;
  public DataConverter.Memento getMemento() {
    ObjectInputStream objStream = null;
    FileUtil util = new FileUtil();
    if (util.isFileExists(ID_FILE)) {
      //read the object from the file
      try {
        objStream = new ObjectInputStream(
                      new FileInputStream(new File(ID_FILE)));
        objMemento = (DataConverter.Memento)
                       objStream.readObject();
        objStream.close();
      } catch (Exception e) {
        System.out.println("Error Reading Memento");
        System.exit(1);
      }
      //delete the old memento
      util.deleteFile(ID_FILE);
    }
    return objMemento;
  }
  public void setMemento(DataConverter.Memento memento) {
    ObjectOutputStream objStream = null;
    //write the object to the file
    try {
      objStream = new ObjectOutputStream(
                    new FileOutputStream(new File(ID_FILE)));
      objStream.writeObject(memento);
      objStream.close();
    } catch (Exception e) {
      System.out.println("Error Writing Memento");
      System.exit(1);
    }
  }
}//end of class
```

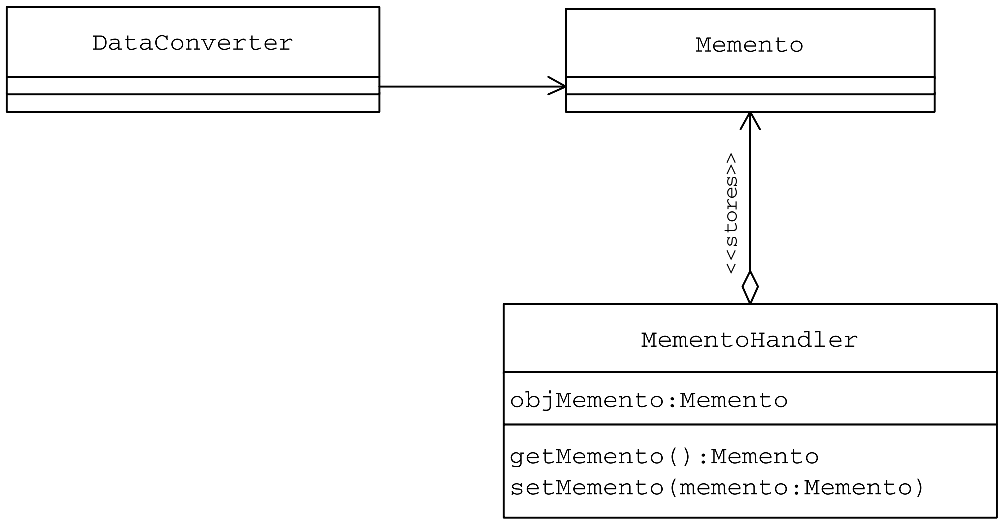

Figure 32.2 Data Conversion Application: Class Association

(that deal with the `DataConverter` and the associated `Memento` object) from having to deal with these details.

This also makes it easy to change the way the Memento is saved. For example, if the Memento needs to be saved to a database instead of a file, changes need to be made only to the `MementoHandler`, without having to alter the implementation of any client class that works with the Memento.

Figure 32.2 shows the association between different classes in the example data conversion application.

Figure 32.3 shows the application message flow.

PRACTICE QUESTIONS

1. Design and implement a Java applet that allows users to design a customized wedding gown. A preview of the dress should be displayed with default settings. Users should be able to select from a set of different neck and sleeve types. After every selection, the preview image should be updated. Users should be able to undo a selection to go back to their previous selection and the preview should get updated accordingly. Apply the Memento pattern in designing the undo operation.
2. Consider a simple shopping cart application that remembers shopping cart contents even after a user has logged out. Next time, when the user logs onto the Web site, the shopping cart should be shown with previously selected items and the user should be allowed to continue to shop in the new session. Identify how the Memento pattern can be used in preserving and restoring the state of an unfinished order.

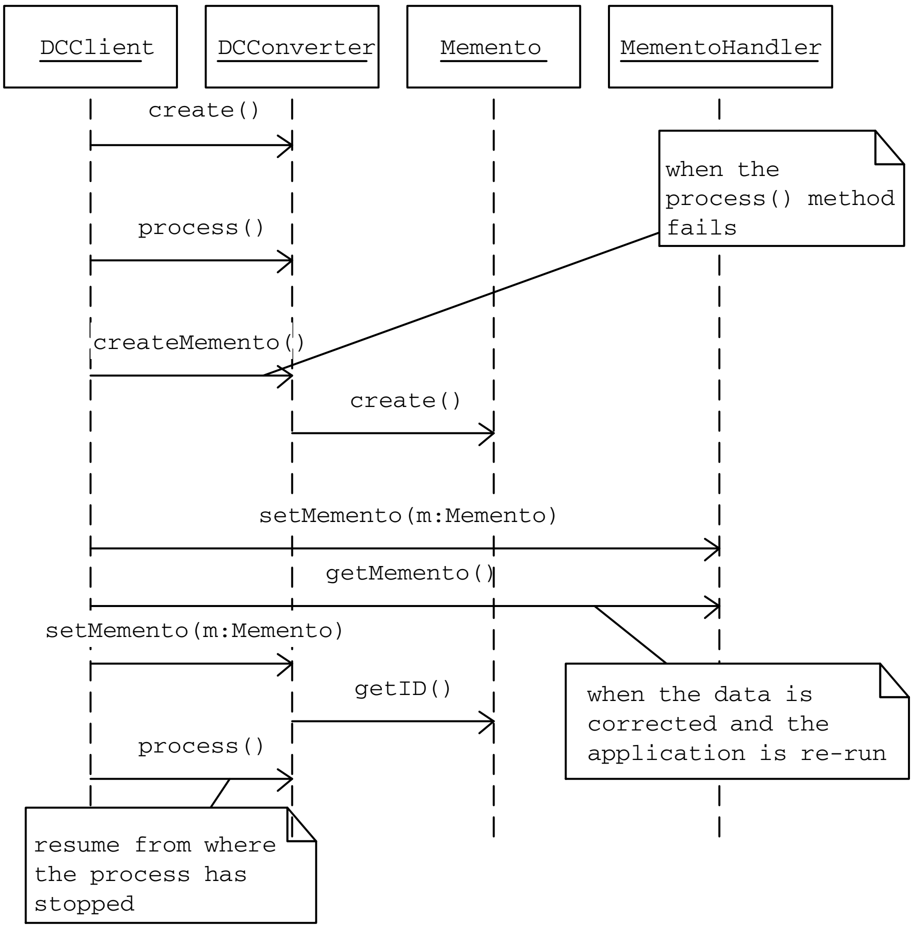

Figure 32.3 Application Message Flow