17

FLYWEIGHT

This pattern was previously described in GoF95.

DESCRIPTION

Every object can be viewed as consisting of one or both of the following two sets of information:

1. *Intrinsic Information* — The intrinsic information of an object is independent of the object context. That means the intrinsic information is the common information that remains constant among different instances of a given class. For example, the company information on a visiting card is the same for all employees.
2. *Extrinsic Information* — The extrinsic information of an object is dependent upon and varies with the object context. That means the extrinsic information is unique for every instance of a given class. For example, the employee name and title are extrinsic on a visiting card as this information is unique for every employee.

Consider an application scenario that involves creating a large number of objects that are unique only in terms of a few parameters. In other words, these objects contain some intrinsic, invariant data that is common among all objects. This intrinsic data needs to be created and maintained as part of every object that is being created. The overall creation and maintenance of a large group of such objects can be very expensive in terms of memory-usage and performance.

The Flyweight pattern can be used in such scenarios to design a more efficient way of creating objects.

The Flyweight pattern suggests separating all the intrinsic common data into a separate object referred to as a *Flyweight* object. The group of objects being created can share the `Flyweight` object as it represents their intrinsic state. This eliminates the need for storing the same invariant, intrinsic information in every object; instead it is stored only once in the form of a single `Flyweight` object. As a result, the client application can realize considerable savings in terms of the memory-usage and the time.

© 2004 by CRC Press LLC

Table 17.1 Flyweight Requirements

Serial Number	*Description*
1	There exists only one object of a given flyweight kind and is shared by all the other appropriate objects.
2	Client objects should not be allowed to create flyweight instances directly. At the same time, client objects should have a way of accessing a required `Flyweight` object when needed.

When the Flyweight pattern is applied, it is important to make sure that the requirements listed in Table 17.1 are satisfied.

HOW TO DESIGN A FLYWEIGHT IN JAVA

One of the ways to design a flyweight in Java is to design it as a singleton similar to the **`Flyweight`** class in Figure 17.1.

DESIGN HIGHLIGHTS

- The **`Flyweight`** class is designed with a private constructor. This is to prevent client objects from creating **`Flyweight`** instances by directly accessing its constructor.
- In general, a singleton is expected to maintain only one instance of itself. That is, the singleton nature is at the *class type level*. When a flyweight is designed as a singleton, it exhibits the singleton nature at the *flyweight type level*, not at the class type level. In other words, the singleton

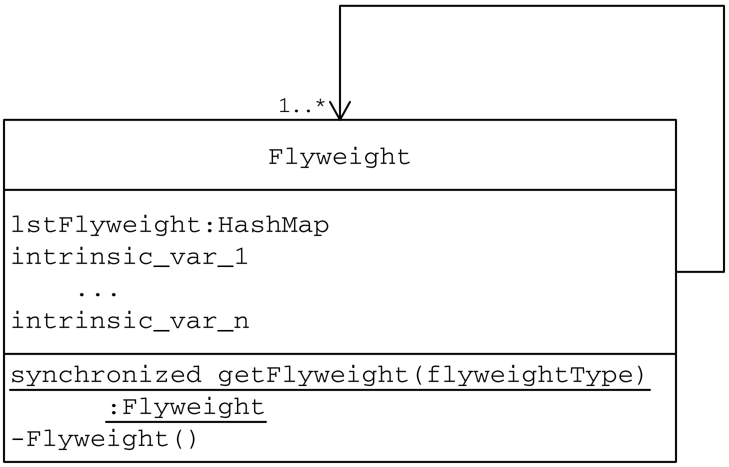

Figure 17.1 Flyweight as a Singleton

© 2004 by CRC Press LLC

`Flyweight` maintains a single instance of itself for every possible flyweight type in the system. These flyweight objects are stored in the `lstFlyweight` static variable. This can be viewed as a variation of the Singleton pattern.

Whenever a client needs to create an instance of a given flyweight type, it invokes the static `getFlyweight` method, passing the required flyweight type as an argument. The `getFlyweight` method is designed as a synchronized class level method to make it thread safe.

As part of its implementation of the `getFlyweight` method, the `Flyweight` checks to see if an instance of itself corresponding to the requested flyweight type already exists in the `lstFlyweight HashMap`.

