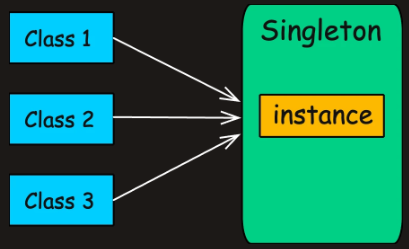
**Singleton Design Pattern**

In software development, we often require classes that can only have **one object**.

Example: thread pools, caches, loggers etc.



**What is Singleton Pattern?**

Singleton Pattern is a creational design pattern that guarantees a class has only one instance and provides a global point of access to it.

Only the singleton class should be permitted to create its own objects.

In Python, one way to implement the **Singleton** pattern is by controlling instance creation using the \_\_new\_\_ method and providing a class method (get\_instance()) to access the single instance.

* The \_instance class variable holds the one and only instance of the Singleton class.
* Python does not have private constructors, but we ensure only one instance exists by overriding \_\_new\_\_.
* The get\_instance() method is a class method that provides global access to the Singleton instance.

**Implementation**

**1. Lazy Initialization**

Creates the instance only when needed, but **not thread-safe**.

class LazySingleton:

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

*# Usage*

s1 = LazySingleton()

s2 = LazySingleton()

print(s1 is s2) *# True (same instance)*

**2. Thread-Safe Singleton (Synchronized)**

Uses a **lock** (threading.Lock) to ensure thread safety.

import threading

class ThreadSafeSingleton:

\_instance = None

\_lock = threading.Lock()

def \_\_new\_\_(cls):

with cls.\_lock:

if cls.\_instance is None:

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

*# Usage (thread-safe)*

s1 = ThreadSafeSingleton()

s2 = ThreadSafeSingleton()

print(s1 is s2) *# True*

**3. Double-Checked Locking (Optimized Thread Safety)**

Reduces locking overhead by checking twice (like Java’s volatile pattern).

import threading

class DoubleCheckedSingleton:

\_instance = None

\_lock = threading.Lock()

def \_\_new\_\_(cls):

if cls.\_instance is None: *# First check (no lock)*

with cls.\_lock: *# Acquire lock only if needed*

if cls.\_instance is None: *# Second check (locked)*

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

*# Usage (thread-safe & optimized)*

s1 = DoubleCheckedSingleton()

s2 = DoubleCheckedSingleton()

print(s1 is s2) *# True*

**4. Eager Initialization (Instant Creation)**

Creates the instance **when the class is loaded** (like Java’s static final).

class EagerSingleton:

\_instance = super().\_\_new\_\_(cls) *# Created at class definition*

def \_\_new\_\_(cls):

return cls.\_instance

*# Usage*

s1 = EagerSingleton()

s2 = EagerSingleton()

print(s1 is s2) *# True*

**5. Bill Pugh Singleton (Lazy + Thread-Safe via Inner Class)**

Uses a **nested class** to delay initialization (like Java’s SingletonHelper).

class BillPughSingleton:

class \_\_SingletonHelper:

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = super().\_\_new\_\_(cls)

return cls.\_instance

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

cls.\_instance = cls.\_\_SingletonHelper().\_instance

return cls.\_instance

*# Usage*

s1 = BillPughSingleton()

s2 = BillPughSingleton()

print(s1 is s2) *# True*

**6. Static Block Initialization (Exception Handling)**

Uses a **class-level initialization block**

class StaticBlockSingleton:

\_instance = None

def \_\_init\_\_(self):

raise RuntimeError("Use get\_instance() instead!")

@classmethod

def get\_instance(cls):

if cls.\_instance is None:

try:

cls.\_instance = super().\_\_new\_\_(cls)

except Exception as e:

raise RuntimeError("Failed to create singleton!")

return cls.\_instance

*# Usage*

s1 = StaticBlockSingleton.get\_instance()

s2 = StaticBlockSingleton.get\_instance()

print(s1 is s2) *# True*

**Real-World Examples of Singleton**

Singleton is useful in scenarios like:

* **Managing Shared Resources** (database connections, thread pools, caches, configuration settings)
* **Coordinating System-Wide Actions** (logging, print spoolers, file managers)
* **Managing State (**user session, application state**)**

**Factory Method Design Pattern**

The **Factory Method Design Pattern** is a **creational pattern** that provides an interface for creating objects in a **superclass**, but allows **subclasses** to alter the type of objects that will be created.

