13

PROTOTYPE

This pattern was previously described in GoF95.

DESCRIPTION

As discussed in earlier chapters, both the Factory Method and the Abstract Factory patterns allow a system to be independent of the object creation process. In other words, these patterns enable a client object to create an instance of an appropriate class by invoking a designated method without having to specify the exact concrete class to be instantiated. While addressing the same problem as the Factory Method and Abstract Factory patterns, the Prototype pattern offers a different, more flexible way of achieveing the same result.

Other uses of the Prototype pattern include:

- When a client needs to create a set of objects that are alike or differ from each other only in terms of their state and it is expensive to create such objects in terms of the time and the processing involved.
- As an alternative to building numerous factories that mirror the classes to be instantiated (as in the Factory Method).

In such cases, the Prototype pattern suggests to:

- Create one object upfront and designate it as a prototype object.
- Create other objects by simply making a copy of the prototype object and making required modifications.

In the real world, we use the Prototype pattern on many occasions to reduce the time and effort spent on different tasks. The following are two such examples:

1. *New Software Program Creation* — Typically programmers tend to make a copy of an existing program with similar structure and modify it to create new programs.
2. *Cover Letters* — When applying for positions at different organizations, an applicant may not create cover letters for each organization individually from scratch. Instead, the applicant would create one cover letter in the

most appealing format, make a copy of it and personalize it for every organization.

As can be seen from the examples above, some of the objects are created from scratch, whereas other objects are created as copies of existing objects and then modified. But the system or the process that uses these objects does not differentiate between them on the basis of how they are actually created. In a similar manner, when using the Prototype pattern, a system should be independent of the creation, composition and representation details of the objects it uses.

One of the requirements of the prototype object is that it should provide a way for clients to create a copy of it. By default, all Java objects inherit the built-in `clone()` method from the topmost `java.lang.Object` class. The built-in `clone()` method creates a clone of the original object as a shallow copy.

SHALLOW COPY VERSUS DEEP COPY

When an object is cloned as a shallow copy:

- The original top-level object and all of its primitive members are duplicated.
- Any lower-level objects that the top-level object contains are not duplicated. Only references to these objects are copied. This results in both the orginal and the cloned object referring to the same copy of the lower-level object. Figure 13.1 shows this behavior.

In contrast, when an object is cloned as a deep copy:

- The original top-level object and all of its primitive members are duplicated.
- Any lower-level objects that the top-level object contains are also duplicated. In this case, both the orginal and the cloned object refer to two different lower-level objects. Figure 13.2 shows this behavior.