- If it exists, the `Flyweight` returns the existing `Flyweight` object to the client.
- If it does not exist:
  - The `Flyweight` creates a new instance of itself corresponding to the requested flyweight type and adds it to the flyweights list maintained in the static `lstFlyweight HashMap` variable. Because the `getFlyweight` method is defined within the `Flyweight` class, it can access its private constructor to create an instance of itself.
  - It returns the newly created `Flyweight` object to the client.

The flyweight design discussed so far meets the requirements listed in Table 17.1. In general, a flyweight is designed solely to represent the intrinsic state of an object. Besides representing an object's intrinsic state, the flyweight contains the required data structures and the implementation to maintain different types of singleton `Flyweight` objects.

As an alternate design strategy, the responsibility of creating and maintaining different singleton `Flyweight` objects can be moved out of the `Flyweight` to a designated `FlyweightFactory`. The `Flyweight` can be designed as an inner class of the `FlyweightFactory` class. Since the `Flyweight` class is defined with a private constructor, external objects are prevented from creating its instances by directly invoking the constructor. But the `FlyweightFactory` can invoke the `Flyweight` class private constructor to create necessary `Flyweight` objects. This is because an outer class can access the private methods of its inner class.

In the new design (Figure 17.2), both the data structure (`lstFlyweight HashMap`) and the behavior (`getFlyweight` method) related to the creation and maintenance of singleton `Flyweight` objects are moved from the `Flyweight` class to the `FlyweightFactory` class. The `Flyweight` instances are used solely to represent an object's intrinsic state.

Whenever a client needs to create an instance of a given flyweight type, it invokes, the `getFlyweight` method on the singleton `FlyweightFactory` instance, passing the required flyweight type as an argument. The singleton `FlyweightFactory` object maintains the list of existing `Flyweight` objects in the `lstFlyweight` instance variable.

© 2004 by CRC Press LLC

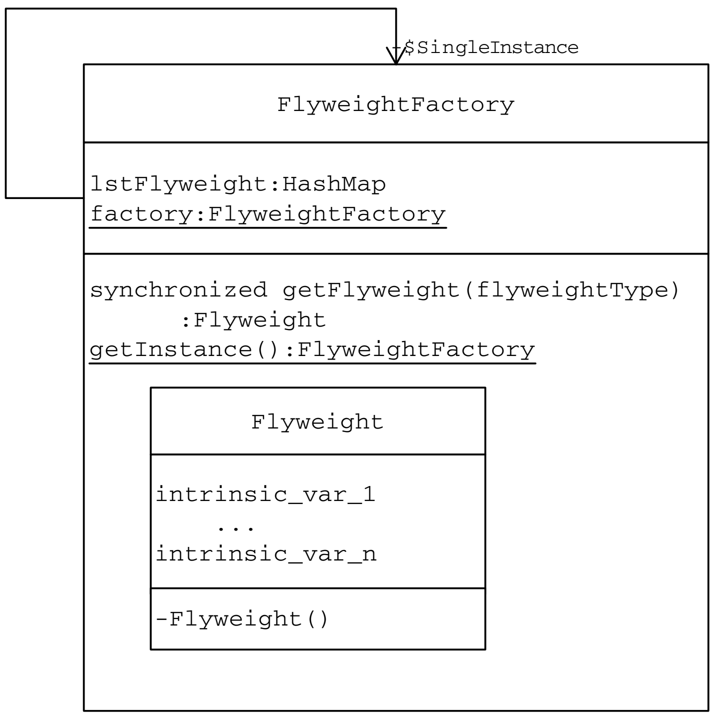

Figure 17.2 Flyweight as an Inner Class inside a Singleton Factory

As part of its implementation of the `getFlyweight` method, the `FlyweightFactory` checks to see if an instance of the `Flyweight` corresponding to the requested flyweight type already exists in the `lstFlyweight HashMap`.

- If it exists, the `FlyweightFactory` returns the existing `Flyweight` object to the client.
- If it does not exist:
  - The `FlyweightFactory` creates a new instance of the `Flyweight` corresponding to the requested flyweight type and adds it to the flyweights list maintained in the `lstFlyweight HashMap` variable.
  - It returns the newly created `Flyweight` object to the client.

The `FlyweighFactory` is designed as a singleton to prevent client objects from creating multiple instances of the `FlyweightFactory`, thereby creating multiple instances of a given flyweight kind.

