**Template**

The **Template Method Design Pattern** is a **behavioral design pattern** that defines the **skeleton of an algorithm** in a base class, but allows **subclasses to override specific steps** of the algorithm without changing its overall structure.

**1. The Problem: Exporting Reports**

Let’s say you’re building a tool that allows your application to **export reports in different formats** — such as **CSV**, **PDF**, and **Excel**.

On the surface, each report exporter has a different output format, but underneath, the overall process is almost identical.



1. **Prepare Data**: Gather and organize the data to be exported.
2. **Open File**: Create or open the output file in the desired format.
3. **Write Header**: Output column headers or metadata (**format-specific**).
4. **Write Data Rows**: Iterate through the dataset and write the rows (**format-specific**).
5. **Write Footer**: Add optional summary or footer info.
6. **Close File**: Finalize and close the output file.

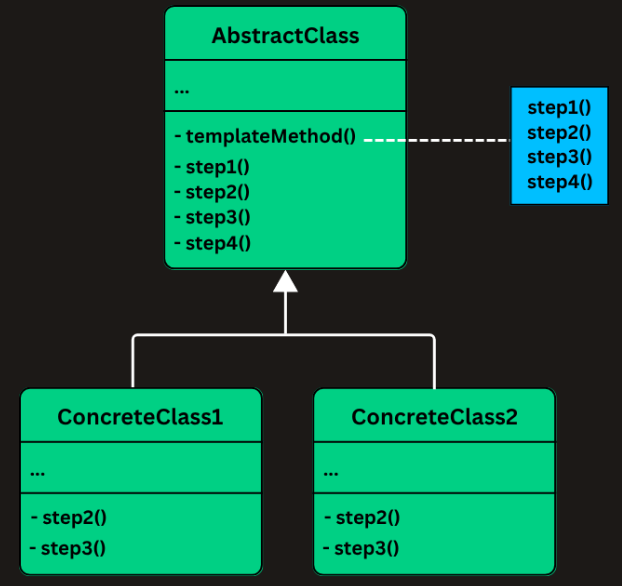
Despite these similarities, if you implement each exporter naively, you'll likely **duplicate a lot of logic**, and that comes at a cost.

**2. What is the Template Method Pattern**

**1. AbstractClass (e.g., AbstractReportExporter):**

* Contains the template method: a final method (e.g., exportReport()) that defines the fixed sequence of steps for the algorithm
* include **hook methods**: concrete methods with default behavior that subclasses can optionally override to customize certain points in the algorithm.

**2. Concrete Classes (e.g., CsvReportExporter, PdfReportExporter) :** Extend the abstract base class.



**3. Implementing Template Method**

**1. Create the Abstract Base Class**

class ReportData:

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

self.headers: List[str] = []

self.rows: List[Dict[str, Any]] = []

class AbstractReportExporter(ABC):

def export\_report(self, data: ReportData, file\_path: str) -> None:

self.\_prepare\_data(data)

self.\_open\_file(file\_path)

self.\_write\_header(data)

self.\_write\_data\_rows(data)

self.\_write\_footer(data)

self.\_close\_file(file\_path)

print(f"Export complete: {file\_path}")

def \_prepare\_data(self, data: ReportData) -> None: *# Hook method*

print("Preparing report data (common step)...")

def \_open\_file(self, file\_path: str) -> None: *# Hook method*

print(f"Opening file '{file\_path}'")

def \_write\_header(self, data: ReportData) -> None:

def \_write\_data\_rows(self, data: ReportData) -> None:

def \_write\_footer(self, data: ReportData) -> None: *# Hook method*

print("Writing footer (default: no footer).")

def \_close\_file(self, file\_path: str) -> None: *# Hook method*

print(f"Closing file '{file\_path}'")

**2. Implement Concrete Exporters**

class CsvReportExporter(AbstractReportExporter):

def \_write\_header(self, data: ReportData) -> None:

print(f"CSV: Writing header: {','.join(data.headers)}")

def \_write\_data\_rows(self, data: ReportData) -> None:

print("CSV: Writing data rows...")

for row in data.rows:

print(f"CSV: {list(row.values())}")