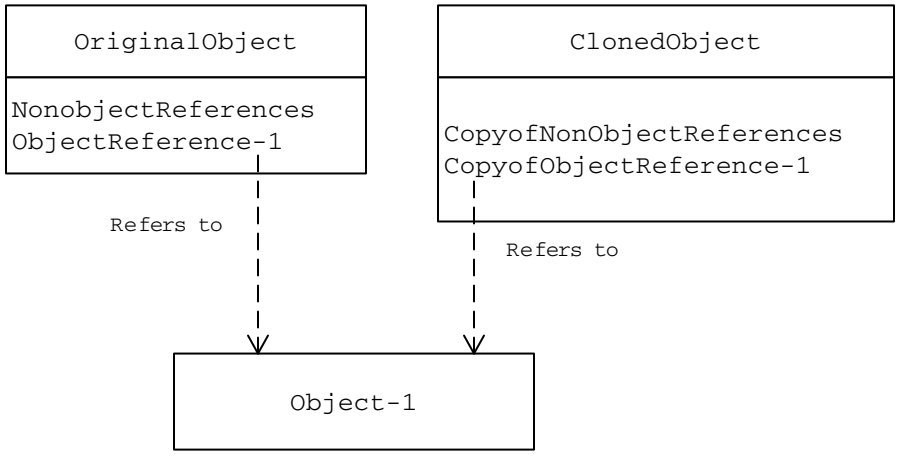

Figure 13.1 Shallow Copy

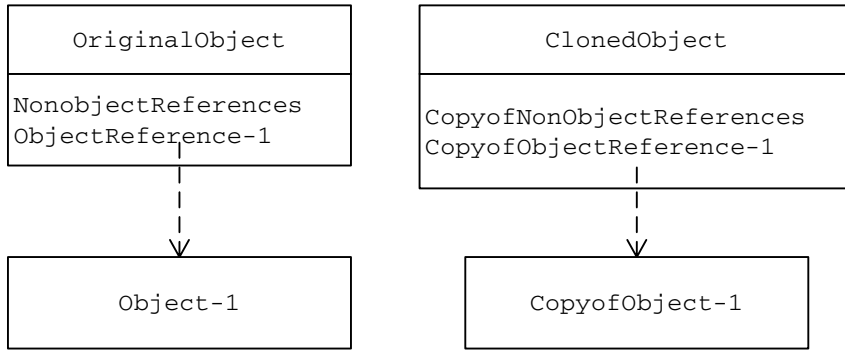

Figure 13.2 Deep Copy

Shallow Copy Example

The following is an example of creating a shallow copy using the built-in `java.lang.Object clone()` method. Let us design a `Person` class (Listing 13.1) as an implementer of the built-in Java `java.lang.Cloneable` interface with two attributes, a string variable `name` and a `Car` object `car`.

In general, a class must implement the `Cloneable` interface to indicate that a field-for-field copy of instances of that class is allowed by the `Object.clone()` method. When a class implements the `Cloneable` interface, it should override the `Object.clone` method with a public method. Note that when the `clone` method is invoked on an object that does not implement the `Cloneable` interface, the exception `CloneNotSupportedException` is thrown.

As part of its implementation of the public `clone` method, the `Person` class simply invokes the built-in `clone` method. The built-in `clone` method creates a clone of the current object as a shallow copy, which is returned to the calling client object.

Let us design a client `ShallowCopyTest` (Listing 13.2) to demonstrate the behavior of a shallow copy object. To demonstrate the fact that the shallow copy process duplicates nonobject references only but not object references, the client:

- Creates an instance of the `Person` class
- Creates a clone of the `Person` object created above and alters the values of its attributes
- Displays the values of its attributes at different stages

When the `Car` object associated with the cloned object is modified, it can be seen that the `Car` object associated with the original object gets affected. This is because the lower-level `Car` object is not duplicated and is shared by both the original and the cloned `Person` objects, whereas the `name` attribute value of the orginal object does not get affected when the cloned object's `name` attribute value is altered. This is because the shallow copy process duplicates attributes that are of primitive types.

Listing 13.1 `Person` Class

```
class Person implements Cloneable {
  //Lower-level object
  private Car car;

  private String name;
  public Car getCar() {
    return car;
  }
  public String getName() {
    return name;
  }
  public void setName(String s) {
    name = s;
  }
  public Person(String s, String t) {
    name = s;
    car = new Car(t);
  }
  public Object clone() {
    //shallow copy
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      return null;
    }
  }
}
class Car {

  private String name;

  public String getName() {
    return name;
  }
  public void setName(String s) {
    name = s;
  }
  public Car(String s) {
    name = s;
  }
```

Listing 13.2 Client `ShallowCopyTest` Class

```
public class ShallowCopyTest {
  public static void main(String[] args) {
    //Original Object
    Person p = new Person("Person-A,""Civic");
    System.out.println("Original (orginal values): " +
                       p.getName() + " - " +
                       p.getCar().getName());
    //Clone as a shallow copy
    Person q = (Person) p.clone();
    System.out.println("Clone (before change): " +
                       q.getName() + " - " +
                       q.getCar().getName());
    //change the primitive member
    q.setName("Person-B");
    //change the lower-level object
    q.getCar().setName("Accord");
    System.out.println("Clone (after change): " +
                       q.getName() + " - " +
                       q.getCar().getName());
    System.out.println(
      "Original (after clone is modified): " +
      p.getName() + " - " + p.getCar().getName());
  }
}
```

