**Adapter Design Pattern**

The **Adapter Design Pattern** is a **structural design pattern** that allows incompatible interfaces to work together by **converting the interface of one class into another that the client expects**.

**1. The Problem: Incompatible Payment Interfaces**

Imagine you’re building the checkout component of an **e-commerce**application.

Your **Checkout Service** is designed to work with a **Payment Interface** for handling payments.

**The Expected Interface**

Here’s the contract your CheckoutService expects any payment provider to follow:

class PaymentProcessor(ABC):

@abstractmethod

def process\_payment(self, amount: float, currency: str) -> None:

pass

@abstractmethod

def is\_payment\_successful(self) -> bool:

pass

@abstractmethod

def get\_transaction\_id(self) -> str:

pass

**Your In-House Implementation**

Your team already has an internal payment processor that fits this interface perfectly:

class InHousePaymentProcessor(PaymentProcessor):

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

self.transaction\_id = None

self.is\_payment\_successful = False

def process\_payment(self, amount: float, currency: str) -> None:

print(f"InHousePaymentProcessor: Processing payment of {amount} {currency}")

self.transaction\_id = f"TXN\_{int(time.time() \* 1000)}"

self.is\_payment\_successful = True

print(f"InHousePaymentProcessor: Payment successful. Txn ID: {self.transaction\_id}")

def is\_payment\_successful(self) -> bool:

return self.is\_payment\_successful

def get\_transaction\_id(self) -> str:

return self.transaction\_id

Your CheckoutService uses this interface and works beautifully with the in-house payment processor:

class CheckoutService:

def \_\_init\_\_(self, payment\_processor: PaymentProcessor):

self.payment\_processor = payment\_processor

def checkout(self, amount: float, currency: str) -> None:

self.payment\_processor.process\_payment(amount, currency)

if self.payment\_processor.is\_payment\_successful():

pass

else:

pass

Here’s how it gets called from your main e-commerce application:

class ECommerceAppV1:

@staticmethod

def main():

processor = InHousePaymentProcessor()

checkout = CheckoutService(processor)

checkout.checkout(199.99, "USD")

Now, management drops a new requirement: integrate with a **legacy third-party payment provider**, widely used and battle-tested… but with a completely different interface.

You now have two incompatible interfaces. Your existing CheckoutService **expects** a PaymentProcessor. But LegacyGateway does **not** implement it and it’s methods and signatures don't match:

* processPayment(double) vs. executeTransaction(double, String)
* isPaymentSuccessful() vs. checkStatus(long)
* getTransactionId() vs. getReferenceNumber() (and their types are different too!)

**2. What is the Adapter Pattern**

The Adapter acts as a **bridge between an incompatible interface and what the client actually expects**.

**Two Types of Adapters**

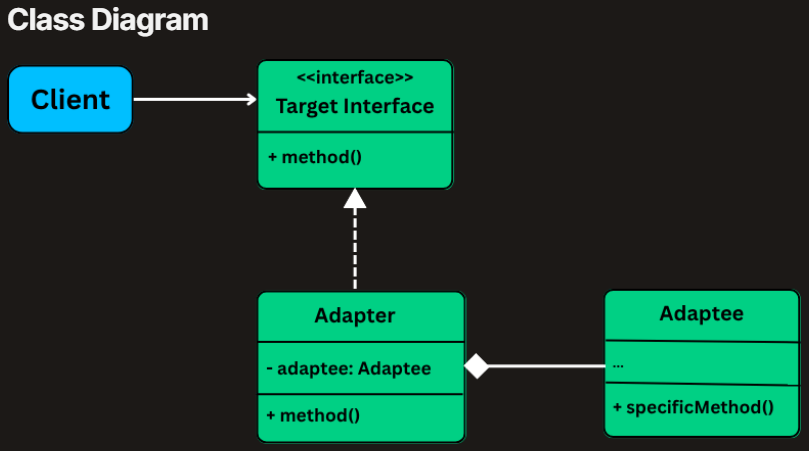
There are two primary ways to implement an adapter, depending on the language and use case:

**1. Object Adapter**

* Uses **composition**: the adapter holds a reference to the adaptee (the object it wraps).
* Allows flexibility and reuse across class hierarchies.
* This is the most common and recommended approach in Java.

**2. Class Adapter**

* Uses **inheritance**: the adapter inherits from both the target interface and the adaptee.
* Requires **multiple inheritance**, which Java doesn’t support for classes.
* More suitable for languages like **C++**.



* **Target Interface (e.g.,**PaymentProcessor**)**: The interface that the client code expects and uses.
* **Adaptee (e.g.,**LegacyGateway**)**: The existing class with an incompatible interface that needs adapting.
* **Adapter**: The class that implements the Target interface and uses the Adaptee internally. It translates calls on the Target interface into calls on the Adaptee's interface.
* **Client (e.g.,**CheckoutService**)**: The part of your system that uses the Target interface.

**3. Implementing Adapter**

To integrate the legacy LegacyGateway class into our modern e-commerce system, we’ll create an **object adapter** called LegacyGatewayAdapter.