Once the requested `Flyweight` object is received, the client can either:

- Create an object with the exclusive extrinsic data and associate the `Flyweight` object with it. This approach still results in the creation of a large number of objects but the design becomes more efficient as the intrinsic data is not duplicated in every object. Instead, it is kept inside a single shared `Flyweight` object (Design Approach I in the following example).

© 2004 by CRC Press LLC

- Send the extrinsic data as part of a method call to the `Flyweight` object. This approach results in the creation of few objects with no duplication (Design Approach II in the following example).

EXAMPLE

To demonstrate the use of the Flyweight pattern, let us design an application that prints out the data for visiting cards of all the employees of a large organization with four major divisional offices. A typical visiting card can be assumed to have the following layout:

```
<<Name of the Employee>>
                <<Title>>
<<Company Name>>
<<Divisional_Office_Address_Lines>>
<<City>><<State>><<ZipCode>>
```

From the visiting card data layout, it can be observed that:

- The name and the title are unique for every employee and can be considered as the extrinsic data.
- The company name remains the same for all employees and every employee working under a divisional office is given the same divisional office address. Therefore the company name and division address part of a visiting card can be treated as the intrinsic data.

One of the simplest strategies for designing this example application is to create a **`VCard`** class representing a visiting card as in Figure 17.3. The **`print()`** method can be implemented to display the visiting card data.

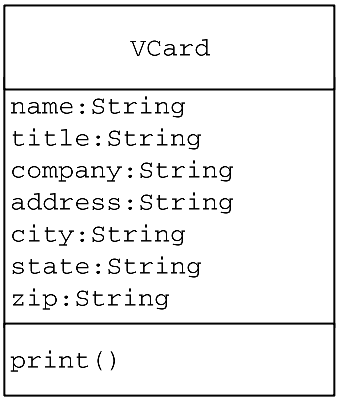

Figure 17.3 Visiting Card: Class Representation

© 2004 by CRC Press LLC

Usually, there will be thousands of employees in a large organization and hence the application may need to create thousands of `VCard` objects. As discussed earlier, the address part of the `VCard` class remains constant for all employees working under a given divisional office. Hence, adapting the class representation depicted in Figure 17.3 could lead to duplicate data being created and maintained as part of every new `VCard` instance created. Using the Flyweight pattern, the need for storing the duplicate data can be eliminated.

DESIGN APPROACH I

In this approach, the extrinsic data is represented as an object and configured with a `Flyweight` object representing its intrinsic data.

Applying the Flyweight pattern, all the intrinsic data can be moved out of the `VCard` class into a separate `Flyweight` class. Let us define an interface `FlyweightIntr` to be implemented by the `Flyweight` class representing the visiting card intrinsic data.

```
public interface FlyweightIntr {
  public String getCompany();
  public String getAddress();
  public String getCity();
  public String getState();
  public String getZip();
}
```

As we discussed under the "How to Design a Flyweight in Java" section earlier, let us define a singleton `FlyweightFactory` (Listing 17.1) with the responsibility of creating and maintaining single instances of different `Flyweight` objects corresponding to different divisions.

The actual `Flyweight` class can be defined within the `FlyweightFactory` as an inner class (Figure 17.4).

The design highlights of the concrete `Flyweight` and the `FlyweightFactory` classes are as follows:

- The concrete `Flyweight` class is designed with a private constructor to prevent external objects from creating Flyweight instances directly by invoking its constructor.
- The concrete `Flyweight` class is designed as an inner class within the `FlyweightFactory` to allow the `FlyweightFactory` to invoke its private constructor.
- The `FlyweightFactory` is responsible for creating and managing different Flyweight instances. It maintains a list of different `Flyweight` objects inside the `lstFlyweight HashMap` instance variable. When a client requests a `Flyweight` object corresponding to a specific division, the `FlyweightFactory` checks the list of existing `Flyweight` objects to see if it is already created. If it is already created and available in the list, the `FlyweightFactory` returns the existing `Flyweight` object. If