When this program is run, the following output is displayed:

```
Original (orginal values): Person-A - Civic
Clone (before change): Person-A - Civic
Clone (after change): Person-B - Accord
Original (after clone is modified): Person-A - Accord
```

Deep Copy Example

The same example above can be redesigned by overriding the built-in `clone()` method to create a deep copy of the `Person` object (Listing 13.3). As part of its implementation of the `clone` method, to create a deep copy, the `Person` class creates a new `Person` object with its attribute values the same as the original object and returns it to the client object.

Listing 13.3 `Person` Class Revised

```
class Person implements Cloneable {
  //Lower-level object
  private Car car;
  private String name;
  public Car getCar() {
    return car;
  }
  public String getName() {
    return name;
  }
  public void setName(String s) {
    name = s;
  }
  public Person(String s, String t) {
    name = s;
    car = new Car(t);
  }
  public Object clone() {
    //Deep copy
    Person p = new Person(name, car.getName());
    return p;
  }
}
class Car {
  private String name;
  public String getName() {
    return name;
  }
  public void setName(String s) {
    name = s;
  }
  public Car(String s) {
    name = s;
  }
}
```

Listing 13.4 Client `DeepCopyTest` Class

```
public class DeepCopyTest {
  public static void main(String[] args) {
    //Original Object
    Person p = new Person("Person-A","Civic");
    System.out.println("Original (orginal values): " +
                       p.getName() + " - " +
                       p.getCar().getName());
    //Clone as a shallow copy
    Person q = (Person) p.clone();
    System.out.println("Clone (before change): " +
                       q.getName() + " - " +
                       q.getCar().getName());
    //change the primitive member
    q.setName("Person-B");
    //change the lower-level object
    q.getCar().setName("Accord");
    System.out.println("Clone (after change): " +
                       q.getName() + " - " +
                       q.getCar().getName());
    System.out.println(
      "Original (after clone is modified): " +
      p.getName() + " - " + p.getCar().getName());
  }
}
```

Similar to the client `ShallowCopyTest`, a new client `DeepCopyTest` (Listing 13.4) can be designed to:

- Create an instance of the `Person` class
- Create a clone of the `Person` object created above and alter the values of its attributes
- Display the values of its attributes at different stages

When the `Car` object associated with the cloned object is modified, it can be seen that the `Car` object associated with the original object *does not* get affected. This is because the lower-level `Car` object is duplicated and is not shared by both the original and the cloned `Person` objects.

Similar to a shallow copy, the `name` attribute value of the orginal object does not get affected when the cloned object's `name` attribute value is altered. This is because in addition to attributes that are object references, the deep copy process duplicates those attributes that are of primitive types.

When the client `DeepCopyTest` is run, it displays the following output. From the output it can be seen that the lower-level `Car` object of the original `Person` object is unaffected when its clone is modified.

```
Original (orginal values): Person-A - Civic
Clone (before change): Person-A - Civic
Clone (after change): Person-B - Accord
Original (after clone is modified): Person-A - Civic
```

EXAMPLE I

Let us consider Practice Question 3 from Chapter 12 — Abstract Factory. The representation of different hosting packages would have resulted in a class hierarchy as shown in Figure 13.3.

Applying the Abstract Factory pattern, the application design would have resulted in a factory class hierarchy as shown in Figure 13.4.

In Figure 13.4, the `HostingPlanFactory` plays the role of an abstract factory whereas `WinPlanFactory` and `UnixPlanFactory` act as concrete factories. Each of these concrete factories would be responsible for the creation of a family of related classes that represent hosting packages on a specific platform as follows:

- `WinPlanFactory` would be responsible for the creation of `WinBasic`, `WinPremium` and `WinPremiumPlus` objects.