**The Adapter Implementation**

class LegacyGatewayAdapter(PaymentProcessor):

def \_\_init\_\_(self, legacy\_gateway: LegacyGateway):

self.legacy\_gateway = legacy\_gateway

self.current\_ref = None *# Store the current transaction reference*

def process\_payment(self, amount: float, currency: str) -> None:

self.legacy\_gateway.execute\_transaction(amount, currency)

self.current\_ref = self.legacy\_gateway.get\_reference\_number()

def is\_payment\_successful(self) -> bool:

return self.legacy\_gateway.check\_status(self.current\_ref)

def get\_transaction\_id(self) -> str:

return f"LEGACY\_TXN\_{self.current\_ref}"

**Client Code**

class ECommerceAppV2:

@staticmethod

def main():

processor = InHousePaymentProcessor()

modern\_checkout = CheckoutService(processor)

print("--- Using Modern Processor ---")

modern\_checkout.checkout(199.99, "USD")

print("\n--- Using Legacy Gateway via Adapter ---")

legacy = LegacyGateway()

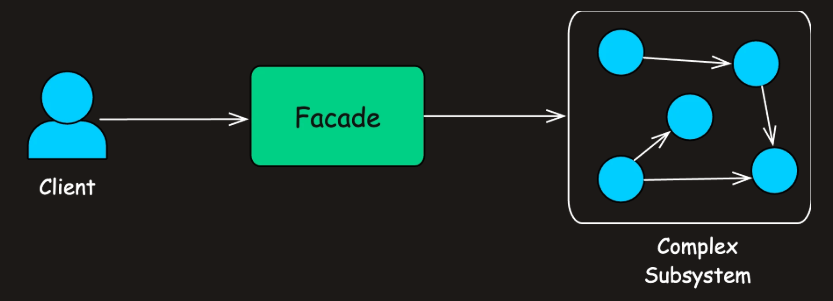
processor = LegacyGatewayAdapter(legacy)

legacy\_checkout = CheckoutService(processor)

legacy\_checkout.checkout(75.50, "USD")

**Facade**

The **Facade Design Pattern** is a **structural design pattern** that provides a unified, simplified interface to a complex subsystem.



**The Problem: Deployment Complexity**

Here’s a simplified version of a typical deployment flow:

1. **Pull the latest code** from a Git repository
2. **Build the project** using a tool like Maven or Gradle
3. **Run automated tests** (unit, integration, maybe end-to-end)
4. **Deploy the build** to a production environment

**Deployment Subsystems**

**1. Version Control System**

Handles interaction with Git or another VCS. Responsible for fetching the latest code.

class VersionControlSystem:

def pull\_latest\_changes(self, branch: str) -> None:

self.\_simulate\_delay()

def \_simulate\_delay(self) -> None:

time.sleep(1)

**2. Build System**

Compiles the codebase, creates an artifact (like a .jar), and returns its location.

class BuildSystem:

def compile\_project(self) -> bool:

print("BuildSystem: Compiling project...")

self.\_simulate\_delay(2)

print("BuildSystem: Build successful.")

return True

def get\_artifact\_path(self) -> str:

path = "target/myapplication-1.0.jar"

print(f"BuildSystem: Artifact located at {path}")

return path

def \_simulate\_delay(self, seconds: float) -> None:

time.sleep(seconds)

**3. Testing Framework**

Executes unit and integration tests. Could also include E2E, mutation testing, or security scans in real-world setups.

class TestingFramework:

def run\_unit\_tests(self) -> bool:

def run\_integration\_tests(self) -> bool:

**4. Deployment Target**

Handles artifact delivery to the server and version activation.

class DeploymentTarget:

def transfer\_artifact(self, artifact\_path: str, server: str) -> None:

print(f"Deployment: Transferring {artifact\_path} to {server}...")

def activate\_new\_version(self, server: str) -> None:

print(f"Deployment: Now live on {server}!")

**The Orchestrator**

The DeploymentOrchestrator is the component trying to coordinate everything. It pulls in all subsystems and defines the **exact sequence of operations** to perform a deployment.

class DeploymentOrchestrator:

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

self.vcs = VersionControlSystem()

self.build\_system = BuildSystem()

self.test\_framework = TestingFramework()

self.deploy\_target = DeploymentTarget()

def deploy\_application(self, branch: str, prod\_server: str) -> bool:

print(f"\n[Orchestrator] Starting deployment for branch: {branch}")

self.vcs.pull\_latest\_changes(branch)

if not self.build\_system.compile\_project():

print("Build failed. Deployment aborted.", file=sys.stderr)

return False

artifact = self.build\_system.get\_artifact\_path()

if not self.test\_framework.run\_unit\_tests() or not self.test\_framework.run\_integration\_tests():

print("Tests failed. Deployment aborted.", file=sys.stderr)

return False

self.deploy\_target.transfer\_artifact(artifact, prod\_server)

self.deploy\_target.activate\_new\_version(prod\_server)

print("[Orchestrator] Deployment successful!")