As follows …

**Client Code**

class ReportAppTemplateMethod:

def main() -> None:

data = ReportData()

data.headers = ["Date", "Product", "Revenue"]

data.rows = [

{"Date": "2023-01-01", "Product": "Widget", "Revenue": 1000},

{"Date": "2023-01-02", "Product": "Gadget", "Revenue": 1500}

]

csv\_exporter = CsvReportExporter()

csv\_exporter.export\_report(data, "sales\_report.csv")

print("\n" + "="\*40 + "\n")

pdf\_exporter = PdfReportExporter()

pdf\_exporter.export\_report(data, "financial\_summary.pdf")

**Visitor**

The **Visitor Design Pattern** is a **behavioral pattern** that lets you **add new operations to existing object structures** without modifying their classes.

**1. The Problem: Adding Operations to a Shape Hierarchy**

Imagine you’re building a **vector graphics editor** that supports multiple shape types:

* Circle , Rectangle and Triangle

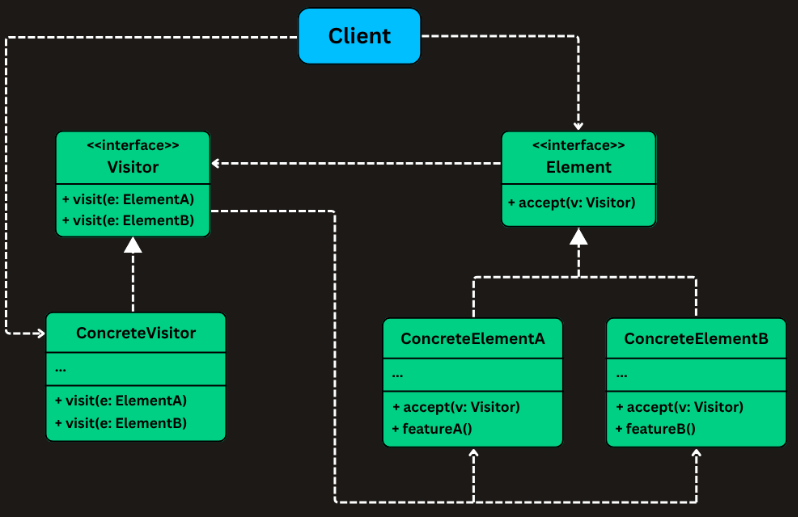
Each shape is part of a common hierarchy and must support a variety of operations, such as:

* **Rendering** on screen , **Calculating area** , **Exporting to SVG** and **Serializing to JSON**.

The simplest approach is to add all of these methods to each shape class:

**2. What is the Visitor Pattern**

The **Visitor Design Pattern** lets you **separate algorithms from the objects on which they operate**.



**1. Element Interface (e.g., Shape)**

* Represents the **objects in your object structure** (such as graphical shapes, document nodes, AST elements).

**2. Concrete Elements (e.g., Circle, Rectangle)**

* Implements the Element interface.

**3. Visitor Interface**

* Declares a set of visit() methods — **one for each concrete element type**.

**4. Concrete Visitors (e.g., AreaCalculatorVisitor)**

* Implements the Visitor interface.

**3. Implementing Visitor Pattern**

**1. Define the Shape Interface (Element)**

class Shape(ABC):

def accept(self, visitor: 'ShapeVisitor') -> None:

**2. Create Concrete Shape Classes (Elements)**

class Circle(Shape):

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

self.radius = radius

def accept(self, visitor: 'ShapeVisitor') -> None:

visitor.visit\_circle(self)

class Rectangle(Shape):

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

self.width = width

self.height = height

def accept(self, visitor: 'ShapeVisitor') -> None:

visitor.visit\_rectangle(self)

**3. Define the Visitor Interface**

class ShapeVisitor(ABC):

def visit\_circle(self, circle: Circle) -> None:

def visit\_rectangle(self, rectangle: Rectangle) -> None:

**4. Implement Concrete Visitors**

class AreaCalculatorVisitor(ShapeVisitor):

def visit\_circle(self, circle: Circle) -> None:

area = math.pi \* circle.radius \*\* 2

print(f"Area of Circle: {area:.2f}")

def visit\_rectangle(self, rectangle: Rectangle) -> None:

area = rectangle.width \* rectangle.height

print(f"Area of Rectangle: {area:.2f}")

class SvgExporterVisitor(ShapeVisitor):

def visit\_circle(self, circle: Circle) -> None:

print(f'<circle r="{circle.radius}" />')

def visit\_rectangle(self, rectangle: Rectangle) -> None:

print(f'<rect width="{rectangle.width}" height="{rectangle.height}" />')

**5. Client Code**

class VisitorPatternDemo:

def main() -> None:

shapes: List[Shape] = [ Circle(5), Rectangle(10, 4), Circle(2.5) ]

print("=== Calculating Areas ===")

area\_calculator = AreaCalculatorVisitor()

for shape in shapes:

shape.accept(area\_calculator)

print("\n=== Exporting to SVG ===")

svg\_exporter = SvgExporterVisitor()

for shape in shapes:

shape.accept(svg\_exporter)

**Mediator**

The **Mediator Design Pattern** is a **behavioral pattern** that defines an object (the **Mediator**) to **encapsulate how a set of objects interact**.

For example, in a GUI form, when a user types into a field, it may enable or disable a button, update a label, and trigger validation — each component talking to the others directly.

But as more components are added, this direct communication becomes hard to manage.

The **Mediator Pattern** solves this by introducing a **central object that handles communication between components**.

**1. The Problem: Tightly Coupled UI Components**

Imagine you're building a **login form** with the following UI components:

* A **username field** , A **password field** , A **login button** and A **status label**

The logic of the form is simple:

* The **login button** should be enabled only when both username and password fields are non-empty.
* When the button is clicked, it should attempt login and display the result in the **status label**.

class TextField:

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

self.\_text = ""

self.\_login\_button = None

def text(self) -> str:

return self.\_text

def text(self, new\_text: str) -> None:

self.\_text = new\_text

print(f"TextField updated: {self.\_text}")

if self.\_login\_button:

self.\_login\_button.check\_enabled()

def set\_login\_button(self, button: 'Button') -> None:

self.\_login\_button = button

class Button:

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

self.\_username\_field = None

self.\_password\_field = None

self.\_status\_label = None

def set\_dependencies(self, username: TextField, password: TextField, status: 'Label') -> None:

self.\_username\_field = username

self.\_password\_field = password

self.\_status\_label = status

def check\_enabled(self) -> None:

enable = (self.\_username\_field and self.\_username\_field.text

and self.\_password\_field and self.\_password\_field.text)

print(f"Login Button is now {'ENABLED' if enable else 'DISABLED'}")

def click(self) -> None:

if (self.\_username\_field and self.\_username\_field.text

and self.\_password\_field and self.\_password\_field.text):

print("Login successful!")

if self.\_status\_label:

self.\_status\_label.text = "✅ Logged in!"

else:

print("Login failed.")

if self.\_status\_label:

self.\_status\_label.text = "❌ Please enter username and password."

class Label:

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

self.\_text = ""

def text(self) -> str:

return self.\_text

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

self.\_text = message

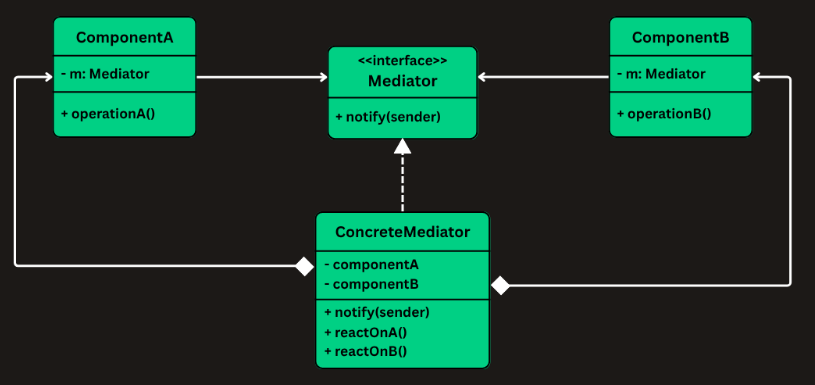
print(f"Status: {self.\_text}")