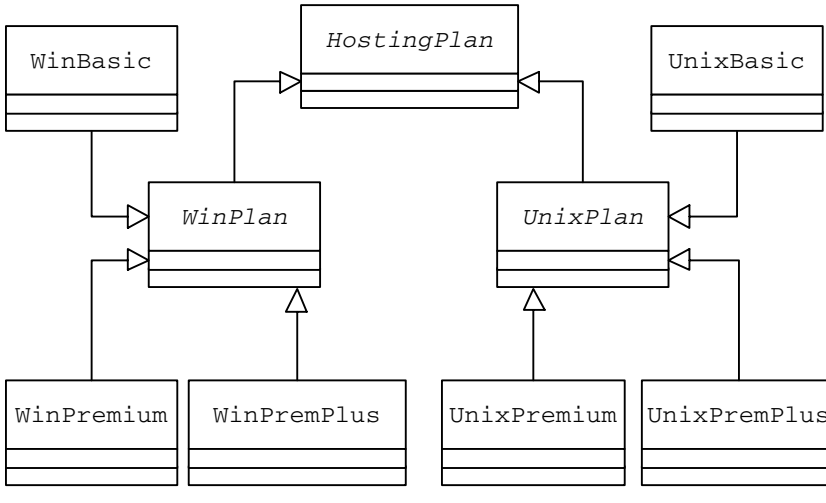

Figure 13.3 Hosting Packages Class Hierarchy

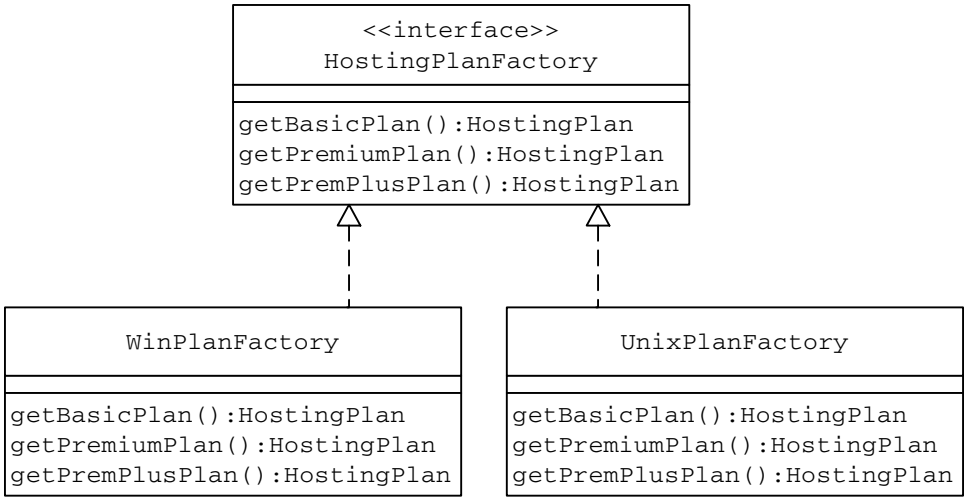

Figure 13.4 Hosting Packages Factory Class Hierarchy

- `UnixPlanFactory` would be responsible for the creation of `UnixBasic`, `UnixPremium` and `UnixPremiumPlus` objects.

Client objects can make use of an appropriate concrete factory class instance to create required `HostingPlan` objects.

Let us design the same application using the Prototype pattern. Applying the Prototype pattern, the `HostingPlanFactory` class hierarchy in Figure 13.4 can be replaced with a single concrete class `HostingPlanKit` (Figure 13.5 and Listing 13.5).

Design Highlights of the `HostingPlanKit` Class

- Maintains different prototypical objects that represent different types of hosting packages in its instance variables.