© 2004 by CRC Press LLC

Listing 17.1 Singleton `FlyweightFactory` Class with Inner `Flyweight` Class

```
//singleton Flyweight Factory
public class FlyweightFactory {
  private HashMap lstFlyweight;
  private static FlyweightFactory factory =
    new FlyweightFactory();
  private FlyweightFactory() {
    lstFlyweight = new HashMap();
  }
  public synchronized FlyweightIntr getFlyweight(
    String divisionName) {
    if (lstFlyweight.get(divisionName) == null) {
      FlyweightIntr fw = new Flyweight(divisionName);
      lstFlyweight.put(divisionName, fw);
      return fw;
    } else {
      return (FlyweightIntr) lstFlyweight.get(divisionName);
    }
  }
  public static FlyweightFactory getInstance() {
    return factory;
  }
  //Inner flyweight class
  private class Flyweight implements FlyweightIntr {
    private String company;
    private String address;
    private String city;
    private String state;
    private String zip;
    private void setValues(String cmp, String addr,
        String cty, String st, String zp) {
      company = cmp;
      address = addr;
      city = cty;
      state = st;
      zip = zp;
  }
```

(continued)

© 2004 by CRC Press LLC

Listing 17.1 Singleton `FlyweightFactory` Class with Inner `Flyweight` Class (Continued)

```
    private Flyweight(String division) {
      //values are hard coded
      //for simplicity
      if (division.equals("North")) {
        setValues("CMP″,"addr1″, "cty1″,"st1″,"10000");
      }
      if (division.equals("South")) {
        setValues("CMP″,"addr2″, "cty2″,"st2″,"20000");
      }
      if (division.equals("East")) {
        setValues("CMP″,"addr3″, "cty3″,"st3″,"30000");
      }
      if (division.equals("West")) {
        setValues("CMP″,"addr4″, "cty4″,"st4″,"40000");
      }
    }
    public String getCompany() {
      return company;
    }
    public String getAddress() {
      return address;
    }
    public String getCity() {
      return city;
    }
    public String getState() {
      return state;
    }
    public String getZip() {
      return zip;
    }
  }//end of Flyweight
}//end of FlyweightFactory
```

not, the `FlyweightFactory` creates the requested `Flyweight` object, stores it inside the list and returns it to the client. Subsequently, when a client object requests a flyweight corresponding to the same division, the `Flyweight` object is not created. Instead, the corresponding `Flyweight`