**What’s Wrong with This Design?**

**1. Tight Coupling**

Every component **knows about others**: the TextField knows the Button, the Button knows the TextField and the Label. A change in one component's logic often requires updates in others

**2. The Mediator Pattern**



**3. Implementing Mediator**

**1. Define the UIMediator Interface**

class UIMediator(ABC):

def component\_changed(self, component: 'UIComponent') -> None:

**2. Define the Abstract Component (UIComponent)**

class UIComponent:

def \_\_init\_\_(self, mediator: UIMediator):

self.\_mediator = mediator

def notify\_mediator(self) -> None:

self.\_mediator.component\_changed(self)

**3. Implement the Components**

class TextField(UIComponent):

def \_\_init\_\_(self, mediator: UIMediator):

super().\_\_init\_\_(mediator)

self.\_text = ""

@property

def text(self) -> str:

return self.\_text

@text.setter

def text(self, new\_text: str) -> None:

self.\_text = new\_text

print(f"TextField updated: {self.\_text}")

self.notify\_mediator()

class Button(UIComponent):

class Label(UIComponent):

**4. Implement the Concrete Mediator (FormMediator)**

class FormMediator(UIMediator):

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

self.\_username\_field: TextField = None

self.\_password\_field: TextField = None

self.\_login\_button: Button = None

self.\_status\_label: Label = None

def set\_username\_field(self, field: TextField) -> None:

self.\_username\_field = field

def set\_password\_field(self, field: TextField) -> None:

self.\_password\_field = field

def set\_login\_button(self, button: Button) -> None:

self.\_login\_button = button

def set\_status\_label(self, label: Label) -> None:

self.\_status\_label = label

def component\_changed(self, component: UIComponent) -> None:

if component in (self.\_username\_field, self.\_password\_field):

enable = (self.\_username\_field.text

and self.\_password\_field.text)

self.\_login\_button.enabled = enable

elif component == self.\_login\_button:

username = self.\_username\_field.text

password = self.\_password\_field.text

if username == "admin" and password == "1234":

self.\_status\_label.text = "✅ Login successful!"

else:

self.\_status\_label.text = "❌ Invalid credentials."

**5. Connect Everything in the Client**

class MediatorApp:

@staticmethod

def main() -> None:

mediator = FormMediator()

username\_field = TextField(mediator)

password\_field = TextField(mediator)

login\_button = Button(mediator)

status\_label = Label(mediator)

mediator.set\_username\_field(username\_field)

mediator.set\_password\_field(password\_field)

mediator.set\_login\_button(login\_button)

mediator.set\_status\_label(status\_label)

*# Simulate user interaction*

username\_field.text = "admin"

password\_field.text = "1234"

login\_button.click() *# Should succeed*

**Memento**

The **Memento Design Pattern** is a **behavioral design pattern** that lets you **capture and store an object’s internal state** so it can be **restored later**, without violating encapsulation.

**1. The Problem: Implementing Undo in a Text Editor**

Imagine you’re building a simple **text editor**. The editor supports basic operations like:

* type(String text) – appends text to the current document
* getContent() – returns the current document text
* undo() – reverts to the previous version of the content

class TextEditorNaive:

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

self.\_content = ""

def type(self, new\_text: str) -> None:

"""Add new text to the content"""

self.\_content += new\_text

print(f"Typed: '{new\_text}' | Content: '{self.\_content}'")

def undo(self, previous\_content: str) -> None:

"""Revert to previous content"""

self.\_content = previous\_content

print(f"Undid to: '{self.\_content}'")

@property

def content(self) -> str:

"""Get current content"""

return self.\_content

def demonstrate\_naive\_approach():

editor = TextEditorNaive()

saved\_state = editor.content

editor.type("Hello")

editor.type(" World")

editor.undo(saved\_state)

print(f"Final content: '{editor.content}'")

