# Prototype Pattern

Making Objects by Copying

## Prototype Pattern

Creating objects from scratch can be costly.

Use a **prototype**: clone an existing object and tweak it.

**Example:** Copy–paste an essay, then edit instead of rewriting.

## The Problem

- Creating objects from scratch might be expensive or complex.

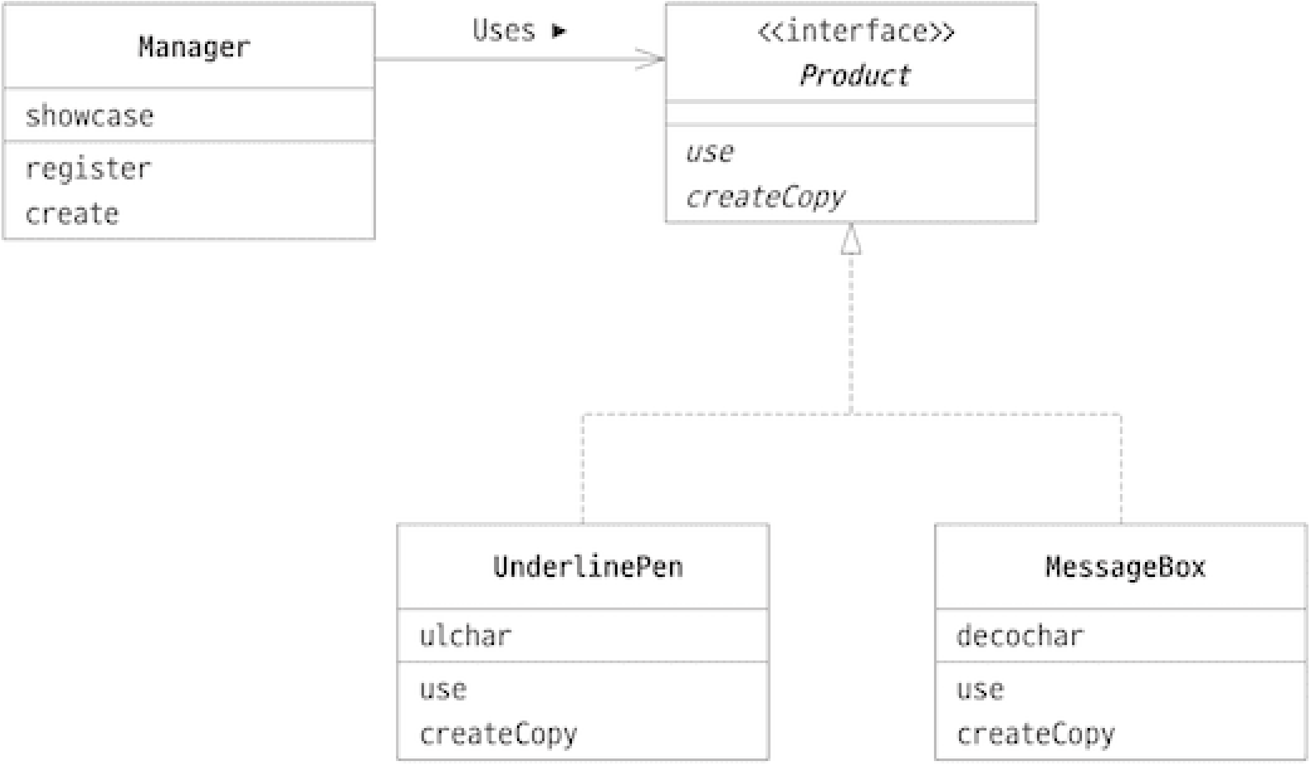
The challenge: how to create **new objects efficiently** by **copying existing ones** rather than building from scratch?

### The *Prototype* as the Solution

- Create new objects by copying this prototype.
- The prototype `delegates` the cloning process to the objects themselves.

## The Design

The createCopy() method is also known as the `clone()` method.



