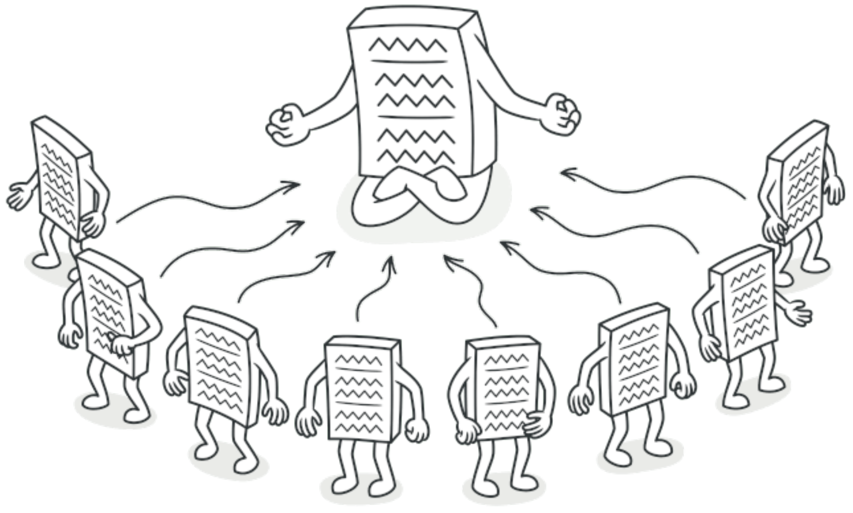
Design Patterns - Creational

Edwin Carlinet

Singleton



Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

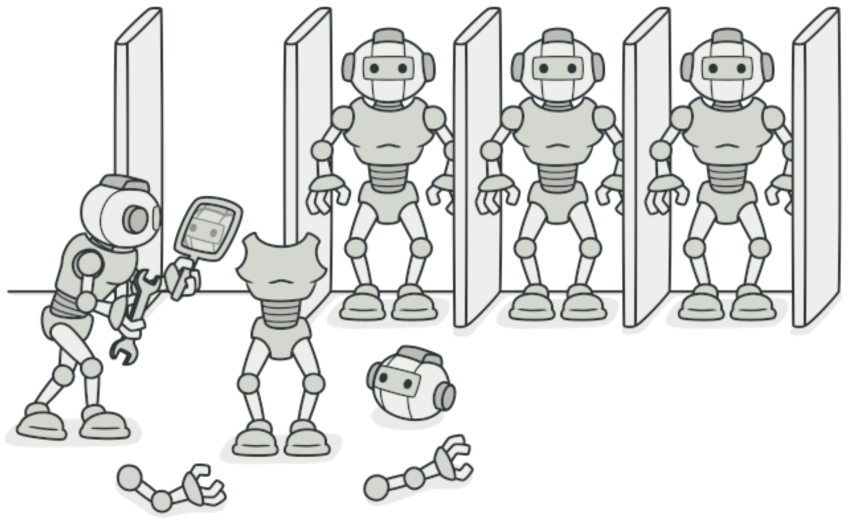
Exercise

Write a Python class `Logger` that implements the Singleton design pattern. The class has a method `log` that receives a message and prints it to the console with the timestamp. It also has a `color` attribute that defines the color of the message. Add a unit-test that tests that only one instance of the class is created throughout the program's execution.

```python
from colorama import Fore

logger1 = Logger()
logger1.log('Hello, world!')
logger2 = Logger()
logger2.color = Fore.RED
logger2.log('Hello, world!')
logger1.log('Hello, world!')
```

Prototype



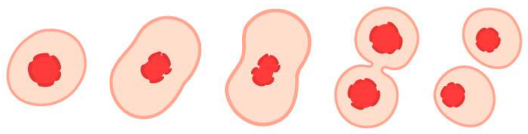
Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Exercise

You're tasked with creating a prototype for biological cells. Define an abstract class `Cell` representing the prototype for biological cells. Implement concrete subclasses `RedBloodCell` and `WhiteBloodCell` representing specific types of cells. Each cell has attributes such as `size`, and `function`, but the function is common to all instances of the same concrete class.

Draw the UML diagram for the classes.

Cellular division



We have initially a list of cells. Each cell can divide into two cells of the same type. We want to implement the method `next_generation` that returns a new list of cells after one generation where each cell has been cloned.

```python
cells = [RedBloodCell(5), WhiteBloodCell(10), RedBloodCell(6)]
next_gen = next_generation(cells)
print(cells)
print(next_gen)
```