return True

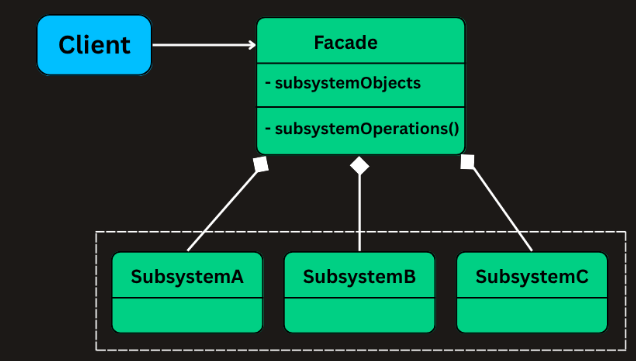
**What’s Wrong with This Design?**

The DeploymentOrchestrator — which essentially acts as your "client" — must be aware of every subsystem:

* What methods to call
* In what sequence
* What to do on success or failure

**The Facade Design Pattern**

The Facade Pattern introduces a **high-level interface** that hides the complexities of one or more subsystems and exposes only the functionality needed by the client.



**Facade (e.g., DeploymentFacade)**

Knows **which subsystem classes** to use and **in what order**. Delegates requests to appropriate subsystem methods without exposing internal details to the client.

**Subsystem Classes (e.g., VersionControlSystem, BuildSystem)**

Provides the actual business logic to handle a specific task. **Do not know** about the facade. Can still be used independently if needed.

**Client (e.g., our main application or a script)**

Uses the Facade to initiate a deployment, instead of interacting with the subsystem classes directly.

**Implementing Façade**

Rather than forcing the client to call each of these subsystems in the correct order, the facade **abstracts this coordination logic** and offers a **clean, high-level method** like deployApplication() that executes the full workflow.

class DeploymentFacade:

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

self.vcs = VersionControlSystem()

self.build\_system = BuildSystem()

self.testing\_framework = TestingFramework()

self.deployment\_target = DeploymentTarget()

def deploy\_application(self, branch: str, server\_address: str) -> bool:

success = True

try:

self.vcs.pull\_latest\_changes(branch)

if not self.build\_system.compile\_project():

return False

artifact\_path = self.build\_system.get\_artifact\_path()

if not self.testing\_framework.run\_unit\_tests():

return False

if not self.testing\_framework.run\_integration\_tests():

return False

self.deployment\_target.transfer\_artifact(artifact\_path, server\_address)

self.deployment\_target.activate\_new\_version(server\_address)

except Exception as e:

success = False

return success

*# Supporting classes (same as before but included for completeness)*

class VersionControlSystem:

def pull\_latest\_changes(self, branch: str) -> None:

class BuildSystem:

def compile\_project(self) -> bool:

def get\_artifact\_path(self) -> str:

class TestingFramework:

def run\_unit\_tests(self) -> bool:

def run\_integration\_tests(self) -> bool:

class DeploymentTarget:

def transfer\_artifact(self, artifact\_path: str, server: str) -> None:

def activate\_new\_version(self, server: str) -> None:

**Decorator**

The **Decorator Design Pattern** is a **structural pattern** that lets you **dynamically add new behavior or responsibilities to objects** without modifying their underlying code.

**1. The Problem: Adding Features to a Text Renderer**

Imagine you’re building a **rich text rendering system**. At the core of your system is a TextView component that renders plain text on screen.

Soon, product requirements evolve:You need to support **bold** text , Then **italic** text and Then **underlined** text and so on.

**Naive Approach: Subclassing for Every Combination**

**class** TextView(ABC):

**class** BoldTextView(TextView):

**class** ItalicTextView(TextView): and so on.

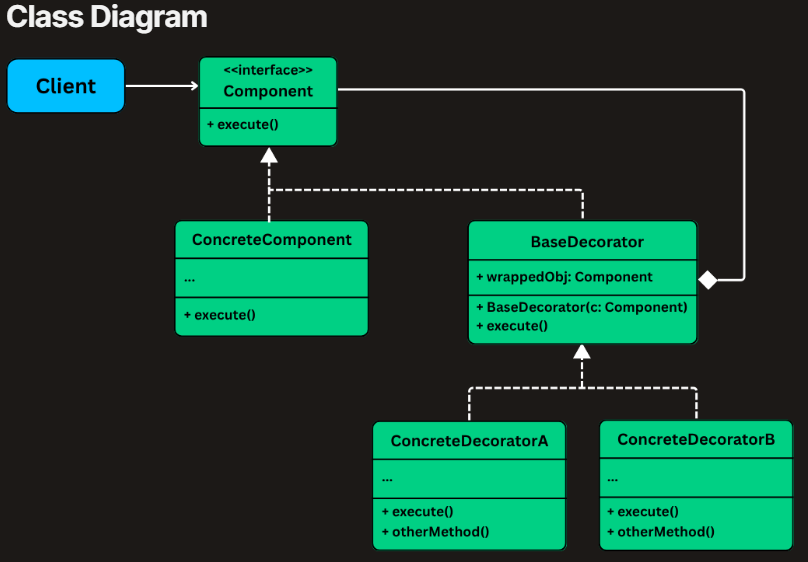
**Why This Approach Quickly Falls Apart**

**1. Class Explosion**

For every new combination of features, you need to create a new subclass

**2. The Decorator Pattern**

At its core, the pattern relies on **wrapping** an object inside another object (a **decorator**) that implements the **same interface** and adds new behavior before or after delegating to the wrapped object.