© 2004 by CRC Press LLC

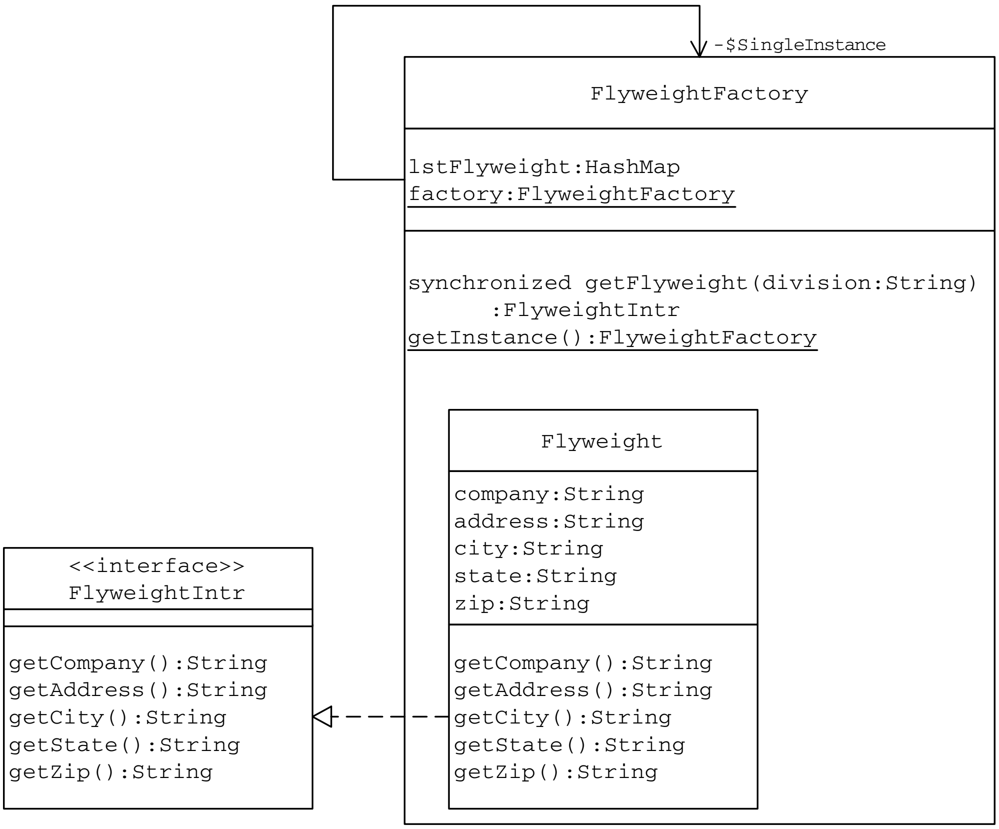

Figure 17.4 `FlyweightFactory` with an Inner `Flyweight` Class

object from the `lstFlyweight` `HashMap` instance variable is returned to the client.

- The `FlyweightFactory` is designed as a singleton so that it guarantees the uniqueness of `Flyweight` objects.

After the intrinsic data is removed, the extrinsic data still remains within the `VCard` class (Listing 17.2). As part of its constructor, the `VCard` accepts a `Flyweight` instance representing its intrinsic data. The `print()` method displays extrinsic data from the `VCard` and intrinsic data from the associated `Flyweight` object.

With these objects in place, in order to print the visiting card data, a client object such as `FlyweightTest` (Listing 17.3):

1. Creates an instance of the singleton `FlyweightFactory`.
2. Requests an appropriate `Flyweight` object (for every employee) by invoking the `getFlyweight` method on the singleton `FlyweightFactory` instance, passing the division that the employee works for as an argument. In response, the `FlyweightFactory` returns a `Flyweight` instance corresponding to the specified division. Since the example organization is

© 2004 by CRC Press LLC

Listing 17.2 `VCard` Class Using a `Flyweight` Object to Represent the Intrinsic Data

```
public class VCard {
  String name;
  String title;
  FlyweightIntr objFW;
  public VCard(String n, String t, FlyweightIntr fw) {
    name = n;
    title = t;
    objFW = fw;
  }
  public void print() {
    System.out.println(name);
    System.out.println(title);
    System.out.println(objFW.getAddress() + "-" +
                       objFW.getCity() + "-" +
                       objFW.getState() + "-" +
                       objFW.getZip());
    System.out.println(" – – – – – – – – ");
  }
}
```

assumed to have only four divisions, there will be a maximum of four `Flyweight` objects created when the application is run.

3. Receives the requested `Flyweight` object and then associates the Flyweight with the `VCard` instance representing the extrinsic data by passing the `Flyweight` object as an argument to the `VCard` constructor.

Figure 17.5 shows the overall class association.

The following sequence diagram in Figure 17.6 depicts the message flow during the creation and display of the visiting card data of an employee working for a specific divisional office.

This approach requires the creation of a `VCard` object for every employee in addition to the four `Flyweight` objects. The fact that the intrinsic data is not duplicated in every `VCard` object results in savings in terms of the memory usage and the time it takes to create objects.

DESIGN APPROACH II

Extrinsic data passed to the flyweight as part of a method call and was not represented as an object (Listing 17.4).

This design approach requires the following two changes to the application design discussed in Design Approach I.

© 2004 by CRC Press LLC

Listing 17.3 Client `FlyweightTest` Class

```
public class FlyweightTest {
  public static void main(String[] args) throws Exception {
  Vector empList = initialize();
    FlyweightFactory factory =
      FlyweightFactory.getInstance();
    for (int i = 0; i < empList.size(); i++) {
      String s = (String) empList.elementAt(i);
      StringTokenizer st = new StringTokenizer(s, ,"");
      String name = st.nextToken();
      String title = st.nextToken();
      String division = st.nextToken();
      FlyweightIntr flyweight =
        factory.getFlyweight(division);
      //associate the flyweight
      //with the extrinsic data object.
      VCard card = new VCard(name, title, flyweight);
      card.print();
    }
  }
  private static Vector initialize() {
          …
          …
  }
}
```

- The print method:
  - Needs to be moved from the `VCard` class to the `Flyweight` class.
  - Signature needs to be changed from

```
public void print()
```

to

```
public void print(String name, String title)
```

in order to accept the extrinsic data as arguments.

- Should be implemented to display the extrinsic data passed to it along with the intrinsic data it represents.

