Prototype

Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Uses of Prototype Pattern

- 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).

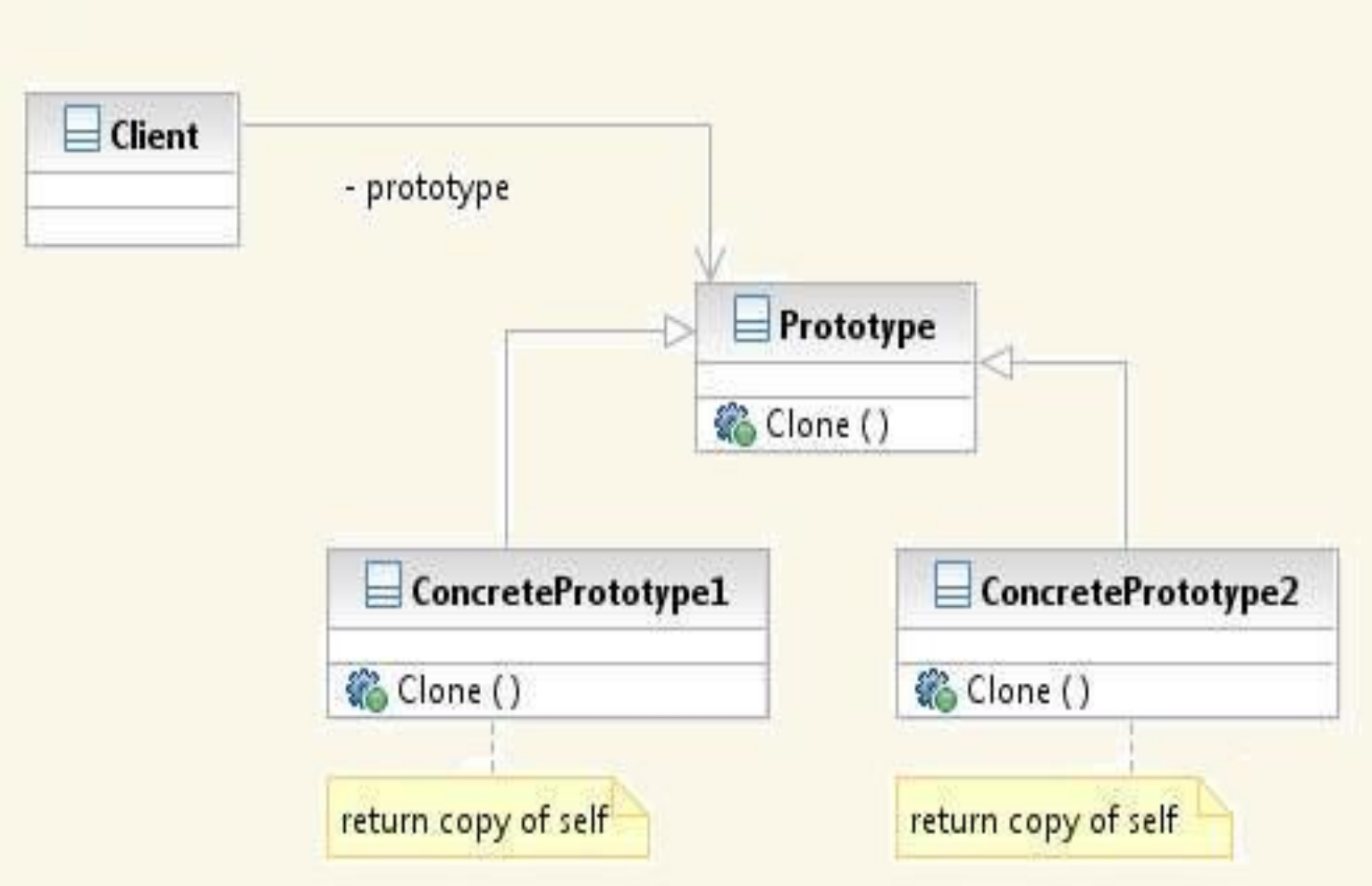
Advantages

- avoid subclasses of an object creator in the client application, like the abstract factory pattern does.
- avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.

Prototype 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.

