Creational

Prototype

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Solves:

*Copying an object "from the outside" **isn't** always possible.* Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object – however not all objects can be copied as fields may be private/not visible. Additionally this creates dependency on the original object class (you have to know the object's class to create a duplicate) and you do not know the concrete class of the duplicate object (parameter in a method accepts any objects that follow some interface).

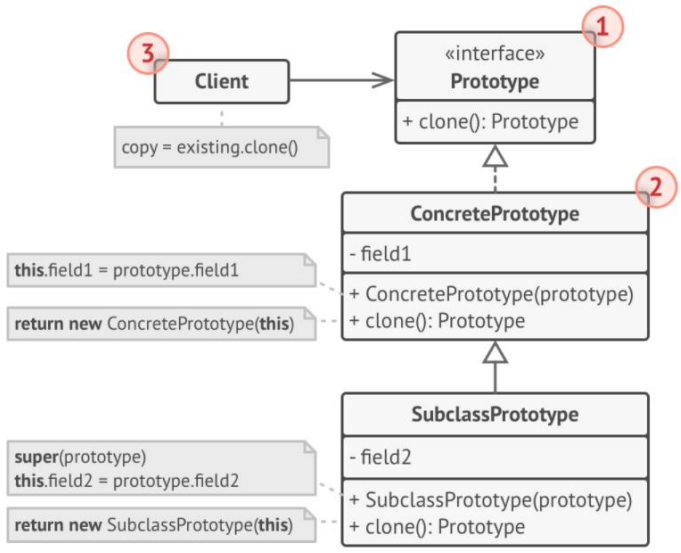
Implement:

The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The pattern declares a common interface for all objects that support cloning. This interface lets you clone an object without coupling your code to the class of that object. Usually, such an interface contains just a single `clone` method.

The implementation of the `clone` method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one. You can even copy private fields because most programming languages let objects access private fields of other objects that belong to the same class.

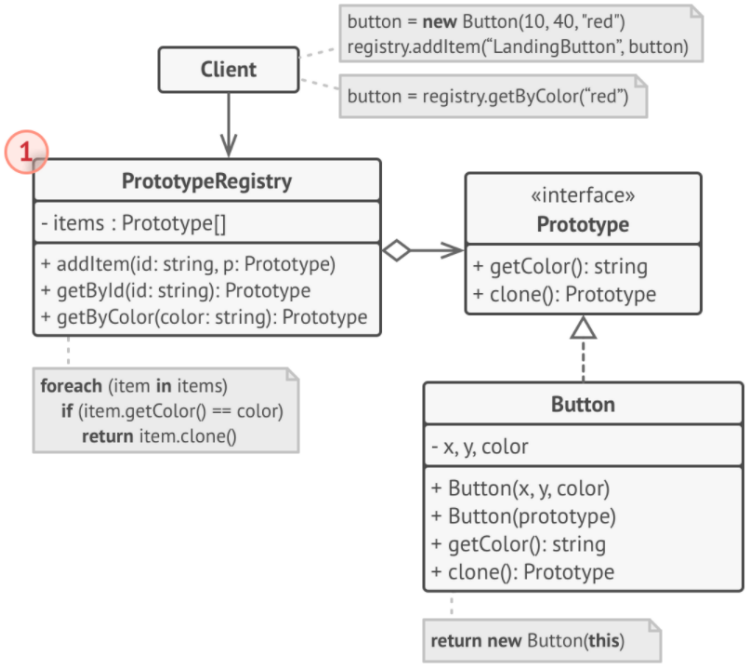
`An object that supports cloning is called a *prototype*. When your objects have dozens of fields and hundreds of possible configurations, cloning them might serve as an alternative to subclassing.

Structure:



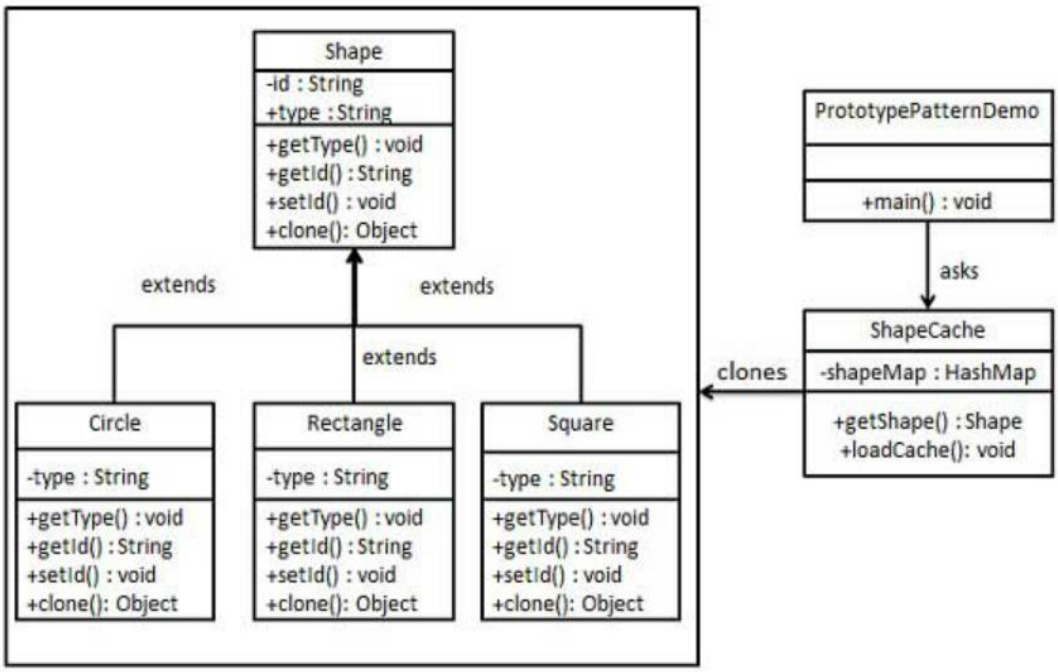
1. The **Prototype** interface declares the cloning methods. In most cases, it's a single `clone` method.
2. The **Concrete Prototype** class implements the cloning method. In addition to copying the original object's data to the clone, this method may also handle some edge cases of the cloning process related to cloning linked objects, untangling recursive dependencies, etc.
3. The **Client** can produce a copy of any object that follows the prototype interface.

Prototype registry implementation



1. The **Prototype Registry** provides an easy way to access frequently-used prototypes. It stores a set of pre-built objects that are ready to be copied. The simplest prototype registry is a `name → prototype` hash map. However, if you need better search criteria than a simple name, you can build a much more robust version of the registry.

Code structure:



Instances of Application

> **Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.**

This happens a lot when your code works with objects passed to you from 3rd-party code via some interface. The concrete classes of these objects are unknown, and you couldn't depend on them even if you wanted to.

The Prototype pattern provides the client code with a general interface for working with all objects that support cloning. This interface makes the client code independent from the concrete classes of objects that it clones.

>Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects. Somebody could have created these subclasses to be able to create objects with a specific configuration.

The Prototype pattern lets you use a set of pre-built objects, configured in various ways, as prototypes.

Instead of instantiating a subclass that matches some configuration, the client can simply look for an appropriate prototype and clone it.

How to Implement

1. Create the prototype interface and declare the `clone` method in it. Or just add the method to all classes of an existing class hierarchy, if you have one.
2. A prototype class must define the alternative constructor that accepts an object of that class as an argument. The constructor must copy the values of all fields defined in the class from the passed object into the newly created instance. If you're changing a subclass, you must call the parent constructor to let the superclass handle the cloning of its private fields.

   If your programming language doesn't support method overloading, you may define a special method for copying the object data. The constructor is a more convenient place to do this because it delivers the resulting object right after you call the `new` operator.

3. The cloning method usually consists of just one line: running a `new` operator with the prototypical version of the constructor. Note, that every class must explicitly override the cloning method and use its own class name along with the `new` operator. Otherwise, the cloning method may produce an object of a parent class.
4. Optionally, create a centralized prototype registry to store a catalog of frequently used prototypes.

   You can implement the registry as a new factory class or put it in the base prototype class with a static method for fetching the prototype. This method should search for a prototype based on search criteria that the client code passes to the method. The criteria might either be a simple string tag or a complex set of search parameters. After the appropriate prototype is found, the registry should clone it and return the copy to the client.

   Finally, replace the direct calls to the subclasses' constructors with calls to the factory method of the prototype registry.

Pros and Cons

Pro

- You can clone objects without coupling to their concrete classes.
- You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- You can produce complex objects more conveniently.
- You get an alternative to inheritance when dealing with configuration presets for complex objects.

Con

- Cloning complex objects that have circular references might be very tricky.

Relations with other pattern

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).
- **Abstract Factory** classes are often based on a set of **Factory Methods**, but you can also use **Prototype** to compose the methods on these classes.
- **Prototype** can help when you need to save copies of **Commands** into history.
- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.
- **Prototype** isn't based on inheritance, so it doesn't have its drawbacks. On the other hand, *Prototype* requires a complicated initialization of the cloned object. **Factory Method** is based on inheritance but doesn't require an initialization step.
- Sometimes **Prototype** can be a simpler alternative to **Memento**. This works if the object, the state of which you want to store in the history, is fairly straightforward and doesn't have links to external resources, or the links are easy to re-establish.
- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.