### Step 1: Understand the Players

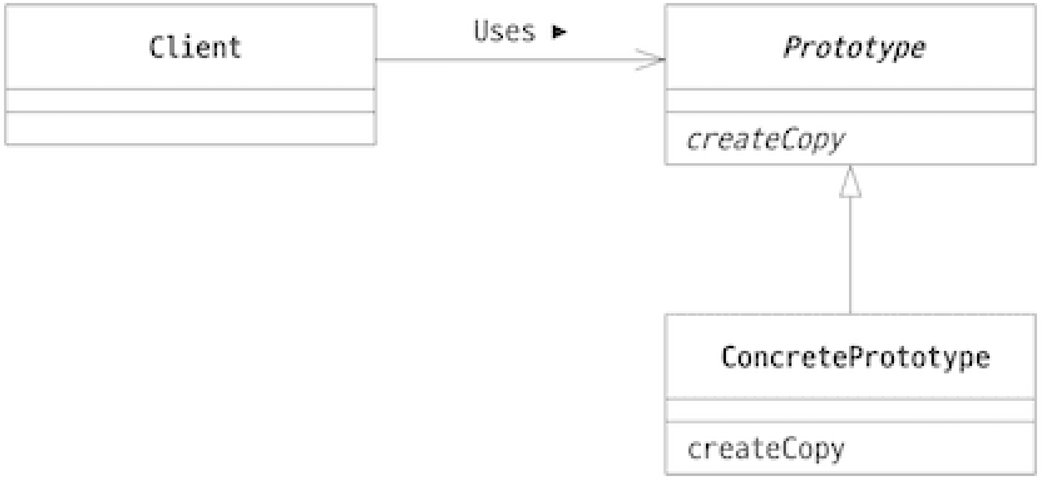
In this design, we have players:

- *Prototype* (defines cloning interface)
- **ConcretePrototype** (implements cloning)

The client/Manager uses prototypes to create objects.

- **Client/Manager**

## Step 2: Separation of *abstraction* and concretion



- **Client/Manager** works with *Prototype* interface, not concrete classes.
- **ConcretePrototypes** implement the cloning behavior.

### Step 3: Cloning Process

- Objects are responsible for cloning themselves.
- Usually implemented through a `clone()` or `createCopy()` method.
- Can use `shallow copy` or `deep copy` depending on needs.

## Code

- Main Method
- Framework (Prototype Interface & Manager)
- Concrete Prototypes (UnderlinePen & MessageBox)

# Main Method

```python
from underline_pen import UnderlinePen
from message_box import MessageBox

def main():
    upen = UnderlinePen('-')
    mbox = MessageBox('*')

    u = upen.clone()
    m = mbox.clone()

    u.use("Hello, world.")
    m.use("Hello, world.")
```

## Step 1: Create prototypes

```
upen = UnderlinePen('-')
mbox = MessageBox('*')
```

- Create prototype instances with specific configurations.

## Step 2: Clone & Use

```
    u = upen.clone()
    m = mbox.clone()

    u.use("Hello, world.")
    m.use("Hello, world.")
```

## Step 2: Product (Prototype Interface)

```python
import copy

class Product:
    def use(self, s):
        pass

    def clone(self): # clone
        try:
            return copy.deepcopy(self)
        except Exception as e:
            print(f"Error creating copy: {e}")
            return None
```

## Prototype Implementation

### UnderlinePen

```python
class UnderlinePen(Product):
    def __init__(self, ulchar):
        self.ulchar = ulchar

    def use(self, s):
        print(s)
        print(self.ulchar * len(s))
```

## MessageBox

```
class MessageBox(Product):
    def __init__(self, decochar):
        self.decochar = decochar

    def use(self, s):
        decolen = 1 + len(s) + 1
        print(self.decochar * decolen)
        print(f"{self.decochar}{s}{self.decochar}")
        print(self.decochar * decolen)
```