© 2004 by CRC Press LLC

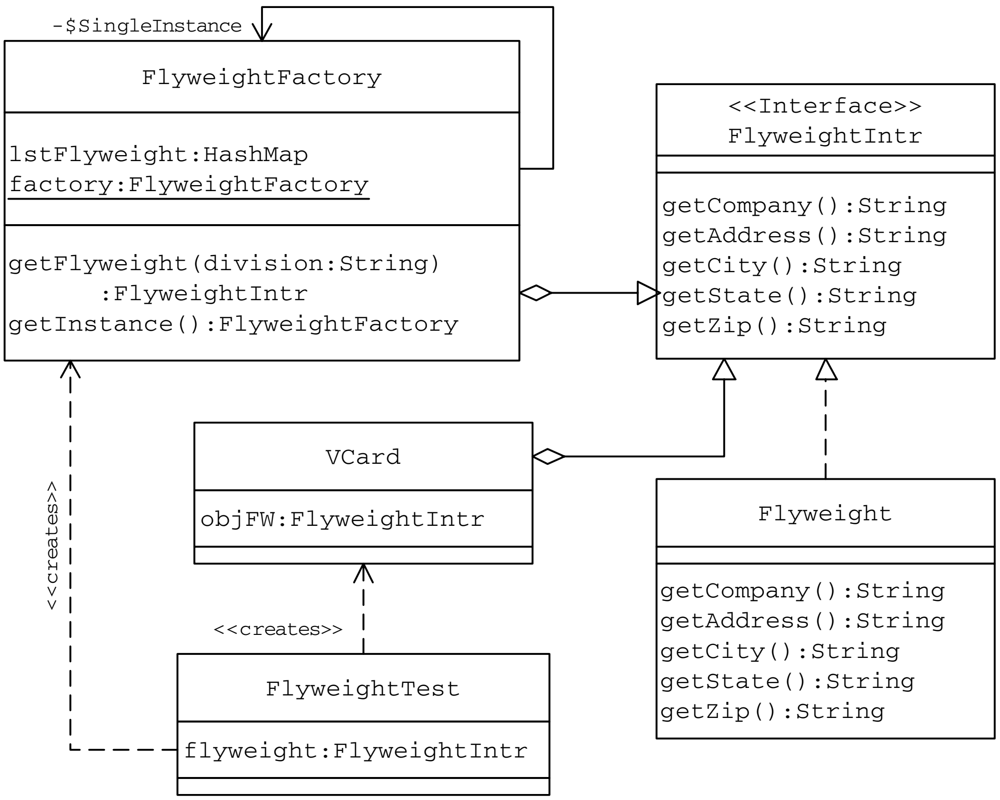

Figure 17.5 Design Approach I: Class Association

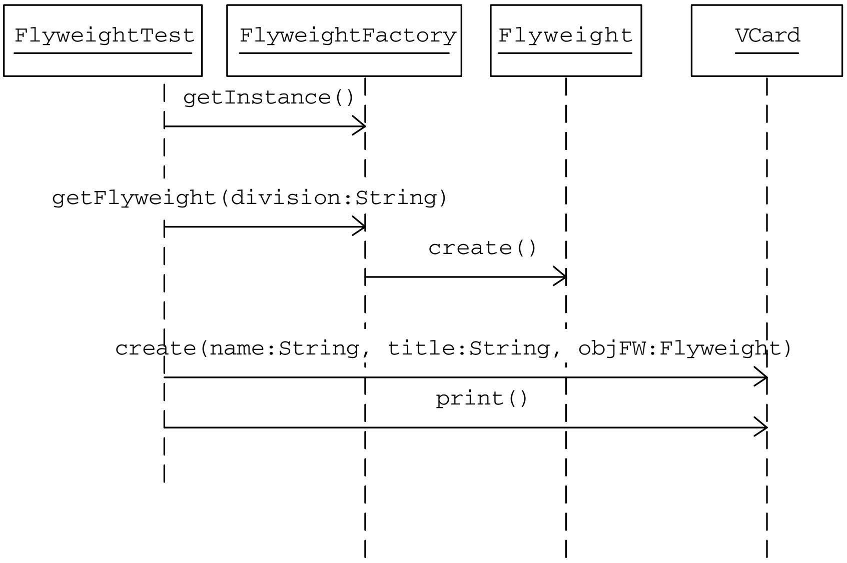
FlyweightTest
FlyweightFactory
Flyweight
VCard
getInstance()
getFlyweight(division:String)
create()
create(name:String, title:String, objFW:Flyweight)
print()

