Prototype design pattern

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.

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 create 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 constructor.

```
user: User = User(1, "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. Let's try to simulate with a sleep of 5 seconds.

```
class User:
    def __init__(self, id: int, first_name: str, last_name: str, email: str, ph
        self.id = id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone

        sleep(5) # Simulate API call
```

The above piece of code initializes the user object with the given values and then sleeps for 5 seconds to simulate an API call. Every time we have to create a new user, we have to call the API and wait for 5 seconds. This can be costly if we have to create a lot of users. Similarly, there can be many similar use cases where we have to create a new object from scratch, and it can be costly to do so. Few examples are:

1. Calling an API to either get the values for the fields or to perform some other costly operations.
2. Creating a large object that requires a lot of memory.
3. Creating an object that requires a lot of time to initialize.
4. Performing a lot of calculations to create an object.

The prototype pattern can be used to avoid this cost by cloning an existing object and modifying the fields that are necessary.

Cloning

Cloning an object is the process of creating a new object with the same values as the existing object. The existing object is known as the prototype. The newly created object is known as the clone. The clone can then be modified as per the needs of the client code. The prototype pattern can be used to avoid the cost of creating new objects by cloning an existing object and modifying the fields that are necessary.

There are two ways to clone an object:

1. `Shallow copy` - A shallow copy of an object copies all the fields of the object to the new object. If the field is a primitive type, then a new copy of the primitive type is created and assigned to the new object. If the field is a reference type, then the reference to the object is copied to the new object. This means that the new object and the existing object will be pointing to the same object. Any changes made to the object will be reflected in both the objects.
2. `Deep copy` - A deep copy of an object copies all the fields of the object to the new object. The difference between a shallow copy and a deep copy is that in a deep copy, a field of reference type is also cloned. This means that the new object and the

existing object will be pointing to different objects. Any changes made to the object will not be reflected in both the objects.

Difference between shallow copy and deep copy

- A shallow copy is much faster than a deep copy since it does not have to clone the fields of reference type.
- A deep copy is much slower, but it is safer since the new object and the existing object will be pointing to different objects and any changes made to the object will not be reflected in both the objects.

If you have an object with fields of primitive types, then you can use a shallow copy. If you have an object with fields of reference types and require consistency between the objects, then you can use a deep copy.

Steps to implement the prototype pattern

These are the steps to implement the prototype pattern:
Step 1 - Create a cloneable interface

Create a common base class for all the objects that can be cloned. This class will contain the `clone()` method that will be used to clone the object.

```
from abc import ABC, abstractmethod

class Cloneable(ABC):

    @abstractmethod
    def clone(self):
        pass
```

Step 2 - Implement the cloneable interface

Create a concrete class that implements the `clone()` method. These are the classes of the objects that can be cloned. For us, it is the `User` class.

```python
import copy

class User(Cloneable):
    def __init__(self, id: int, first_name: str, last_name: str, email: str, ph
        self.id = id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone

        sleep(5) # Simulate API call

    def clone(self) -> User:
        return copy.deepcopy(self)
```

The `copy` module provides a `deepcopy()` and `copy()` method that can be used to create a deep copy and a shallow copy of an object respectively.

It uses a default implementation of the `__deepcopy__()` and `__copy__()` methods to create a deep copy and a shallow copy of an object respectively. The `deepcopy()` method recursively copies all the fields of the object. The `copy()` method copies only the fields of primitive types. If the field is a reference type, then the reference to the object is copied to the new object.

You can override the `__deepcopy__()` and `__copy__()` methods to create a custom implementation of the `deepcopy()` and `copy()` methods respectively.

Step 3 - Clone the prototype and modify it

Now we can create an object of the `User` class and clone it. The clone will be a new object with the same values as the existing object. The clone can then be modified as per the needs of the client code. Here, the initial object is known as the prototype and the newly created object is known as the clone.

```python
# Create the prototype
prototype: User = User(1, "John", "Doe", "john@doe.in", "1234567890")

# Clone the prototype
clone: User = prototype.clone()

# Modify the clone
clone.id = 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

A recurring requirement would be to have a place to store all the prototypes. You would not want to create a new prototype every time you need to clone an object.

The prototype pattern can be extended to use a registry of pre-defined prototypes. The registry can be used to store a set of 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.

We start by creating a `UserRegistry` interface that contains the `get_prototype()` and `add_prototype()` methods. The `get_prototype()` method is used to get a prototype from the registry. The `add_prototype()` method is used to add a prototype to the registry. We can also add utility method `clone()` which will clone the prototype and return the clone.

```python
from abc import ABC, abstractmethod

class UserRegistry(ABC):

    @abstractmethod
    def get_prototype(self, role: str) -> Optional[User]:
        pass

    @abstractmethod
    def add_prototype(self, role: str, user: User) -> None:
        pass

    def clone(self, role: str) -> Optional[User]:
        prototype = self.get_prototype(role)
        if prototype is None:
            raise Exception(f"Prototype with role {role} not found")

        return prototype.clone()
```

Now we can implement the `UserRegistry` interface to create a concrete class that implements the `get_prototype()` and `add_prototype()` methods. This class will be used to store the prototypes in a dictionary. The key of the map will be the role of the user and the value will be the prototype object.

```python
class UserRegistryImpl(UserRegistry):
    def __init__(self):
        self.prototypes: dict[str, User] = {}

    def get_prototype(self, role: str) -> Optional[User]:
        if role not in self.prototypes:
            return None

        return self.prototypes[role]

    def add_prototype(self, role: str, user: User) -> None:
        self.prototypes[role] = user
```

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

```
# Create the registry
registry: UserRegistry = UserRegistryImpl()

# Create the prototypes
prototype: User = User(1, "John", "Doe", "john@doe.in", "1234567890")

# Add the prototypes to the registry
registry.add_prototype("student", prototype)

# Clone the prototype
clone: User = registry.clone("student")
clone.id = 2
```

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.

🖥️ Code (https://github.com/scaleracademy/lld-python/tree/main/design-patterns/src/creational/prototype)