Prototype and Factory design patterns

Key terms

Prototype

> The prototype pattern is a creational design pattern that can be used to create objects that are similar to each other. The pattern is used to avoid the cost of creating new objects by cloning an existing object and avoiding dependencies on the class of the object that needs to be cloned.

Factory

> The factory pattern is a creational design pattern that can be used to create objects without specifying the exact class of the object that will be created. The pattern is used to avoid dependencies on the class of the object that needs to be created.

Prototype

> Prototype allows us to hide the complexity of making new instances from the client. The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations. The existing object acts as a prototype and contains the state of the object. The newly copied object may change same properties only if required. This approach saves costly resources and time, especially when object creation is a heavy process.

Let us say we have to created a new `User` API and we want to test it. To test it, we need to create a new user. We can create a new user by using the `new` keyword.

```
User user = new User("John", "Doe", "john@doe.in", "1234567890");
```

We might be calling a separate API to get these random values for the user. So each time we want to create a new user we have to call the API. Instead, we can create a new user by cloning an existing user and modifying the fields that are necessary. This way we can avoid calling the API each time we want to create a new user. To clone an existing user, we have to implement a common interface for all the user objects `clone()`

```java
public abstract class User {
    public abstract User clone();
}
```

Then we can create an initial user object which is known as the prototype and then clone it using the `clone()` method.

```java
User user = new User("John", "Doe", "john@doe.in", "1234567890");
User user2 = user.clone();
user2.setId(2);
```

Apart from reducing the cost of creating new objects, the prototype pattern also helps in reducing the complexity of creating new objects. The client code does not have to deal with the complexity of creating new objects. It can simply clone the existing object and modify it as per its needs. The client code does not have a dependency on the class of the object that it is cloning.

Prototype Registry

The prototype pattern can be extended to use a registry of pre-defined prototypes. The registry can be used to store a set of pre-defined prototypes. The client code can then request a clone of a prototype from the registry instead of creating a new object from scratch. The registry can be implemented as a key-value store where the key is the name of the prototype and the value is the prototype object.

For example, we might want to create different types of users. A user with a `Student` role, a user with a `Teacher` role, and a user with an `Admin` role. Each such different type of user might have some fields that are specific to the type so the fields to be copied might be different. We can create a registry of pre-defined prototypes for each of these roles.

```java
interface UserRegistry {
    User getPrototype(UserRole role);
    void addPrototype(UserRole role, User user);
}
```

Now we can implement the `UserRegistry` interface and store the pre-defined prototypes in a map.

```
class UserRegistryImpl implements UserRegistry {
    private Map<UserRole, User> registry = new HashMap<>();

    @Override
    public User getPrototype(UserRole role) {
        return registry.get(role).clone();
    }

    @Override
    public void addPrototype(UserRole role, User user) {
        registry.put(role, user);
    }
}
```

The client code can request a prototype from the registry, clone it, and modify it as per its needs.

```
UserRegistry registry = new UserRegistryImpl();
registry.addPrototype(UserRole.STUDENT, new Student("John", "Doe",
"john@doe.in", "1234567890", UserRole.STUDENT, "CS"));

User user = registry.getPrototype(UserRole.STUDENT);
user.setId(1);
```

Recap

- The prototype pattern is a creational design pattern that can be used to create objects that are similar to each other.
- Recreating an object from scratch can be costly as we might have to call an API to get the values for the fields or to perform some other costly operations. The prototype pattern can be used to avoid this cost by cloning an existing object and modifying the fields that are necessary.
- Also, the client code does not have to deal with the complexity of creating new objects. It can simply clone the existing object and modify it as per its needs.
- To implement the prototype pattern, we follow these steps:
  1. `Clonable interface` - Create a common interface for all the objects that can be cloned.
  2. `Object class` - Create a concrete class that implements the common interface and overrides the `clone()` method.
  3. `Registry` - Create a registry of pre-defined prototypes with `register` and `get` methods.
  4. `Prototype` - Create a prototype object and store in the registry.
  5. `Clone` - Request a clone of the prototype from the registry and modify it as per its needs.