* **Component (e.g.,** TextView): Declares the common interface (execute()) for all core and decorated objects
* **ConcreteComponent (e.g.,** PlainTextView): The base object that can be dynamically decorated
* **BaseDecorator (abstract):**Implements the component interface and stores a reference to the component to be decorated
* **ConcreteDecorators** (BoldDecorator, ItalicDecorator, etc.): Extend the base decorator to add new functionality before/after calling the wrapped component’s method

**3. Implementing Decorator Pattern**

**1. Define the Component Interface**

class TextView(ABC):

@abstractmethod

def render(self) -> None:

pass

**2. Implement the Concrete Component**

class PlainTextView(TextView):

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

self.text = text

def render(self) -> None:

print(self.text, end='')

**3. Create the Abstract Decorator**

class TextDecorator(TextView):

def \_\_init\_\_(self, inner: TextView):

self.inner = inner

**4. Implement Concrete Decorators**

class BoldDecorator(TextDecorator):

def render(self) -> None:

print("<b>", end='')

self.inner.render()

print("</b>", end='')

class ItalicDecorator(TextDecorator):

def render(self) -> None:

print("<i>", end='')

self.inner.render()

print("</i>", end='')

class UnderlineDecorator(TextDecorator):

def render(self) -> None:

print("<u>", end='')

self.inner.render()

print("</u>", end='')

**5. Compose Decorators in Client Code**

if \_\_name\_\_ == "\_\_main\_\_":

text = PlainTextView("Hello, world!")

print("Plain: ", end='')

text.render()

print()

print("Bold: ", end='')

bold\_text = BoldDecorator(text)

bold\_text.render()

print()

print("Italic + Underline: ", end='')

italic\_underline = UnderlineDecorator(ItalicDecorator(text))

italic\_underline.render()

print()

print("Bold + Italic + Underline: ", end='')

all\_styles = UnderlineDecorator(ItalicDecorator(BoldDecorator(text)))

all\_styles.render()

print()

**Composite**

The **Composite Design Pattern** is a **structural pattern** that lets you **treat individual objects and compositions of objects uniformly**.

It allows you to build tree-like structures (e.g., file systems, UI hierarchies, organizational charts) where **clients can work with both single elements and groups of elements using the same interface.**

**1. The Problem: Modeling a File Explorer**

 The system needs to represent:

* **Files** – simple items that have a name and a size.
* **Folders** – containers that can hold files **and** other folders (even nested folders).

support operations such as:

* getSize() – return the total size of a file or folder (which is the sum of all contents).
* printStructure() – print the name of the item, including indentation to show hierarchy.
* delete() – delete a file or a folder and everything inside it.

**The Naive Approach**

class File:

def \_\_init\_\_(self, name: str, size: int):

self.name = name

self.size = size

def get\_size(self) -> int:

return self.size

def print\_structure(self, indent: str = "") -> None:

print(f"{indent}{self.name}")

def delete(self) -> None:

print(f"Deleting file: {self.name}")

class Folder:

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

self.name = name

self.contents: List[Union[File, 'Folder']] = []

def get\_size(self) -> int:

total = 0

for item in self.contents:

if isinstance(item, File):

total += item.get\_size()

elif isinstance(item, Folder):

total += item.get\_size()

return total

def print\_structure(self, indent: str = "") -> None:

print(f"{indent}{self.name}/")

for item in self.contents:

if isinstance(item, File):

item.print\_structure(indent + " ")

elif isinstance(item, Folder):

item.print\_structure(indent + " ")

def delete(self) -> None:

for item in self.contents:

if isinstance(item, File):

item.delete()

elif isinstance(item, Folder):

item.delete()

print(f"Deleting folder: {self.name}")

**What’s Wrong With This Approach?**

**1. Repetitive Type Checks**

Operations like getSize(), printStructure(), and delete() require repeated instanceof checks and downcasting — leading to duplicated and fragile logic.

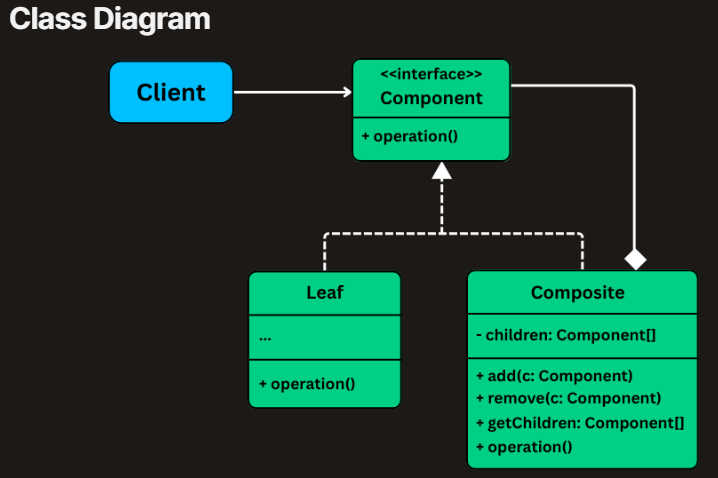
**2. No Shared Abstraction**

There’s no common interface for File and Folder, which means you can’t treat them uniformly.