**1. The Problem: Sending Notifications**

Imagine you're building the backend for a web application that sends notifications to users.

At first, it’s simple. You're only sending **email notifications**.

A single class takes care of that.

class EmailNotification:

def send(self, message):

print(f"Sending Email: {message}")

class NotificationService:

def send\_notification(self, message):

email = EmailNotification()

email.send(message)

*# Usage*

service = NotificationService()

service.send\_notification("Hello via Email!")

But then comes a new requirement: support **SMS notifications**.update your NotificationService class by adding a new if block

class SMSNotification:

def send(self, message):

print(f"Sending SMS: {message}")

class NotificationService:

def send\_notification(self, notify\_type, message):

if notify\_type == "EMAIL":

EmailNotification().send(message)

elif notify\_type == "SMS":

SMSNotification().send(message)

*# Usage*

service = NotificationService()

service.send\_notification("EMAIL", "Hello via Email!")

service.send\_notification("SMS", "Hello via SMS!")

A few weeks later, product wants to send **push notifications** to mobile devices.

Then, marketing wants to experiment with **Slack alerts**. Then **WhatsApp**.

This becomes a nightmare to maintain:

* It violates key design principles, especially the **Open/Closed Principle**—the idea that classes should be open for extension but closed for modification.

**2. Clean It Up with a Simple Factory**

* You create a **separate class**(a “factory”) whose only job is to **centralize and encapsulate object creation**.
* The notification service no longer needs to know which concrete class to instantiate. It simply asks the factory for the right type of notification.

**(a) Define the Notification Interface**

from abc import ABC, abstractmethod

class Notification(ABC):

@abstractmethod

def send(self, message: str) -> None:

pass

**(b) Implement Concrete Notifications**

class EmailNotification(Notification):

def send(self, message: str) -> None:

print(f"Sending Email: {message}")

class SMSNotification(Notification):

def send(self, message: str) -> None:

print(f"Sending SMS: {message}")

class PushNotification(Notification):

def send(self, message: str) -> None:

print(f"Sending Push: {message}")

**(c) Create the Simple Factory**

class SimpleNotificationFactory:

@staticmethod

def create\_notification(notify\_type: str) -> Notification:

if notify\_type == "EMAIL":

return EmailNotification()

elif notify\_type == "SMS":

return SMSNotification()

elif notify\_type == "PUSH":

return PushNotification()

else:

raise ValueError("Unknown notification type")

**(d) Refactor NotificationService**

class NotificationService:

def send\_notification(self, notify\_type: str, message: str) -> None:

notification = SimpleNotificationFactory.create\_notification(notify\_type)

notification.send(message)

**3. What is Factory Method**

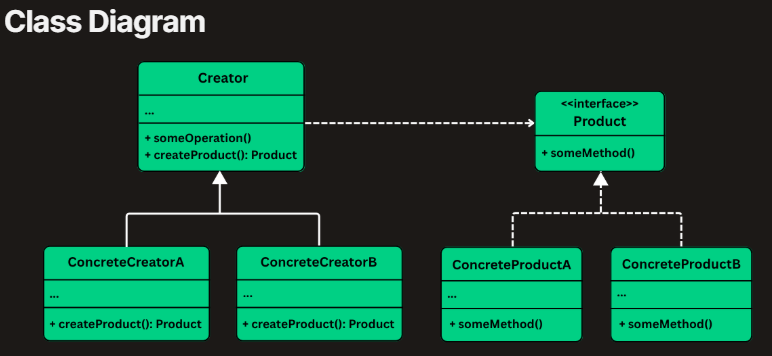
The **Factory Method Pattern** takes the idea of object creation and hands it off to **subclasses**. Instead of one central factory deciding what to create, you **delegate the responsibility to specialized classes** that know exactly what they need to produce.

**Real-World Analogy**

Think of a **food delivery platform**. You place an order. If the system is designed like a Simple Factory, there’s one centralized kitchen deciding whether to cook pizza, sushi, or burgers.

But with the Factory Method, each restaurant (Pizza Place, Sushi Bar, Burger Joint) has **its own kitchen** and knows how to prepare its food. The platform just asks the appropriate kitchen to handle it.