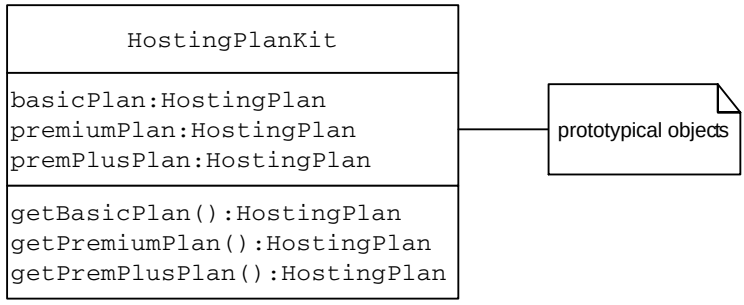

Figure 13.5 Single Class Equivalent of the Abstract Factory Class Hierarchy

Listing 13.5 `HostingPlanKit` Class

```
public class HostingPlanKit {
  private HostingPlan basicPlan;
  private HostingPlan premiumPlan;
  private HostingPlan premPlusPlan;
  public HostingPlanKit(HostingPlan basic, HostingPlan premium,
      HostingPlan premPlus) {
    basicPlan = basic;
    premiumPlan = premium;
    premPlusPlan = premPlus;
  }
  public HostingPlan getBasicPlan() {
    return (HostingPlan) basicPlan.clone();
  }
  public HostingPlan getPremiumPlan() {
    return (HostingPlan) premiumPlan.clone();
  }
  public HostingPlan getPremPlusPlan() {
    return (HostingPlan) premPlusPlan.clone();
  }
}
```

- Offers a set of methods that can be used by different client objects to get access to objects representing different hosting plans. As part of its implementation of these methods, it returns copies of the prototypical objects it contains.

For a client object to be able to make use of a `HostingPlanKit` instance, the `HostingPlanKit` instance must be configured with appropriate prototypical objects.

Let us design a separate class `HostingPlanManager` (Figure 13.6) with the responsibility of configuring a `HostingPlanKit` object with appropriate prototypical objects and return it to client objects.

```
public class HostingPlanManager {
  public static HostingPlanKit getHostingPlanKit(
    String platform) {
    HostingPlan basicPlan = null;
    HostingPlan premiumPlan = null;
    HostingPlan premPlusPlan = null;
    if (platform.equalsIgnoreCase("Win")) {
```

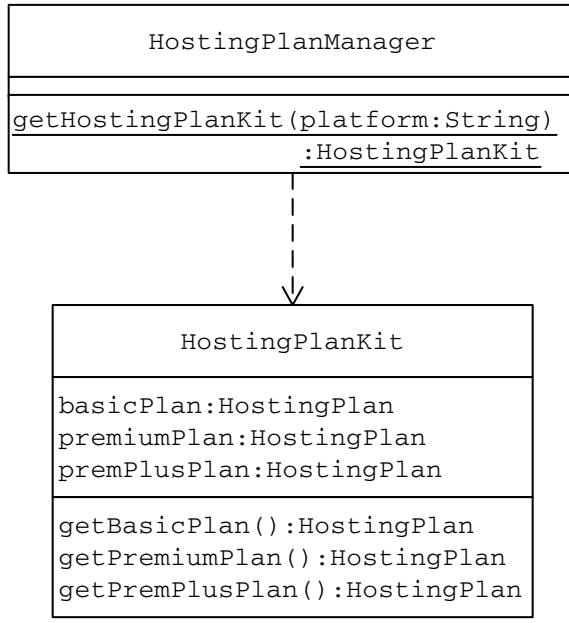

Figure 13.6 `HostingPlanManager` Class Representation

```
        basicPlan = new WinBasic();
        premiumPlan = new WinPremium();
        premPlusPlan = new WinPremPlus();
      }
      if (platform.equalsIgnoreCase("Unix")) {
        basicPlan = new UnixBasic();
        premiumPlan = new UnixPremium();
        premPlusPlan = new UnixPremPlus();
      }
      return new HostingPlanKit(basicPlan, premiumPlan,
              premPlusPlan);
    }
  }
```

The `HostingPlanManager` offers a static method `getHostingPlanKit` that can be used by client objects to get access to a `HostingPlanKit` object configured with prototypical `HostingPlan` objects that represent hosting plans on the specified platform. As an alternative design strategy, the static method `getHostingPlanKit` can be designed as part of the `HostingPlanKit` class itself.