Structure


Client
- prototype
Prototype
Clone ( )
ConcretePrototype1
Clone ( )
ConcretePrototype2
Clone ( )
return copy of self
return copy of self

Participants

- Prototype
  - Describes an interface for cloning itself
- Concrete Prototype
  - Implements an operation for cloning itself.
- Client
  - Creates a new object by asking a prototype to clone itself.

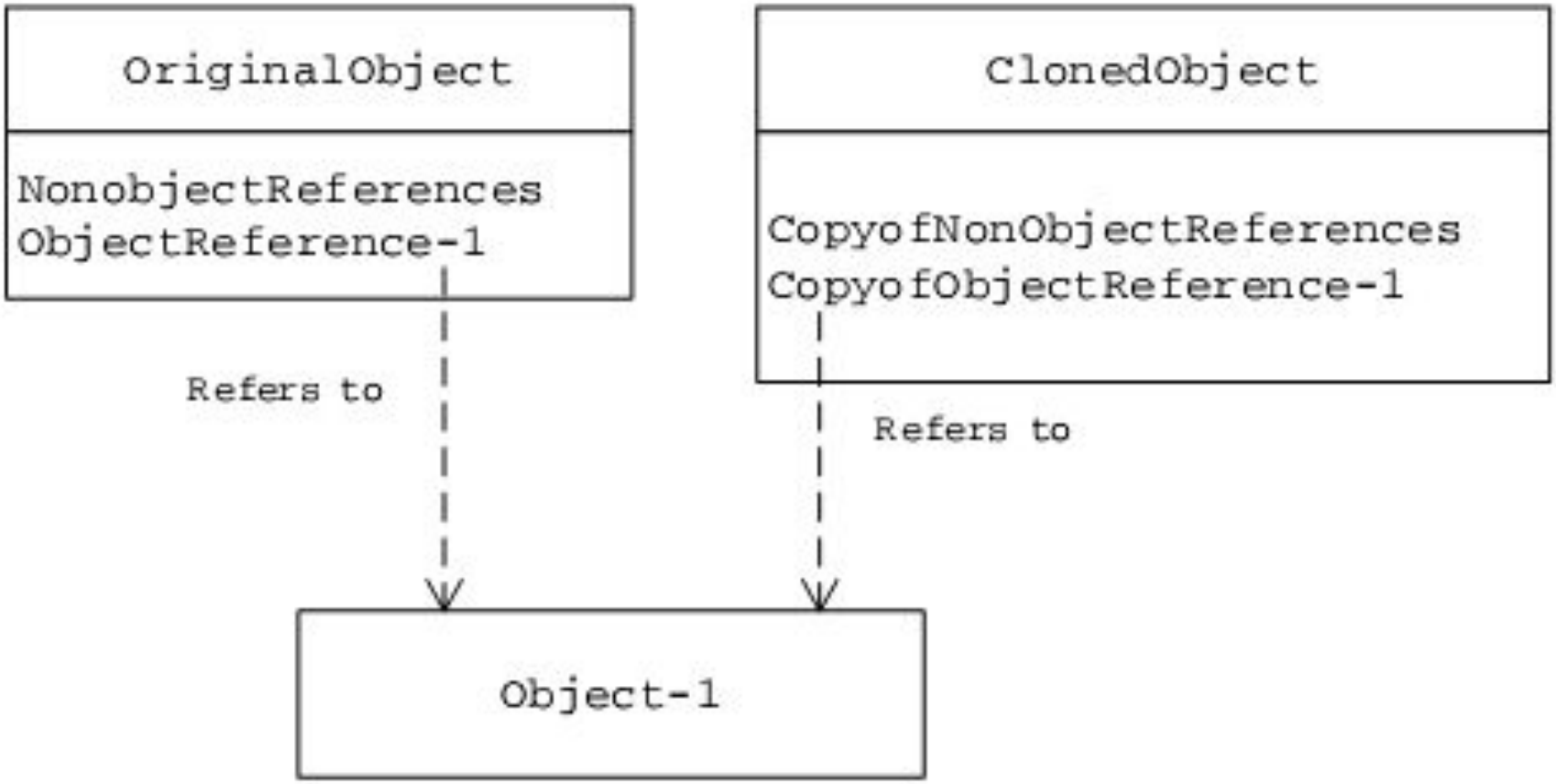
Real world examples

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.