**Class Diagram**



**Abstract Factory Design Pattern**

**1. The Problem: Platform-Specific UI**

Imagine you're building a **cross-platform desktop application** that must support both **Windows** and **macOS**.

To provide a good user experience, your application should render **native-looking UI components** for each operating system like:

* Buttons
* Checkboxes
* Text fields
* Menus

In your first attempt, you might implement platform-specific UI components like this:

* Windows UI Elements : WindowsButton , WindowsCheckbox
* MacOS UI Elements : MacOSButton , MacOSCheckbox

Then, in your application logic, you end up doing something like:

class App:

@staticmethod

def main():

os\_name = sys.platform *# Gets the OS identifier*

if os\_name == "win32":

button = WindowsButton()

checkbox = WindowsCheckbox()

button.paint()

checkbox.paint()

elif os\_name == "darwin": *# 'darwin' is the system name for macOS*

button = MacOSButton()

checkbox = MacOSCheckbox()

button.paint()

checkbox.paint()

Problems in this approach are:

* everywhere you create UI components, you must check the OS manually.
* You can’t treat buttons or checkboxes generically. There’s no common interface like Button or Checkbox to work with.
* If I want it to suitable for linux then I should create separate button or checkbox

**2. What is Abstract Factory**

The **Abstract Factory Design Pattern** is a **creational pattern** that provides an interface for creating families of related or dependent objects without specifying their concrete classes.

It’s particularly useful in situations where:

* You need to create objects that must be **used together** and are part of a consistent family (e.g., GUI elements like buttons, checkboxes, and menus).
* Your system must support **multiple configurations**, environments, or product variants (e.g., light vs. dark themes, Windows vs. macOS look-and-feel).
* You want to **enforce consistency** across related objects, ensuring that they are all created from the same factory.

**Class Diagram**

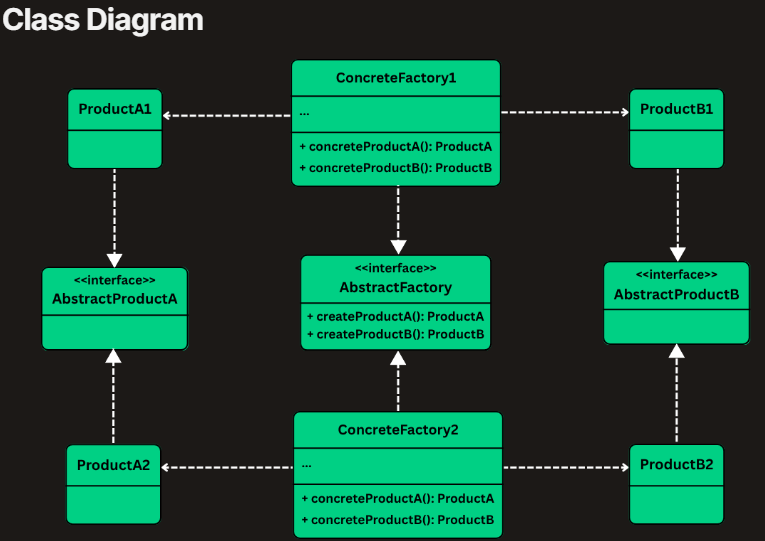
1. Abstract Factory (GUIFactory)

2. Concrete Factory (WindowsFactory, MacOSFactory)

3. Abstract Product (Button, Checkbox)

4. Concrete Product (WindowsButton, MacOSCheckbox)

5. Client (Application)

****

**3. Implementing Abstract Factory**

**1. Define Abstract Product Interfaces**

class Button(ABC):

@abstractmethod

def paint(self):

pass

@abstractmethod

def onClick(self):

pass

class Checkbox(ABC):

@abstractmethod

def paint(self):

pass

@abstractmethod

def onSelect(self):

pass

**2. Create Concrete Product Classes**

class WindowsButton(Button):

def paint(self):

print("Painting a Windows-style button.")

def onClick(self):

print("Windows button clicked.")

class WindowsCheckbox(Checkbox):

def paint(self):

print("Painting a Windows-style checkbox.")

def onSelect(self):

print("Windows checkbox selected.")

class MacOSButton(Button):

def paint(self):

print("Painting a macOS-style button.")

def onClick(self):

print("MacOS button clicked.")

class MacOSCheckbox(Checkbox):

def paint(self):

print("Painting a macOS-style checkbox.")

def onSelect(self):

print("MacOS checkbox selected.")