Once the `HostingPlanKit` object is received, a client can make use of `getBasicPlan/getPremiumPlan/getPremPlusPlan` methods to get access to `HostingPlan` objects.

```
public class TestClient {
  public static void main(String[] args) {
    HostingPlanManager manager = new HostingPlanManager();
    HostingPlanKit kit = manager.getHostingPlanKit("Win");
    HostingPlan plan = kit.getBasicPlan();
    System.out.println(plan.getFeatures());
    plan = kit.getPremiumPlan();
    System.out.println(plan.getFeatures());
  }
}
```

EXAMPLE II

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. A user gets all the permissions defined for all groups that his or her account is part of. Let us build an application to facilitate the creation of user accounts. For simplicity, let us consider only two groups — Supervisor and AccountRep — representing users who are supervisors and account representatives, respectively.

Let us define a `UserAccount` class (Figure 13.7 and Listing 13.6) that represents a typical user account.

A typical `UserAccount` object maintains user-specific data such as firstname and lastname as strings and maintains the set of user permissions in the form of a vector.

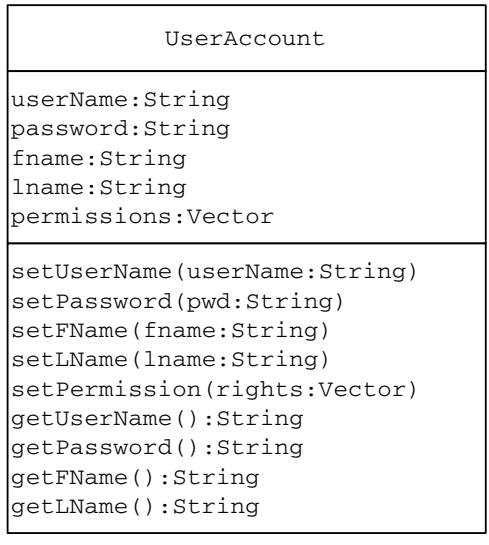

Figure 13.7 `UserAccount` Representation

Listing 13.6 `UserAccount` Class

```
public class UserAccount {
  private String userName;
  private String password;
  private String fname;
  private String lname;
  private Vector permissions = new Vector();
  public void setUserName(String uName) {
    userName = uName;
  }
  public String getUserName() {
    return userName;
  }
  public void setPassword(String pwd) {
    password = pwd;
  }
  public String getPassword() {
    return password;
  }
  public void setFName(String name) {
    fname = name;
  }
  public String getFName() {
    return fname;
  }
  public void setLName(String name) {
    lname = name;
  }
  public String getLName() {
    return lname;
  }
  public void setPermissions(Vector rights) {
    permissions = rights;
  }
  public Vector getPermissions() {
    return permissions;
  }
}
```

For simplicity, let us define the set of permissions for each of the Supervisor and the AccountRep groups in the form of two text files — `supervisor.txt` and `accountrep.txt`, respectively. With this arrangement, one of the simplest ways to create a user account is to:

- Instantiate the `UserAccount` class
- Read permissions from an appropriate data file
- Set these permissions in the `UserAccount` object

Though this approach looks straightforward, it is not efficient as it involves expensive file I/O (input/output) each time an account is created. This process can be designed more efficiently using the Prototype pattern. Applying the Prototype pattern, let us make the following changes to the design.