**2. What is the Composite Pattern**

In a composite structure, each node in the hierarchy shares the same interface, whether it’s a **leaf** (e.g., a File) or a **composite** (e.g., a Folder).

This allows clients to perform operations like getSize(), delete(), or render() **recursively and consistently** across both.



* **Component Interface (e.g., FileSystemItem)**

Declares the common interface for all concrete components

* **Leaf (e.g., File)**

Represents an end object (no children)

* **Composite (e.g., Folder)**

Represents an object that can hold children (including other composites)

**3. Implementing Composite**

**1. Define the Component Interface**

class FileSystemItem(ABC):

@abstractmethod

def get\_size(self) -> int:

pass

@abstractmethod

def print\_structure(self, indent: str) -> None:

pass

@abstractmethod

def delete(self) -> None:

pass

**2. Create the Leaf Class – File**

class File(FileSystemItem):

def \_\_init\_\_(self, name: str, size: int):

self.name = name

self.size = size

def get\_size(self) -> int:

return self.size

def print\_structure(self, indent: str) -> None:

print(f"{indent}- {self.name} ({self.size} KB)")

def delete(self) -> None:

print(f"Deleting file: {self.name}")

**3. Create the Composite Class – Folder**

class Folder(FileSystemItem):

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

self.name = name

self.children: List[FileSystemItem] = []

def add\_item(self, item: FileSystemItem) -> None:

self.children.append(item)

def get\_size(self) -> int:

return sum(item.get\_size() for item in self.children)

def print\_structure(self, indent: str) -> None:

print(f"{indent}+ {self.name}/")

for item in self.children:

item.print\_structure(indent + " ")

def delete(self) -> None:

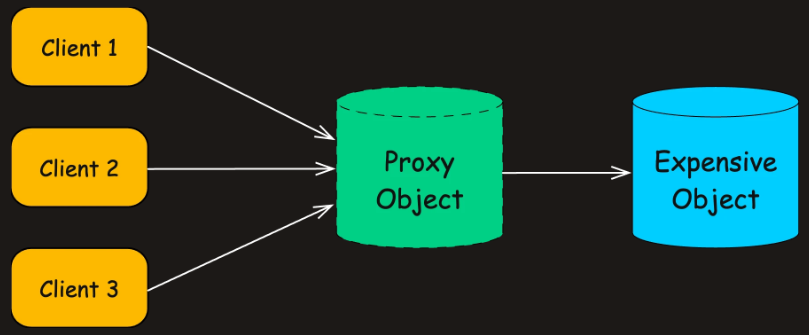
for item in self.children:

item.delete()

print(f"Deleting folder: {self.name}")

**Proxy**

Provides a **placeholder or surrogate** for another object, allowing you to **control access** to it.



A **proxy** sits between the client and the real object, intercepting calls and optionally altering the behavior.

**1. The Problem: Eager Loading**

Imagine you're building an **image gallery application**. Users can scroll through a list of image thumbnails, and when they click on one, the **full high-resolution image** is displayed.

**1. The Image Interface**

class Image(ABC):

def display(self) -> None:

def get\_file\_name(self) -> str:

**2. High-Resolution Image Implementation**

class HighResolutionImage(Image):

def \_\_init\_\_(self, file\_name: str):

self.file\_name = file\_name

self.image\_data = bytearray()

self.\_load\_image\_from\_disk() *# Expensive operation!*

def \_load\_image\_from\_disk(self) -> None:

print(f"Loading image: {self.file\_name} from disk (Expensive Operation)..")

time.sleep(2) *# Simulate disk I/O delay*

self.image\_data = bytearray(10 \* 1024 \* 1024) *# Simulate 10MB memory usage*

print(f"Image {self.file\_name} loaded successfully.")

def display(self) -> None:

print(f"Displaying image: {self.file\_name}")

def get\_file\_name(self) -> str:

return self.file\_name

**3. The Naive Gallery App**

class ImageGalleryAppV1:

@staticmethod

def main() -> None:

print("Application Started. Initializing images for gallery...")

*# Images are created eagerly - loaded even if not viewed!*

image1 = HighResolutionImage("photo1.jpg")

image2 = HighResolutionImage("photo2.png")

image3 = HighResolutionImage("photo3.gif")

print("\nGallery initialized. User might view an image now.")

*# User clicks on image1*

print(f"User requests to display {image1.get\_file\_name()}")

image1.display()

*# User clicks on image3*

print(f"\nUser requests to display {image3.get\_file\_name()}")

image3.display()

print("\nApplication finished.")

**What's Wrong With This Approach?**

**1. . Resource-Intensive Initialization**

Every HighResolutionImage loads its image data at the time of construction, even if the user **never views** the image.