**3. Define the Abstract Factory**

class GUIFactory(ABC):

@abstractmethod

def createButton(self) -> Button:

pass

@abstractmethod

def createCheckbox(self) -> Checkbox:

pass

**4. Implement Concrete Factories**

class WindowsFactory(GUIFactory):

def createButton(self) -> Button:

return WindowsButton()

def createCheckbox(self) -> Checkbox:

return WindowsCheckbox()

class MacOSFactory(GUIFactory):

def createButton(self) -> Button:

return MacOSButton()

def createCheckbox(self) -> Checkbox:

return MacOSCheckbox()

**5. Client Code – Use Abstract Interfaces Only**

class Application:

def \_\_init\_\_(self, factory: GUIFactory):

self.button = factory.createButton()

self.checkbox = factory.createCheckbox()

def renderUI(self):

self.button.paint()

self.checkbox.paint()

**6. Application Entry Point**

class AppLauncher:

@staticmethod

def main():

*# Simulate platform detection*

os\_name = sys.platform

factory: GUIFactory

if os\_name == "win32":

factory = WindowsFactory()

else: *# Assuming macOS for other cases*

factory = MacOSFactory()

app = Application(factory)

app.renderUI()

**Builder Design Pattern**

**1. The Problem: Building Complex HttpRequest Objects**

A system that needs to **configure and create HTTP requests.**

Each HttpRequest can contain a mix of required and optional fields depending on use case.

class HttpRequestTelescoping:

def \_\_init\_\_(self,

url: str,

method: str = "GET",

headers: Optional[Dict[str, str]] = None,

query\_params: Optional[Dict[str, str]] = None,

body: Optional[str] = None,

timeout: int = 30000):

self.url = url

self.method = method

self.headers = headers if headers is not None else {}

self.query\_params = query\_params if query\_params is not None else {}

self.body = body

self.timeout = timeout

**What’s Wrong with This Approach?**

* Multiple parameters of the same type (e.g., String, Map) make it easy to accidentally swap arguments.
* You must follow the exact parameter order, which hurts readability and usability.

**2. What is the Builder Pattern**

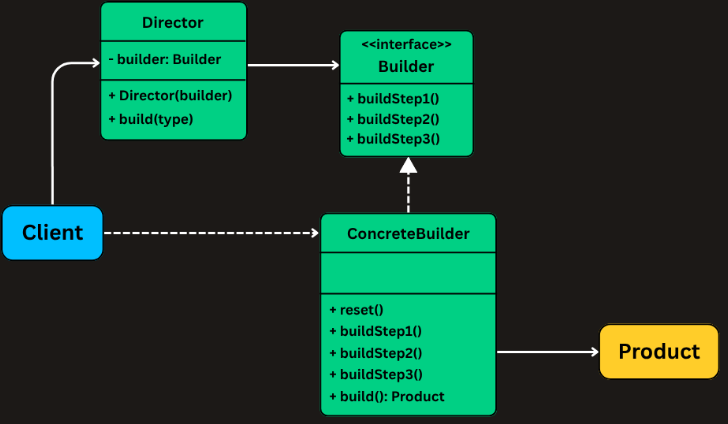
The Builder pattern separates the construction of a complex object from its representation.

In the Builder Pattern:

* The construction logic is encapsulated in a Builder.
* The final object (the "Product") is created by calling a build() method.

**Class Diagram**

* Builder (e.g., HttpRequestBuilder)
* ConcreteBuilder (e.g., StandardHttpRequestBuilder)
* Product (e.g., HttpRequest)
* Director (Optional) (e.g., HttpRequestDirector)

****

**1. Create the Product Class (HttpRequest)**

class HttpRequest:

"""The product class with a private constructor"""

def \_\_init\_\_(self, builder: 'HttpRequest.Builder'):

self.url = builder.url

def \_\_str\_\_(self):

# print those attributes

*# Builder Class*

class Builder:

"""The builder class with fluent interface"""

def \_\_init\_\_(self, url: str):

self.url = url

**2. Using the Builder from Client Code**

class HttpAppBuilderPattern:

@staticmethod

def main():

request1 = HttpRequest.Builder("https://api.example.com/data").build()

**Prototype Design Pattern**

**The Challenge of Cloning Objects**

**Problem 1: Encapsulation Gets in the Way**

This approach assumes that all fields of the object are publicly accessible. But in a well-designed system, many fields are private and hidden behind encapsulation. That means your cloning logic can’t access them directly.