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 original and the cloned object referring to the same copy of the lower-level object.

SHALLOW COPY



Code

```
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);
   }
```

Code

```
    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;
    }
```

Code

```
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());
  }
}
```

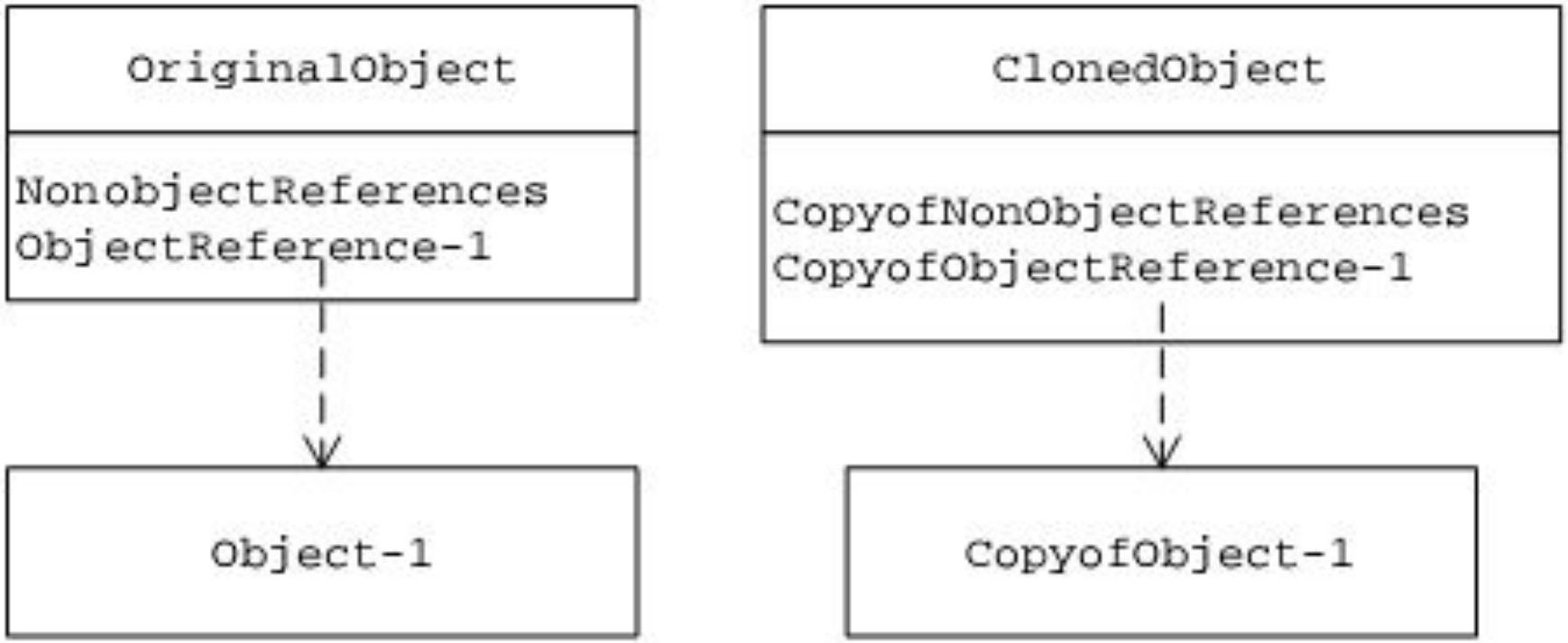
Output

```
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

- 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 original and the cloned object refer to two different lower-level objects.

DEEP COPY


OriginalObject
NonobjectReferences
ObjectReference-1
Object-1
ClonedObject
CopyofNonObjectReferences
CopyofObjectReference-1
CopyofObject-1

Code