Redesign the `UserAccount` Class

The `UserAccount` class needs to be redesigned to provide a way for clients to create a clone of it (Listing 13.7). This can be accomplished by:

- Designing the `UserAccount` class to implement the `Cloneable` interface
- Returning a shallow copy of itself as part of its implementation of the `clone` method

Listing 13.7 `UserAccount` Class Revised

```
public class UserAccount implements Cloneable {
  private String userName;
  private String password;
  private String fname;
  private String lname;
  private Vector permissions = new Vector();
          …
          …
  public Object clone() {
    //Shallow Copy
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      return null;
  }
           …
          …
}
```

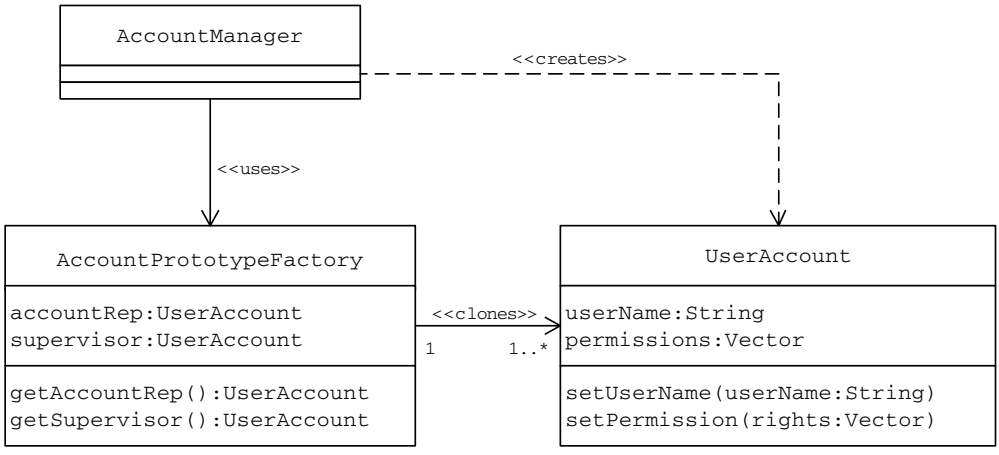

Figure 13.8 `UserAccount` Creation Utility: Class Association

Create a Prototype Factory Class

A new class, `AccountPrototypeFactory`, can be defined to hold prototypical `UserAccount` objects representing Supervisor and AccountRep type accounts. When requested by a client, the `AccountPrototypeFactory` returns a copy of an appropriate `UserAccount` object. Figure 13.8 shows the resulting class association.

```
public class AccountPrototypeFactory {
  private UserAccount accountRep;
  private UserAccount supervisor;
  public AccountPrototypeFactory(UserAccount supervisorAccount,
      UserAccount arep) {
    accountRep = arep;
    supervisor = supervisorAccount;
  }
  public UserAccount getAccountRep() {
    return (UserAccount) accountRep.clone();
  }
  public UserAccount getSupervisor() {
    return (UserAccount) supervisor.clone();
  }
}
```

With these modifications in place, in order to create user accounts, a typical client (Listing 13.8):

Listing 13.8 Client `AccountManager` Class

```
public class AccountManager {
  public static void main(String[] args) {
    /*
       Create Prototypical Objects
    */
    Vector supervisorPermissions =
      getPermissionsFromFile("supervisor.txt");
    UserAccount supervisor = new UserAccount();
    supervisor.setPermissions(supervisorPermissions);
    Vector accountRepPermissions =
      getPermissionsFromFile("accountrep.txt");
    UserAccount accountRep = new UserAccount();
    accountRep.setPermissions(accountRepPermissions);
    AccountPrototypeFactory factory =
      new AccountPrototypeFactory(supervisor,
          accountRep);
    /* Using protype objects to create other user accounts */
    UserAccount newSupervisor = factory.getSupervisor();
    newSupervisor.setUserName("PKuchana");
    newSupervisor.setPassword("Everest");
    System.out.println(newSupervisor);
    UserAccount anotherSupervisor = factory.getSupervisor();
    anotherSupervisor.setUserName("SKuchana");
    anotherSupervisor.setPassword("Everest");
    System.out.println(anotherSupervisor);
    UserAccount newAccountRep = factory.getAccountRep();
    newAccountRep.setUserName("VKuchana");
    newAccountRep.setPassword("Vishal");
    System.out.println(newAccountRep);
  }
          …
          …
}
```

- First creates two `UserAccount` objects representing Supervisor and AccountRep type accounts. These instances are then stored inside the `AccountPrototypeFactory` as prototype objects. This is the only time permissions are read from data files.

