# Intent

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

# Problem

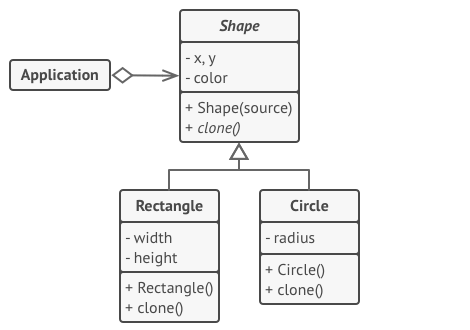
Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to create a new object of the same class. Then you have to go through all the fields of the original object and copy their values over to the new object.

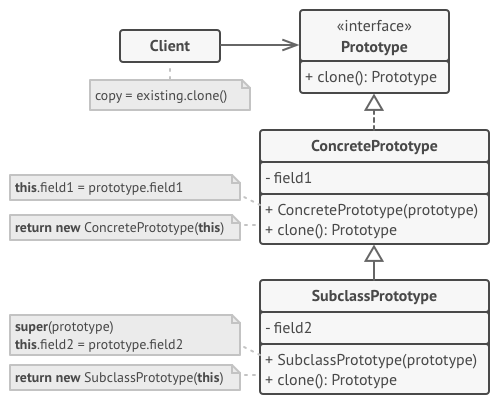
Nice! But there’s a catch. Not all objects can be copied that way because some of the object’s fields may be private and not visible from outside of the object itself.

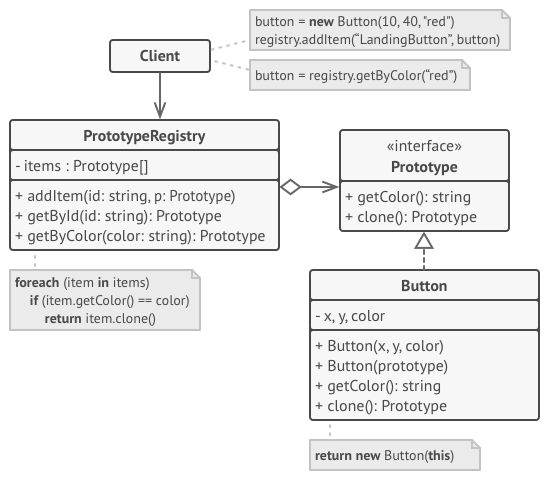
# Solution

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.

# UML







# Relations with Other Patterns

* Many designs start by using [**Factory Method**](https://refactoring.guru/design-patterns/factory-method) (less complicated and more customizable via subclasses) and evolve toward [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory), [**Prototype**](https://refactoring.guru/design-patterns/prototype), or [**Builder**](https://refactoring.guru/design-patterns/builder) (more flexible, but more complicated).
* [**Abstract Factory**](https://refactoring.guru/design-patterns/abstract-factory) classes are often based on a set of [**Factory Methods**](https://refactoring.guru/design-patterns/factory-method), but you can also use [**Prototype**](https://refactoring.guru/design-patterns/prototype) to compose the methods on these classes.
* [**Prototype**](https://refactoring.guru/design-patterns/prototype) can help when you need to save copies of [**Commands**](https://refactoring.guru/design-patterns/command) into history.
* Designs that make heavy use of [**Composite**](https://refactoring.guru/design-patterns/composite) and [**Decorator**](https://refactoring.guru/design-patterns/decorator) can often benefit from using [**Prototype**](https://refactoring.guru/design-patterns/prototype). Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.
* [**Prototype**](https://refactoring.guru/design-patterns/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**](https://refactoring.guru/design-patterns/factory-method) is based on inheritance but doesn’t require an initialization step.
* Sometimes [**Prototype**](https://refactoring.guru/design-patterns/prototype) can be a simpler alternative to [**Memento**](https://refactoring.guru/design-patterns/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**](https://refactoring.guru/design-patterns/abstract-factory), [**Builders**](https://refactoring.guru/design-patterns/builder) and [**Prototypes**](https://refactoring.guru/design-patterns/prototype) can all be implemented as [**Singletons**](https://refactoring.guru/design-patterns/singleton).