```
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);
   }
```

Code

```
    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;
    }}
```

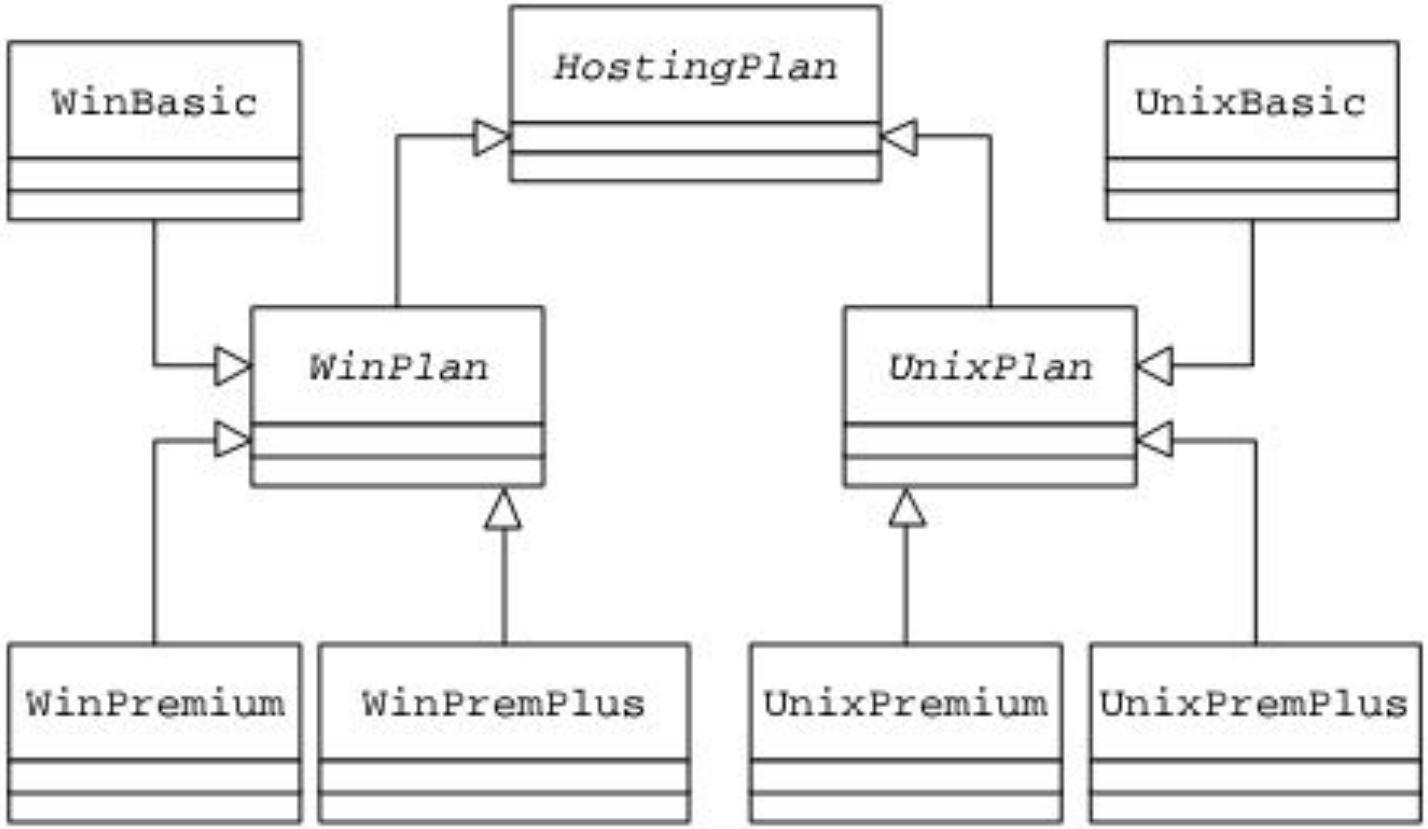
Code