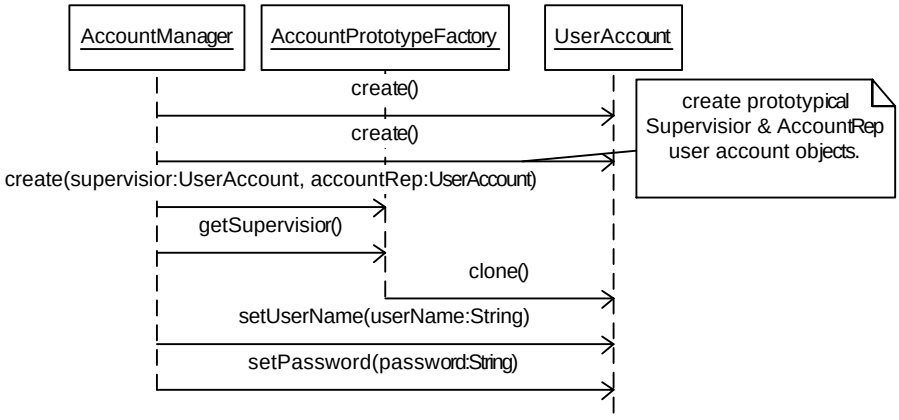

Figure 13.9 `UserAccount` Creation: Message Flow

- Each time a new Supervisor or AccountRep type account needs to be created, the client invokes one of the `getSupervisor` or the `getAccountRep` methods of the `AccountPrototypeFactory`. In response, the `AccountPrototypeFactory` clones an appropriate prototype `UserAccount` object and returns it to the client. Once the `UserAccount` clone is received, the client can make necessary changes such as setting the new username and password.

Unlike the earlier design, this approach does not involve creating each `UserAccount` object from scratch by reading from the data file. Instead, it makes use of object cloning to create new objects. The sequence diagram in Figure 13.9 depicts the message flow when a new supervisor account is created.

PRACTICE QUESTIONS

1. In the example application above, every new Supervisor type account is given exactly the same set of permissions as the prototypical Supervisor `UserAccount` object. Let us consider a new user account group to represent marketing coordinators. In addition to all the permissions of a regular supervisor, a marketing coordinator is to be given access to the color printer. Hence, whenever a marketing coordinator is to be created, the existing Supervisor prototype account object can be cloned and the required new color printer access privilege can be added. In terms of implementation, this means adding a new permission object to the permissions vector after the clone is received through the `getSupervisor` method call. In this case, is the existing shallow copy implementation, of the `clone` method sufficient, or does it need to be changed and why?

2. During the discussion of the Abstract Factory pattern, we designed an application that deals with different types of vehicles. Besides the families of vehicle classes, the application design is comprised of an abstract `VehicleFactory` with two concrete factory subclasses as listed in Table 13.1. Applying the Prototype pattern, redesign this application so that only one concrete factory class is needed. The concrete factory can be configured with the prototypical instance of each vehicle type in the vehicle family. The concrete factory then uses these prototypes to create new objects. Make any necessary assumptions about the application functionality.

Table 13.1 Concrete Factory Classes

Concrete Factory	*Responsibility*
`LuxuryVehicleFactory`	Responsible for creating instances of classes representing luxury vehicles
`NonLuxuryVehicleFactory`	Responsible for creating instances of classes representing nonluxury vehicles