Factory

> The factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

In the above example, the client code is still not completely independent of the class of the object that it is creating. The client code still has to call the `new` keyword to create the prototype of thh object. The client code also has to know the class of the object that it is creating. This is not ideal as the client code should not have to know the class of the object that it is creating. The client code should only know the interface of the object that it is creating. This is where the factory pattern comes into play.

```
User prototype = new User("John", "Doe");
User user = prototype.clone();
```

If we want to just change the name of the class in our next version, the client code will have to be changed making our code backward incompatible. To avoid this, we can use the factory pattern. The factory pattern allows us to create objects without specifying the exact class of the object that will be created. The client code can request an object from a factory object without having to know the class of the object that will be returned. The factory object can create the object and return it to the client code.

Simple Factory

> The simple factory pattern is a creational pattern that provides a static method for creating objects. The method can be used to create objects without having to specify the exact class of the object that will be created. This is done by creating a factory class that contains a static method for creating objects.

Let us create a simple factory class that can be used to create different types of users. The factory class will have a static method that can be used to create different types of users.

```
class UserFactory {
    public static User createUser(UserRole role) {
        switch (role) {
            case STUDENT:
                return new Student("John", "Doe");
            case TEACHER:
                return new Teacher("John", "Doe");
            case ADMIN:
                return new Admin("John", "Doe");
        }
    }
}
```

The client code can request a user object from the factory class without having to know the class of the object that will be returned.

```
User user = UserFactory.createUser(UserRole.STUDENT);
```

The complete steps to implement the simple factory pattern are:

1. `Factory class` - Create a factory class that contains a static method for creating objects.
2. `Conditional` - Use a conditional statement to create the object based on the input.
3. `Request` - Request an object from the factory class without having to know the class of the object that will be returned.

Factory Method

The simple factory method is easy to implement, but it has a few drawbacks. The factory class is not extensible. If we want to add a new type of user, we will have to modify the factory class. Also, the factory class is not reusable. If we want to create a factory for creating different types of objects, we will have to create a new factory class. To overcome these drawbacks, we can use the factory method pattern.

In the factory method the responsibility of creating the object is shifted to the child classes. The factory method is implemented in the base class and the child classes can override the factory method to create objects of their own type. The factory method is also known as the virtual constructor.

```
@AllArgsContructor
abstract class UserFactory {
    public abstract User createUser(String firstName, String lastName);
}

class StudentFactory extends UserFactory {
    @Override
    public User createUser(String firstName, String lastName) {
        return new Student(firstName, lastName);
    }
}
```

The client code can request a user object from the base class without having to know the class of the object that will be returned.

```
UserFactory factory = new StudentFactory();
User user = factory.createUser("John", "Doe");
```

The complete steps to implement the factory method pattern are:

1. `Base factory interface` - Create a factory class that contains a method for creating objects.
2. `Child factory class` - Create a child class that extends the base factory class and overrides the factory method to create objects of its own type.
3. `Request` - Request an object from the factory class without having to know the class of the object that will be returned.

Recap

- The factory pattern is a creational design pattern that can be used to create objects without having to specify the exact class of the object that will be created.
- It reduces the coupling between the client code and the class of the object that it is creating.

- Simple factory - The factory class contains a static method for creating objects. This technique is easy to implement, but it is not extensible and reusable. It violates the open-closed principle and the single responsibility principle.
- Factory method - The responsibility of creating the object is shifted to the child classes. The factory method is implemented in the base class and the child classes can override the factory method to create objects of their own type. This technique is extensible and reusable. It follows the open-closed principle and the single responsibility principle.

Design patterns in different languages

Prototype

Python

- Prototype - I
- Prototype - II
- Prototype - III
- Prototype - Code

Javascript

- Prototype - I
- Prototype - II
- Prototype - III
- Prototype - IV

Factory

Python

- Factory - I
- Factory - II
- Factory - III
- Factory - IV
- Factory - V

Javascript

- Factory - I
- Factory - II
- Factory - III
- Factory - IV
- Factory - V