**Problem 2: Class-Level Dependency**

Even if you could access all the fields, you'd still need to know the concrete class of the object to instantiate a copy.

**The Problem: Spawning Enemies in a Game**

Let’s say you’re developing a 2D shooting game where enemies appear frequently throughoutthe gameplay.

You have several enemy types with distinct attributes: BasicEnemy , ArmoredEnemy and Flying Enemy

Each enemy type comes with predefined properties such as:Health , Speed , Armor , Weapon type and Sprite or appearance

Now, imagine you need to spawn a FlyingEnemy. You might write code like this:

Enemy flying1 = new Enemy("Flying", 100, 10.5, false, "Laser");

Enemy flying2 = new Enemy("Flying", 100, 10.5, false, "Laser");

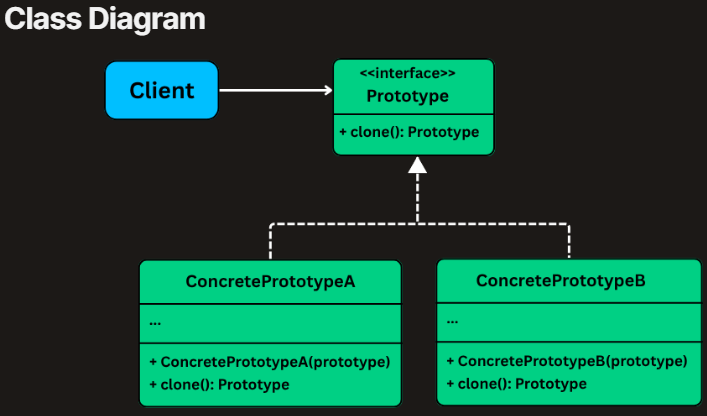
And you’ll do the same for dozens, maybe hundreds, of similar enemies during the game.

But Here’s the Problem

* Repetitive Code
* Scattered Defaults: If the default speed or weapon of FlyingEnemy changes, you need to update it in every single place you created one.

**The Prototype Design Pattern**

The Prototype pattern specifies the kinds of objects to create using a prototypical instance and creates new objects by copying (cloning) this prototype.



* **Define a Prototype Interface**: This declares the clone() method, which every cloneable object must implement.
* **Implement Concrete Prototypes**: Each object that needs to be cloneable (e.g., FlyingEnemy, ArmoredEnemy) implements the Prototype interface and provides logic to **clone itself**.
* **Client Requests a Clone:** When a client needs a new object (e.g., enemy), it doesn’t new it directly — it simply **asks the prototype object to copy itself**.

**Implementing Prototype**

**Step 1: Define the Prototype Interface (EnemyPrototype)**

class EnemyPrototype(ABC):

@abstractmethod

def clone(self) -> 'EnemyPrototype':

pass

**Step 2: Create the Concrete Prototype Class (Enemy)**

class Enemy(EnemyPrototype):

def \_\_init\_\_(self,

enemy\_type: str,

health: int,

speed: float,

armored: bool,

weapon: str):

self.type = enemy\_type

self.health = health

self.speed = speed

self.armored = armored

self.weapon = weapon

def clone(self) -> 'Enemy':

*# Creates a new instance with the same attributes (shallow copy)*

return Enemy(self.type, self.health, self.speed,

self.armored, self.weapon)

def set\_health(self, health: int) -> None:

self.health = health

def print\_stats(self) -> None:

print(f"{self.type} [Health: {self.health}, "

f"Speed: {self.speed}, "

f"Armored: {self.armored}, "

f"Weapon: {self.weapon}]")

**Step 3: Create a Prototype Registry (EnemyRegistry)**

class EnemyRegistry:

def \_\_init\_\_(self):

self.prototypes: Dict[str, Enemy] = {}

def register(self, key: str, prototype: Enemy) -> None:

self.prototypes[key] = prototype

def get(self, key: str) -> Enemy:

prototype = self.prototypes.get(key)

if prototype:

return prototype.clone()

raise ValueError(f"No prototype registered for: {key}")

**Step 4: Using the Registry in Your Game**