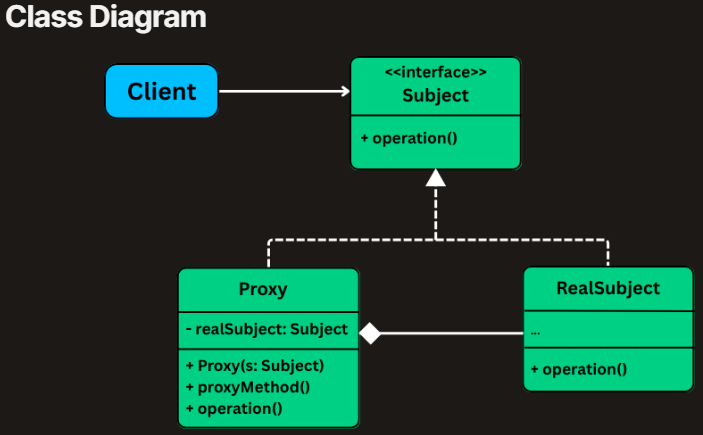
**2. What is the Proxy Pattern**

**Subject (e.g., Image) :** An **interface or abstract class** that defines the common operations shared by both the **RealSubject** and the **Proxy**.

**RealSubject (e.g., HighResolutionImage) :** The **actual object** that performs the real work.

**Proxy (e.g., ImageProxy):**

* **Implements the same interface** as the RealSubject (i.e., Image), allowing it to stand in seamlessly.
* **Holds a reference** to the real object and **controls when and how it is created or accessed**.
* Acts as a gatekeeper, delegating calls to the real object only when appropriate.



Depending on the use case, the Proxy may take different forms:

* **Virtual Proxy:**Defers creation of the real object until it’s actually needed (lazy loading).
* **Protection Proxy:**Performs permission checks before allowing access to certain operations.
* **Remote Proxy:**Handles communication between local and remote objects over a network.
* **Caching Proxy:**Caches expensive results and avoids repeated calls to the real subject.
* **Smart Proxy:**Adds logging, reference counting, or monitoring before/after method calls.

**3. Implementing Proxy**

**1. Create the Proxy Class**

class ImageProxy(Image):

def \_\_init\_\_(self, file\_name: str):

self.file\_name = file\_name

self.\_real\_image = None

print(f"ImageProxy: Created for {file\_name}.Real image not loaded yet")

def get\_file\_name(self) -> str:

*# Can safely return without loading the image*

return self.file\_name

def display(self) -> None:

*# Lazy initialization: Load only when display() is called*

if self.\_real\_image is None:

print(f"ImageProxy: display() requested for {self.file\_name}. Loading high-resolution image...")

self.\_real\_image = HighResolutionImage(self.file\_name)

else:

print(f"ImageProxy: Using cached high-resolution image for {self.file\_name}")

self.\_real\_image.display()

**2. Update the Client Code to Use the Proxy**

class ImageGalleryAppV2:

@staticmethod

def main() -> None:

print("Application Started. Creating proxies for gallery images...")

*# Only proxies are created initially - no heavy loading*

image1 = ImageProxy("photo1.jpg")

image2 = ImageProxy("photo2.png")

image3 = ImageProxy("photo3.gif")

print("\nGallery proxies initialized. No images loaded yet.")

*# User clicks on image1*

print(f"\nUser requests to display {image1.get\_file\_name()}")

image1.display()

*# User clicks on image3*

print(f"\nUser requests to display {image3.get\_file\_name()}")

image3.display()

*# Display image1 again (should use cached version)*

print(f"\nUser requests to display {image1.get\_file\_name()} again")

image1.display()

print("\nApplication finished. Note: photo2.png was never loaded.")

**4. Extending with Other Proxy Types**

**1. Adding a Protection Proxy**

A **Protection Proxy** controls access to sensitive operations based on **authorization rules**. For example, only users with an ADMIN role should be able to view confidential images.

class ProtectedImageProxy(ImageProxy):

def \_\_init\_\_(self, file\_name: str):

super().\_\_init\_\_(file\_name)

def \_check\_access(self, user\_role: str) -> bool:

print(f"ProtectionProxy: Checking access for role: {user\_role} on file: {self.file\_name}")

*# Basic access control rule:*

*# - ADMIN can see everything*

*# - Others can only see non-secret files*

return user\_role == "ADMIN" or "secret" not in self.file\_name

def display(self, user\_role: str = "GUEST") -> None:

if not self.\_check\_access(user\_role):

print(f"ProtectionProxy: Access denied for {self.file\_name}")

return

if self.\_real\_image is None:

print("ImageProxy: Loading image for authorized access...")

self.\_real\_image = HighResolutionImage(self.file\_name)

self.\_real\_image.display()

**Bridge**

The **Bridge Design Pattern** is a **structural pattern** that lets you **decouple an abstraction from its implementation**, allowing the two to vary **independently**.