Figure 17.6 Message Flow: Design Approach I

© 2004 by CRC Press LLC

Listing 17.4 Revised `FlyweightFactory` Class

```
public interface FlyweightIntr {
  public String getCompany();
  public String getAddress();
  public String getCity();
  public String getState();
  public String getZip();
  public void print(String name, String title);
}
//singleton Flyweight Factory
public class FlyweightFactory {
          …
          …
    //Inner flyweight class
    private class Flyweight implements FlyweightIntr {
          …
          …
      public void print(String name, String title) {
        System.out.println(name);
        System.out.println(title);
        System.out.println(getAddress() + "-" + getCity() +
                             "-" + getState() + "-" + getZip());
        System.out.println(" – – – – – – – – – ");
      }
  }//end of Flyweight
}//end of FlyweightFactory
```

- `VCard` class:
  - Since the extrinsic data is to be passed to the `Flyweight` object and hence the `VCard` class is no longer needed, this class can be removed from the design.

In the new design, the client `FlyweightTest` (Listing 17.5):

- First creates an instance of the singleton `FlyweightFactory`. The design and implementation of the `FlyweightFactory` remains the same as in Design Approach I.
- For every employee, the application requests the `FlyweightFactory` for an appropriate `Flyweight` object by passing the division that the employee works for.

© 2004 by CRC Press LLC

Listing 17.5 Client `FlyweightTest` Class

```
public class FlyweightTest {
  public static void main(String[] args) throws Exception {
    Vector empList = initialize();
    FlyweightFactory factory =
      FlyweightFactory.getInstance();
    for (int i = 0; i < empList.size(); i++) {
      String s = (String) empList.elementAt(i);
      StringTokenizer st = new StringTokenizer(s, ,"");
      String name = st.nextToken();
      String title = st.nextToken();
      String division = st.nextToken();
      FlyweightIntr flyweight =
        factory.getFlyweight(division);
      //pass the extrinsic data
      //as part of a method call.
      flyweight.print(name, title);
    }
  }
          …
          …
}
```

- Once the requested **`Flyweight`** object is received, the client invokes the print method on the **`Flyweight`** object by passing the extrinsic data (employee name and title).

Figure 17.7 shows the class association in the revised design.

The sequence diagram in Figure 17.8 depicts the message flow in the new design during the creation and display of the visiting card data of an employee working for a specific divisional office.

Because the extrinsic data is not designed as an object, this approach requires the creation of only four **`Flyweight`** objects, each corresponding to a divisional office with no duplication. This substantially reduces the memory usage and the time required for the object creation.

PRACTICE QUESTIONS

1. Let us consider an online job site that receives XML data files from different employers with current openings in their organizations. When the number of vacancies is small, employers can enter details online. When the number