```
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());
  }
}
```

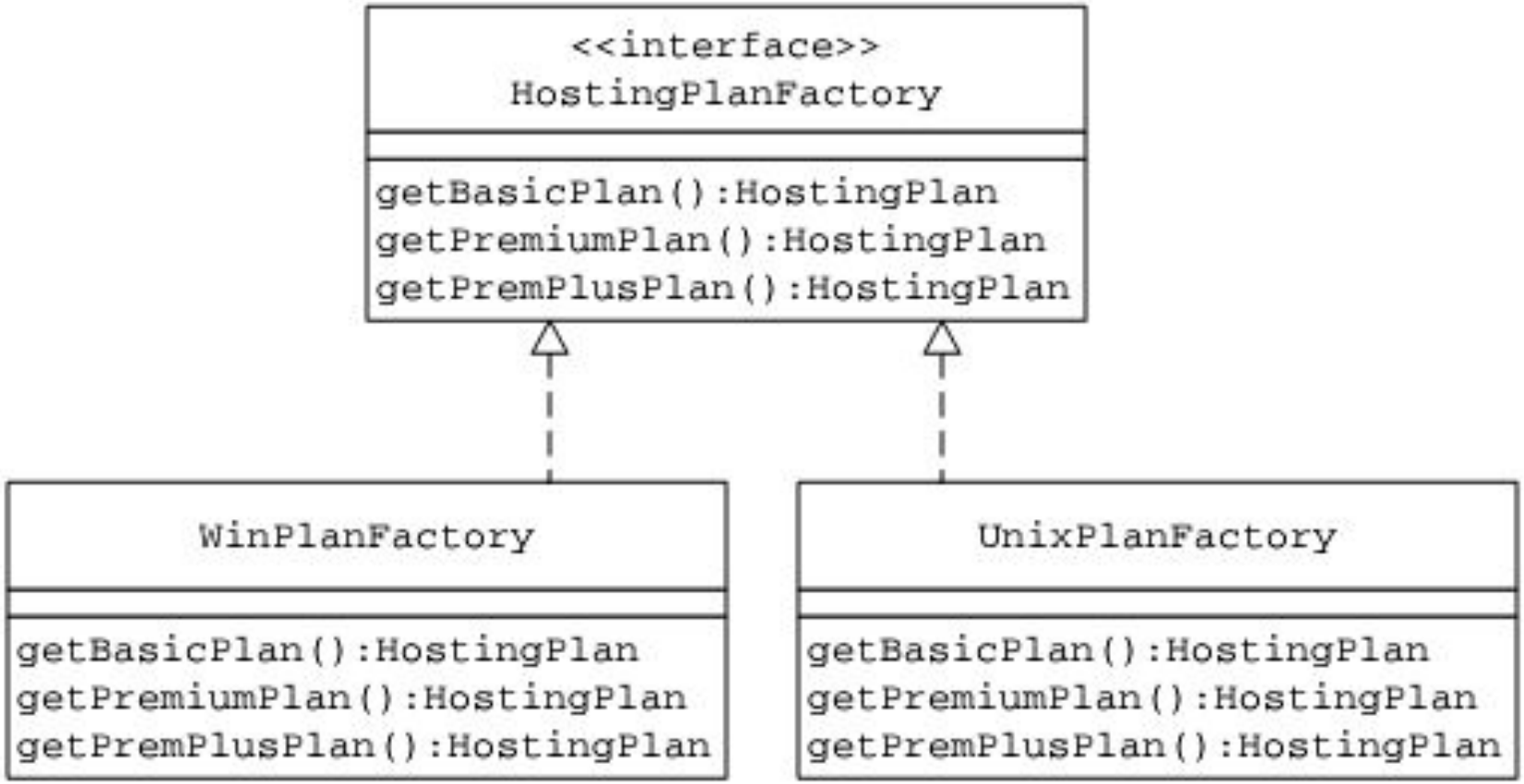
Output

```
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
Representation of different hosting packages