## Output Example

### UnderlinePen with '-':

```
Hello, world.
-------------
```

### MessageBox with '*':

```
***************
*Hello, world.*
***************
```

# Discussion

## Python Implementation Details

### Deep Copy vs Shallow Copy

```python
import copy

# Shallow copy - copies object, shares mutable references
shallow = copy.copy(original)

# Deep copy - copies object and all nested objects
deep = copy.deepcopy(original)
```

- Shallow Copy: Fast but shares mutable objects
- Deep Copy: Slower but completely independent

### Dynamic Patch

The self inside clone() refers to the actual object you called it on — in this case, the MessageBox instance m.

```
m = mbox.clone()
```

- The clone() is defined in the Product base class.
- But Python uses dynamic dispatch: the method is looked up on the class of the object you call it on (m).
- Since m is a MessageBox, self is bound to that MessageBox object, not the abstract Product.

## Circular Dependency Danger

When objects reference each other, deep copying can cause infinite recursion to cause `Circular Dependency`.

Need to track already copied objects to handle cycles properly.

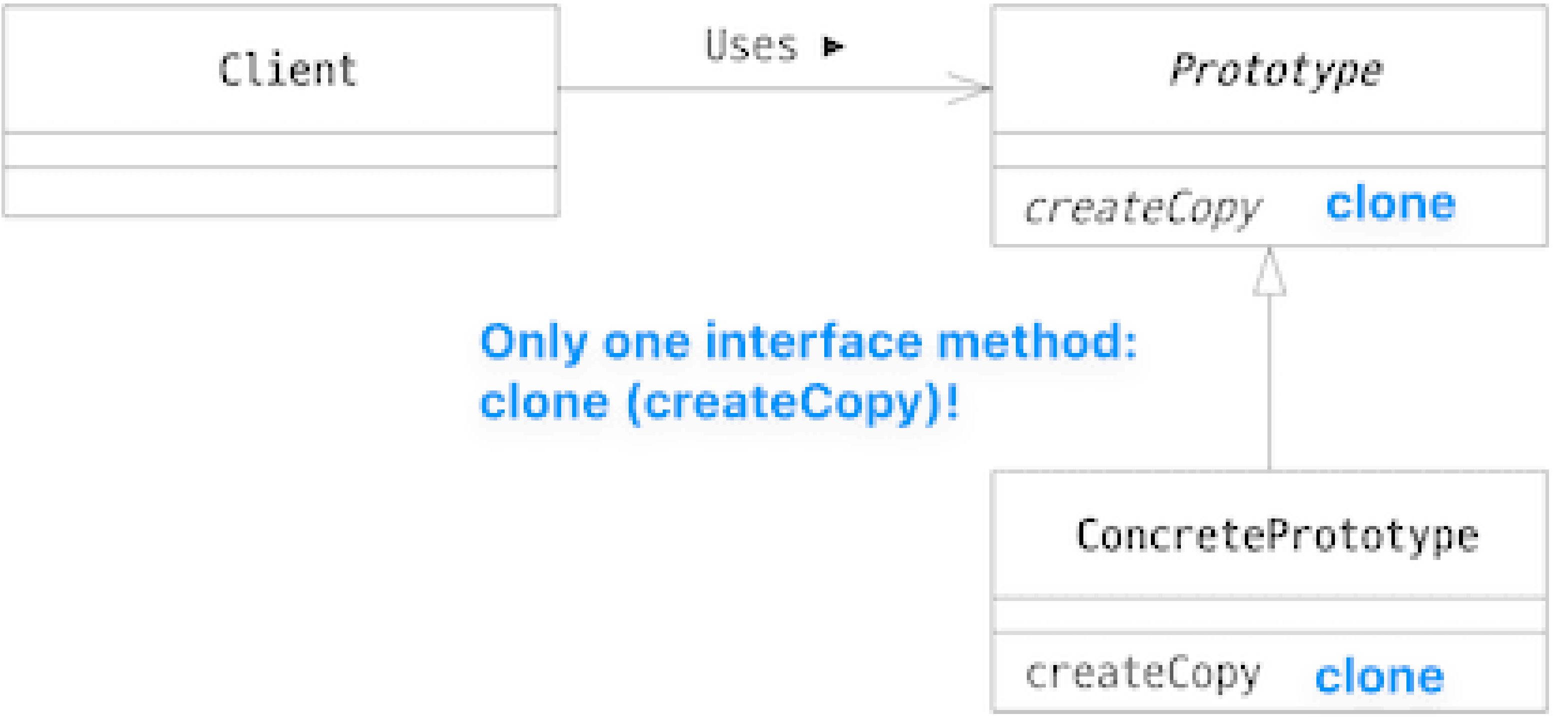
## Prototype Benefits

- **Performance**: Avoid expensive initialization
- **Simplicity**: No need to know concrete classes
- **Flexibility**: Add/remove prototypes at runtime
- **Configuration**: Pre-configured objects as templates

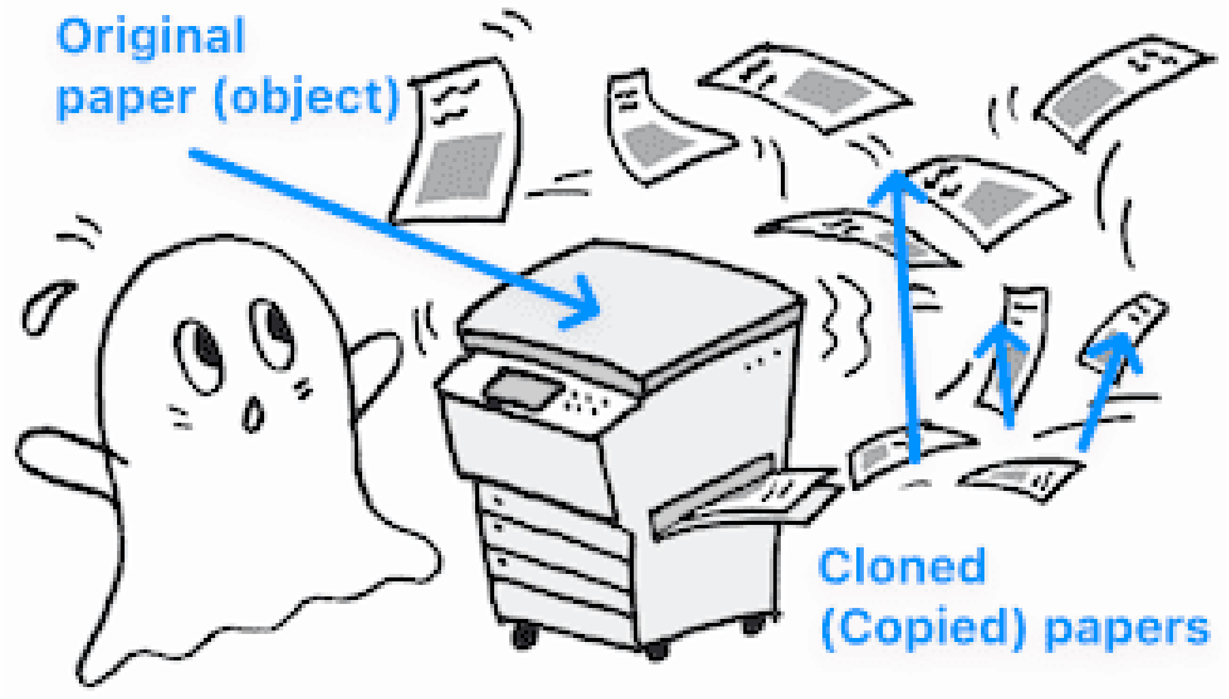
## When to Use Prototype

- Object creation is expensive (database connections, file operations)
- Objects have complex initialization
- Need many similar objects with slight variations
- Want to avoid large inheritance hierarchies

# UML



Only one interface method:
clone (createCopy)!



Original
paper (object)
Cloned
(Copied) papers