The **Bridge Pattern** splits a class into two separate hierarchies:

* One for the **abstraction** (e.g., shape, UI control)
* One for the **implementation** (e.g., rendering engine, platform)

**1. The Problem: Drawing Shapes**

Imagine you're building a **cross-platform graphics library**. It supports rendering **shapes** like circles and rectangles using different rendering approaches:

* 🟢 **Vector rendering** – for scalable, resolution-independent output
* 🔵 **Raster rendering** – for pixel-based output

Now, you need to support:

* Drawing **different shapes** (e.g., Circle, Rectangle)
* Using **different renderers** (e.g., VectorRenderer, RasterRenderer)

class Shape(ABC):

def draw(self):

**Circle Variants**

class VectorCircle(Shape):

def draw(self):

print("Drawing Circle as VECTORS")

class RasterCircle(Shape):

def draw(self):

print("Drawing Circle as PIXELS")

**Rectangle Variants**

class VectorRectangle(Shape):

def draw(self):

print("Drawing Rectangle as VECTORS")

class RasterRectangle(Shape):

def draw(self):

print("Drawing Rectangle as PIXELS")

**Client Code**

s1 = VectorCircle()

s2 = RasterRectangle()

s1.draw() *# Output: Drawing Circle as VECTORS*

s2.draw() *# Output: Drawing Rectangle as PIXELS*

**Why This Quickly Breaks Down**

**1. Class Explosion**

Every new combination of shape and rendering method requires a new subclass:

**2. What is the Bridge Pattern**

The **Bridge Design Pattern** lets you **split a class into two separate hierarchies** — one for the **abstraction** and another for the **implementation** — so that they can evolve independently.

**1. Abstraction (e.g., Shape)**

The high-level interface that defines the abstraction's core behavior. It maintains a reference to an Implementor and delegates work to it.

**2. RefinedAbstraction (e.g., Circle, Rectangle)**

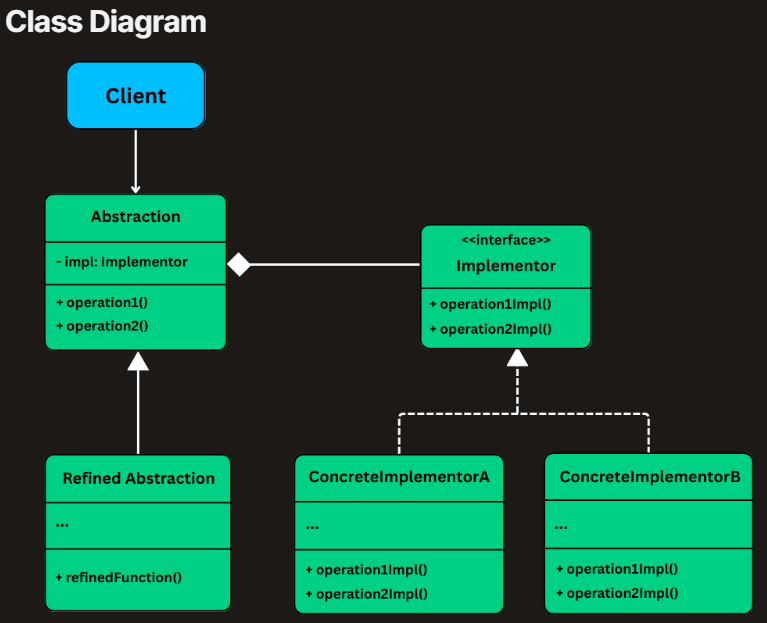
A concrete subclass of Abstraction that adds additional behaviors or logic. It still relies on the implementor for actual execution.

**3. Implementor (e.g., Renderer)**

An interface that declares the operations to be implemented by concrete implementors. These are the low-level operations.

**4. ConcreteImplementors (e.g., VectorRenderer,RasterRenderer)**

Platform- or strategy-specific classes that implement the Implementor interface. They contain the actual logic for performing the delegated operations.



**3. Implementing Bridge**

**1: Define the Implementor Interface (Renderer)**

class Renderer(ABC):

def render\_circle(self, radius: float) -> None:

def render\_rectangle(self, width: float, height: float) -> None:

**2. Create Concrete Implementations of the Renderer**

class VectorRenderer(Renderer):

def render\_circle(self, radius: float) -> None:

print(f"Drawing a circle of radius {radius} using VECTOR rendering.")

def render\_rectangle(self, width: float, height: float) -> None:

print(f"Drawing a rectangle {width}x{height} using VECTOR rendering.")

class RasterRenderer(Renderer):

def render\_circle(self, radius: float) -> None:

print(f"Drawing pixels for a circle of radius {radius} (RASTER).")

def render\_rectangle(self, width: float, height: float) -> None:

print(f"Drawing pixels for a rectangle {width}x{height} (RASTER).")

**3. Define the Abstraction (Shape)**

class Shape(ABC):

def \_\_init\_\_(self, renderer: Renderer):

self.renderer = renderer

def draw(self) -> None:

**4. Create Concrete Shapes**

class Circle(Shape):

def \_\_init\_\_(self, renderer: Renderer, radius: float):

super().\_\_init\_\_(renderer)

self.radius = radius

def draw(self) -> None:

self.renderer.render\_circle(self.radius)

class Rectangle(Shape):

def \_\_init\_\_(self, renderer: Renderer, width: float, height: float):

super().\_\_init\_\_(renderer)

self.width = width

self.height = height

def draw(self) -> None:

self.renderer.render\_rectangle(self.width, self.height)

**5. Client Code**

vector = VectorRenderer()

raster = RasterRenderer()

circle1 = Circle(vector, 5)

circle2 = Circle(raster, 5)

rectangle1 = Rectangle(vector, 10, 4)

rectangle2 = Rectangle(raster, 10, 4)

circle1.draw() *# Vector*

circle2.draw() *# Raster*

rectangle1.draw() *# Vector*

rectangle2.draw() *# Raster*

**Flyweight**

Focuses on **efficiently sharing common parts of object state** across many objects to **reduce memory usage**.