Applying Abstract Factory

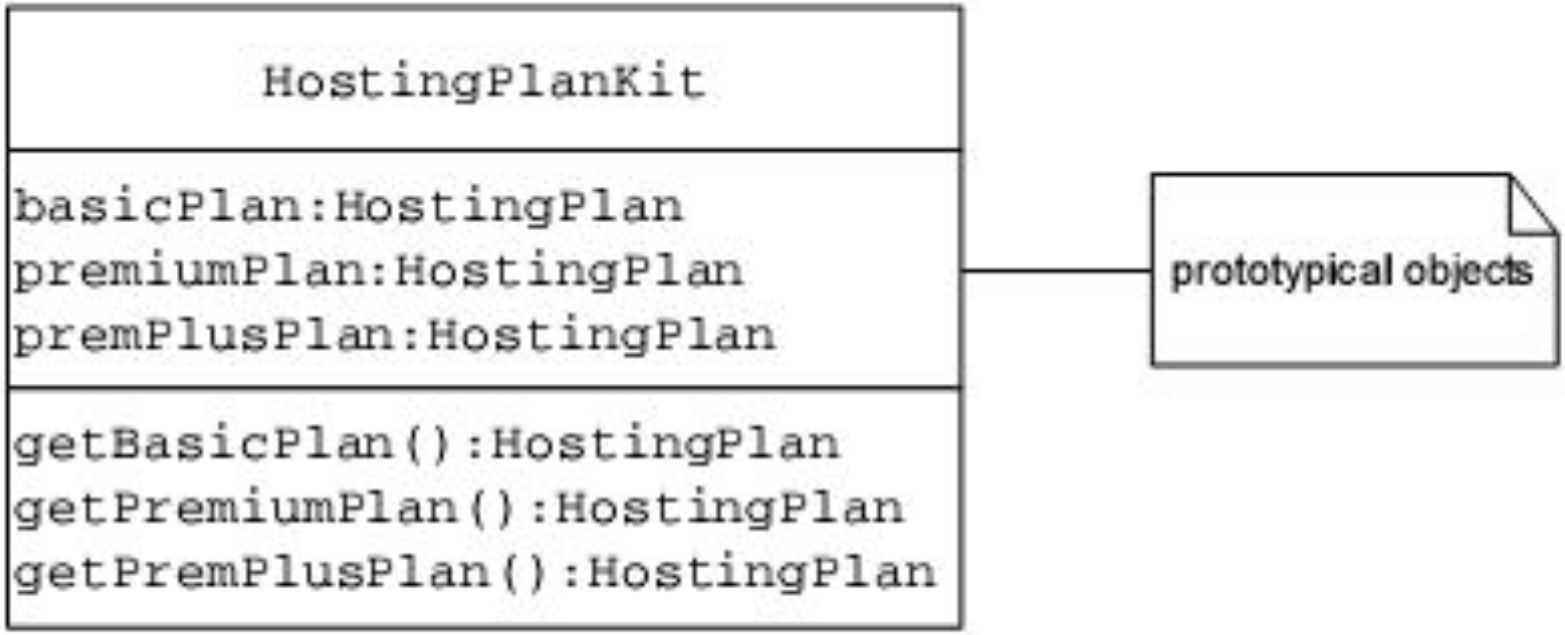


Applying Abstract Factory

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

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

Design Highlights of the HostingPlanKit Class



Design Highlights of the 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;
   }
```

Design Highlights of the HostingPlanKit Class

```
    public HostingPlan getBasicPlan() {
     return (HostingPlan) basicPlan.clone();
    }
    public HostingPlan getPremiumPlan() {
     return (HostingPlan) premiumPlan.clone();
    }
    public HostingPlan getPremPlusPlan() {
     return (HostingPlan) premPlusPlan.clone();
    }
}
```

Design Highlights of the HostingPlanManager Class

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

Design Highlights of the HostingPlanManager Class

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

Client

```
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());
  }
}
```

Usage

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

Example II

- A computer user in an organization is associated with a user account. A user account can be part of one or more groups. Permissions on different resources are defined at group level. A user gets all the permissions defined for all groups that his or her account is part of. Build an application to facilitate the creation of user accounts. Consider two groups – Supervisor and AccountRep.

```
UserAccount
---------------------------------
userName:String
password:String
fname:String
lname:String
permissions:Vector
---------------------------------
setUserName(userName:String)
setPassword(pwd:String)
setFName(fname:String)
setLName(lname:String)
setPermission(rights:Vector)
getUserName():String
getPassword():String
getFName():String
getLName():String
```

User Account 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;
  }
```

User Account Class

```
public void setPassword(String pwd) {
 password = pwd;
}
public String getPassword() {
 return password;
}
public void setFName(String name) {
 fname = name;
}
public String getFName() {
 return fname;
}
```

User Account Class

```
public void setLName(String name) {
 lname = name;
}
public String getLName() {
 return lname;
}
public void setPermissions(Vector rights) {
 permissions = rights;
}
public Vector getPermissions() {
 return permissions;
}
}
```