© 2004 by CRC Press LLC

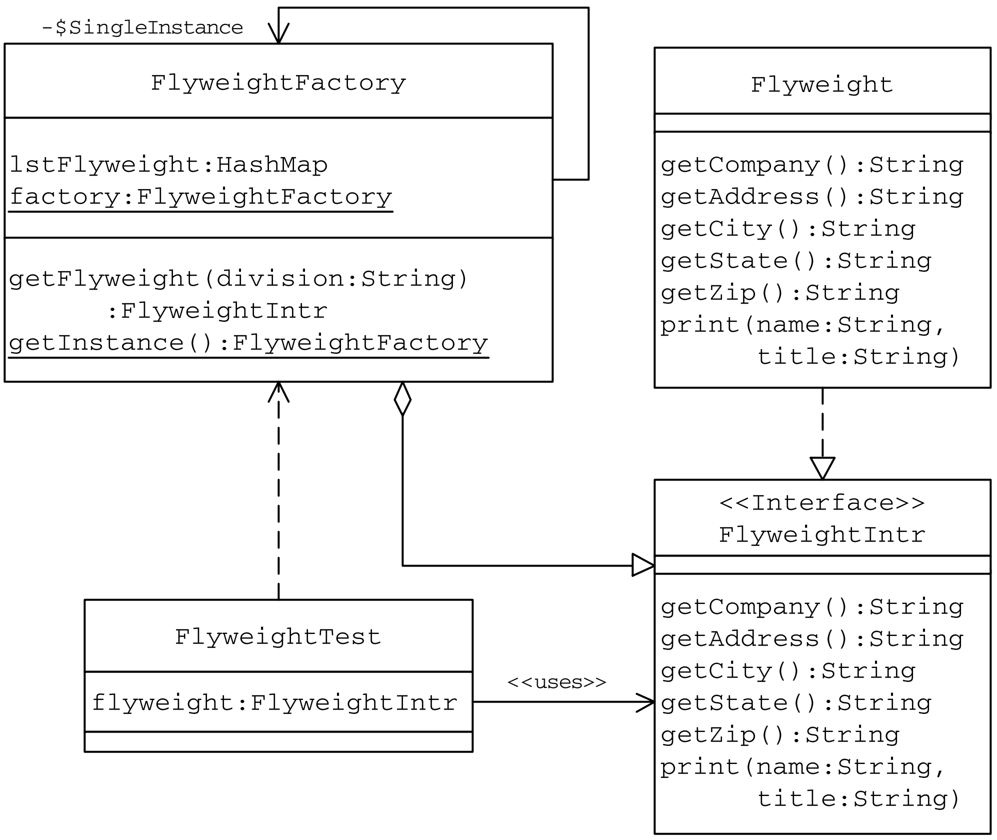

Figure 17.7 Design Approach II: Class Association

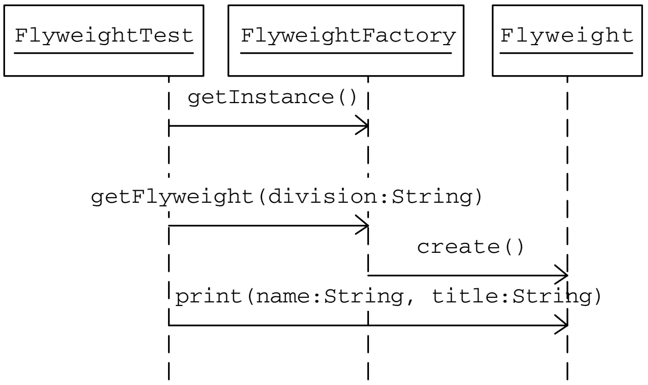
FlyweightTest
FlyweightFactory
Flyweight
getInstance()
getFlyweight(division:String)
create()
print(name:String, title:String)

Figure 17.8 Message Flow: Design Approach II

of vacancies is large, employers upload details in the form of an XML file. Once the XML file is received, it needs to be parsed and processed. Let us assume the XML file to have the following details:

a. Job title
b. Minimum qualifications
c. Medical insurance
d. Dental insurance
e. Vision care
f. 401K
g. Minimum number of hours of work
h. Paid vacation
i. Employer name
j. Employer address

In general, details from (c) through (j) are all considered to be the same for all jobs posted by a given employer. Apply the Flyweight pattern to design the process of parsing the input XML file and creating different `Job` objects.

2. A computer user in a typical organization is associated with a user account. A user account can be part of one or more groups. Permissions on different resources (such as servers, printers, etc.) are defined at the group level. Users get all the permissions defined for all groups that their accounts are part of. Let us consider an organization with three different user groups — `Administrators`, `FieldOfficers` and `SalesReps`. Further assume that the organization is in the process of migrating user accounts from one server to a different server environment. As part of this process, all user accounts are first exported to an XML file as follows.

```
<Users>
  <User>
    <UserName>PKuchana</UserName>
    <Password>PKuchana</Password>
    <Group>FieldOfficers</Group>
    <Permissions>
      <Permission>Perm1</Permission>
      <Permission>Perm2</Permission>
    </Permissions>
  </User>
  <User>
    <UserName>VKuchana</UserName>
    <Password>VKuchana</Password>
    <Group>SalesReps</Group>
    <Permissions>
      <Permission>Perm1</Permission>
      <Permission>Perm4</Permission>
```

© 2004 by CRC Press LLC

```
      </Permissions>
    </User>
      …
      …
  </Users>
```

It is to be noted that permissions for all accounts in a given group are the same. This can be considered as the intrinsic data. The user name and the password details vary from user to user and should be treated as extrinsic data.
User accounts in the new server environment are created using the exported XML file. Make any necessary assumptions and design an application using the Flyweight pattern to parse the XML file to create different user account objects.

© 2004 by CRC Press LLC