**1. The Problem: Rendering Characters**

Imagine you're building a **rich text editor** that needs to render characters on screen.

Every character (a, b, c, ..., z, punctuation, etc.) must be displayed with formatting information.

class CharacterGlyph:

def \_\_init\_\_(self, symbol: str, font\_family: str, font\_size: int, color: str, x: int, y: int):

self.symbol = symbol *# e.g., 'a', 'b', etc.*

self.font\_family = font\_family *# e.g., "Arial"*

self.font\_size = font\_size *# e.g., 12*

self.color = color *# e.g., "#000000"*

self.x = x *# position X*

self.y = y *# position Y*

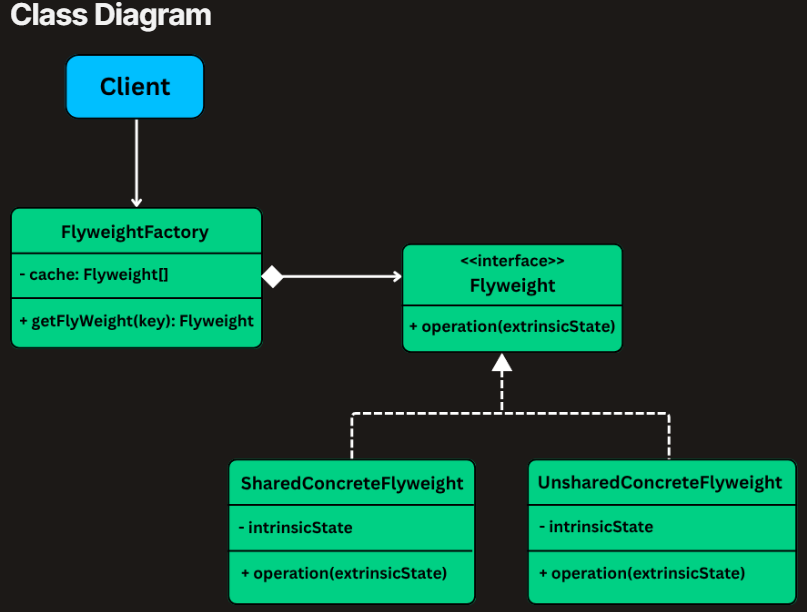
def draw(self) -> None:

print(f"Drawing '{self.symbol}' in {self.font\_family}, "

f"size {self.font\_size}, color {self.color} at ({self.x},{self.y})")

**2. What is the Flyweight Pattern**

The **Flyweight Pattern** minimizes memory usage by **sharing as much data as possible** between similar objects.



**Flyweight Interface:** Declares a method like draw(x, y) that takes extrinsic state (position)

**ConcreteFlyweight:** Implements the flyweight and stores **intrinsic state** like font and symbol

**FlyweightFactory:**Caches and reuses flyweights to avoid duplication

**3. Implementing Flyweight**

**1. Define the Flyweight Interface**

class CharacterFlyweight(ABC):

def draw(self, x: int, y: int) -> None:

**2. Implement the Concrete Flyweight**

class CharacterGlyph(CharacterFlyweight):

def \_\_init\_\_(self, symbol:str, font\_family:str, font\_size:int, color: str):

self.symbol = symbol

self.font\_family = font\_family

self.font\_size = font\_size

self.color = color

def draw(self, x: int, y: int) -> None:

print(f"Drawing '{self.symbol}' [Font: {self.font\_family}, "

f"Size: {self.font\_size}, Color: {self.color}] at ({x},{y})")

**3. Create the Flyweight Factory**

class CharacterFlyweightFactory:

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

self.\_flyweights = {}

def get\_flyweight(self, symbol: str, font\_family: str, font\_size: int, color: str) -> CharacterFlyweight:

key = f"{symbol}{font\_family}{font\_size}{color}"

if key not in self.\_flyweights:

self.\_flyweights[key] = CharacterGlyph(symbol, font\_family, font\_size, color)

return self.\_flyweights[key]

@property

def flyweight\_count(self) -> int:

return len(self.\_flyweights)

**4. Create the Client**

class TextEditorClient:

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

self.factory = CharacterFlyweightFactory()

self.document = []

def add\_character(self, char: str, x: int, y: int, font: str, size: int, color: str) -> None:

glyph = self.factory.get\_flyweight(char, font, size, color)

self.document.append(RenderedCharacter(glyph, x, y))

def render\_document(self) -> None:

for rc in self.document:

rc.render()

print(f"Total flyweight objects used: {self.factory.flyweight\_count}")

class RenderedCharacter:

def \_\_init\_\_(self, glyph: CharacterFlyweight, x: int, y: int):

self.glyph = glyph

self.x = x

self.y = y

def render(self) -> None:

self.glyph.draw(self.x, self.y)