Simplest way to create an user account

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

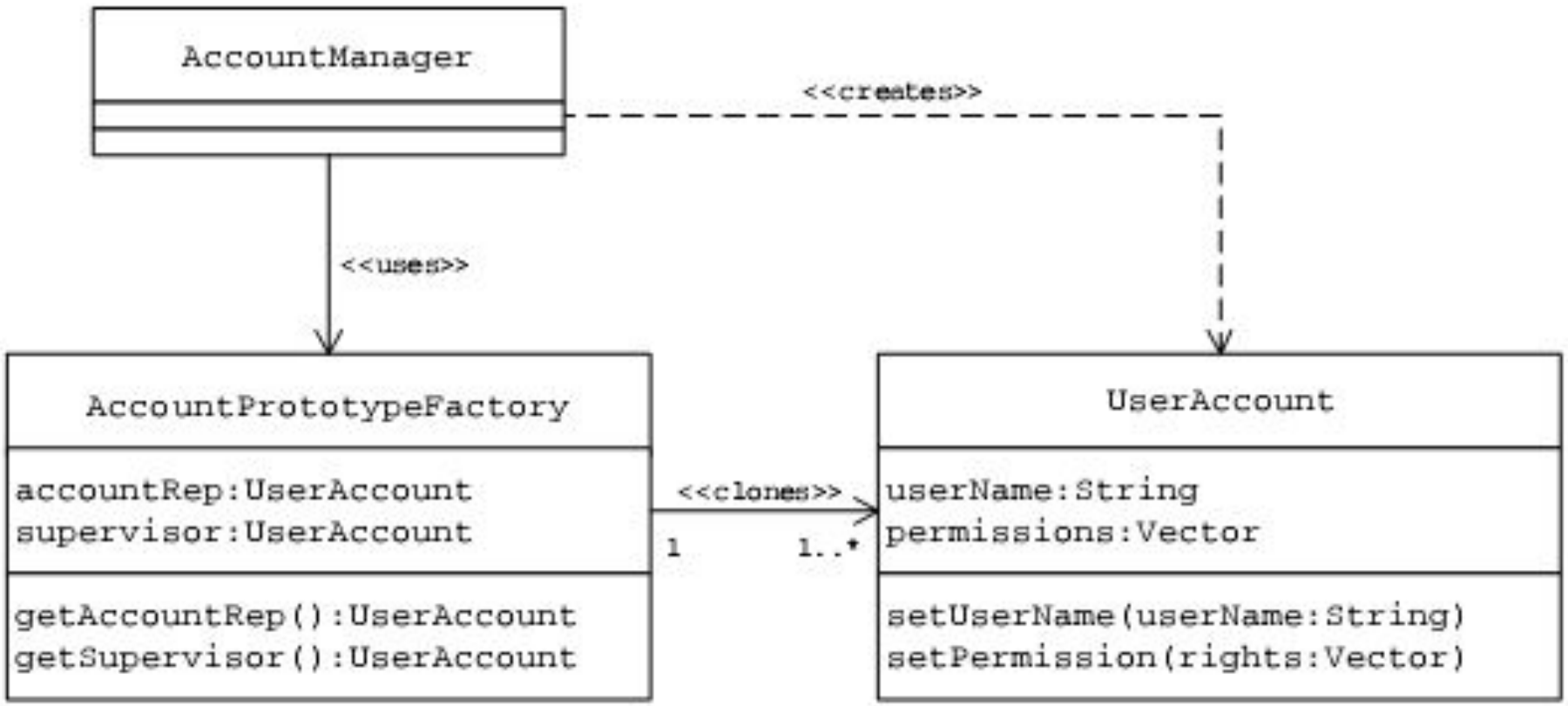
Redesign the UserAccount Class

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

Revised User Account Class

```
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;
   }   }   }
```

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.

```
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();
   }
}
```

Client AccountManager class

```
public class AccountManager {
  public static void main(String[] args) {
  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);
```

Client AccountManager class

```
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);  }   }
```

Coding Hints

1. The client object creates two UserAccount objects representing Supervisor and AccountRep type accounts. These instances are stored inside the AccountPrototypeFactory as prototype objects. This is the only time permissions are read from data files.
2. Each time a new Supervisor or AccountRep type account needs to be created, the client invokes one of the getSupervisor() or getAccountRep() methods of the AccountPrototypeFactory. The AccountPrototypeFactory clones an appropriate prototype UserAccount object and returns it to the client, which can make necessary changes such as setting the user name and password.