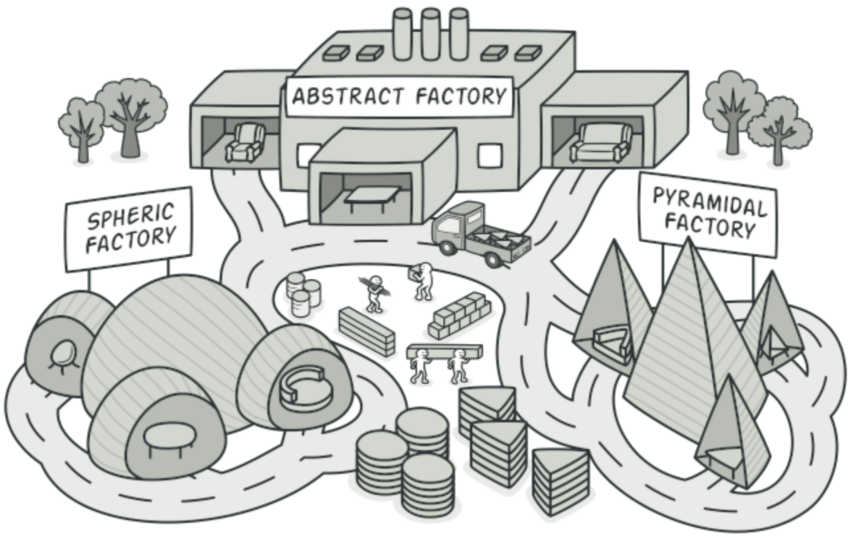
```
[RedBloodCell(size=5), WhiteBloodCell(size=10), RedBloodCell(size=6)]
[RedBloodCell(size=5), WhiteBloodCell(size=10), RedBloodCell(size=6),
 RedBloodCell(size=5), WhiteBloodCell(size=10), RedBloodCell(size=6)]
```

Prototype implementation

1. Update the uml diagram to include the method `clone` in the `Cell` class.
2. Implement the method `clone` in the `Cell` classes that returns a new instance of the same class.
3. Implement the method `next_generation` that clones each cell in the list and returns a new list of cells.

In a first version, you have to do without the deepcopy module.

Abstract Factory

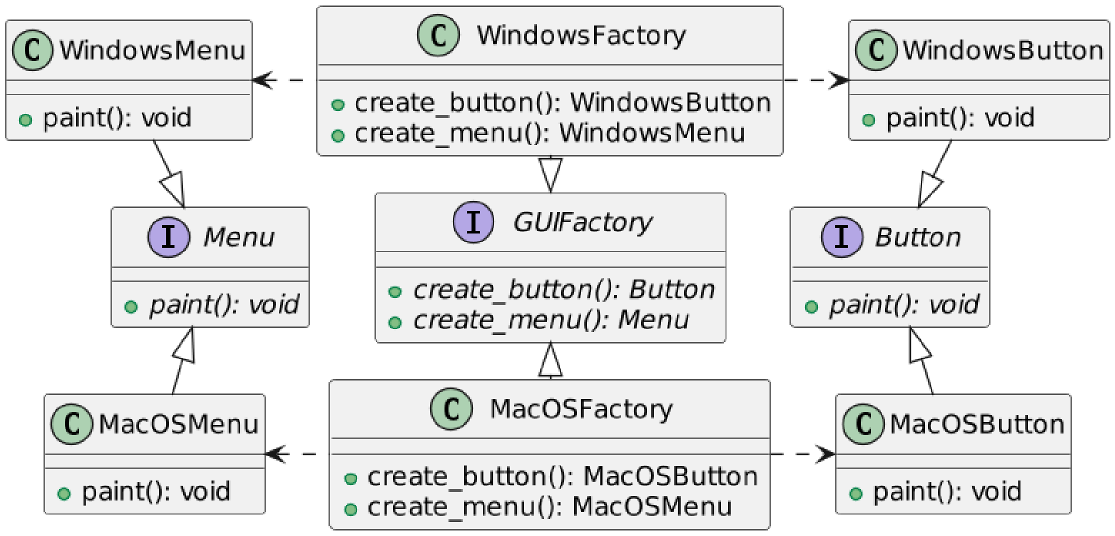


Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Problem

You're developing an application that needs to create GUI components such as buttons and menus for three different operating systems: Windows, macOS, and Linux. Implement the Factory design pattern to create a GUI Components factory for each OS so that the application is not dependent on the concrete classes of the components, neither on the OS.

```python
def application_layout(factory):
    button = factory.create_button()
    menu = factory.create_menu()
    button.paint()
    menu.paint()
```



Steps

1. Define an abstract class `GUIFactory` with methods for creating buttons and menus.
2. Implement concrete subclasses `WindowsGUIFactory`, `MacOSGUIFactory`, and `LinuxGUIFactory`, each representing a factory for creating GUI components for the respective operating system. Each subclass should override methods for creating buttons and menus, returning instances of `WindowsButton`, `MacOSButton`, `LinuxButton`, `WindowsMenu`, `MacOSMenu`, and `LinuxMenu`.
3. Define concrete classes `WindowsButton`, `MacOSButton`, and `LinuxButton`, each representing a button implementation for the respective OS. Similarly, define classes for menus (`WindowsMenu`, `MacOSMenu`, `LinuxMenu`).
4. Create an application that uses the GUI Components factory to create buttons and menus for the current operating system.