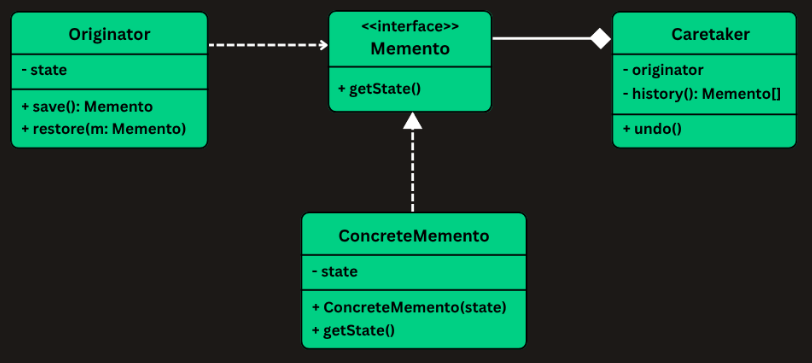
**What’s Wrong with This Design?**

**1. Encapsulation is Broken**

* The client must **manually fetch and store internal state** (getContent()) just to implement undo.

**2. What is the Memento Pattern**

The **Memento Design Pattern** allows an object to **save and restore its state** without exposing its internal structure.



**1. Originator (e.g., TextEditor) :** The object whose internal state you want to capture and restore later**.**

**2. Memento :** A passive object that holds the snapshot of the Originator’s state at a given point in time.

**3. ConcreteMemento (Optional)**

**4. Caretaker :** The **Caretaker** is responsible for **storing, managing, and restoring** mementos.

**3. Implementing Memento**

**1. Create the Memento Class - TextEditorMemento**

class TextEditorMemento:

"""Immutable snapshot of editor state"""

\_\_slots\_\_ = ['\_state'] *# Memory optimization*

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

self.\_state = state

@property

def state(self) -> str:

return self.\_state

**2. Create the Originator – TextEditor**

class TextEditor:

"""The object whose state we want to save/restore"""

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

self.\_content = ""

def type(self, new\_text: str) -> None:

"""Add text to the editor"""

self.\_content += new\_text

print(f"Typed: '{new\_text}'")

@property

def content(self) -> str:

return self.\_content

def save(self) -> TextEditorMemento:

"""Create memento of current state"""

print(f"Saving state: '{self.\_content}'")

return TextEditorMemento(self.\_content)

def restore(self, memento: TextEditorMemento) -> None:

"""Restore state from memento"""

self.\_content = memento.state

print(f"Restored state to: '{self.\_content}'")

**3. Create the Caretaker – TextEditorUndoManager**

class TextEditorUndoManager:

"""Manages the undo history"""

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

self.\_history: List[TextEditorMemento] = []

def save(self, editor: TextEditor) -> None:

"""Save current editor state"""

self.\_history.append(editor.save())

def undo(self, editor: TextEditor) -> None:

"""Undo last operation"""

if not self.\_history:

print("Nothing to undo.")

return

editor.restore(self.\_history.pop())

**4. Client Code**

**Chain of Responsibility**

The **Chain of Responsibility Design Pattern** is a **behavioral pattern** that lets you **pass requests along a chain of handlers**, allowing each handler to decide whether to process the request or pass it to the next handler in the chain.

**1. The Problem: Handling HTTP Requests**

class Request:

def \_\_init\_\_(self, user: str, user\_role: str, request\_count: int, payload: str):

self.user = user

self.user\_role = user\_role

self.request\_count = request\_count

self.payload = payload

class RequestHandler:

def handle(self, request: Request) -> None:

if not self.\_authenticate(request):

print("Request Rejected: Authentication failed.")

return

if not self.\_authorize(request):

print("Request Rejected: Authorization failed.")

return

if not self.\_rate\_limit(request):

print("Request Rejected: Rate limit exceeded.")

return

if not self.\_validate(request):

print("Request Rejected: Invalid payload.")

return

print("Request passed all checks. Executing business logic...")

*# Proceed to business logic*

def \_authenticate(self, req: Request) -> bool:

return req.user is not None

def \_authorize(self, req: Request) -> bool:

return req.user\_role == "ADMIN"

def \_rate\_limit(self, req: Request) -> bool:

return req.request\_count < 100

def \_validate(self, req: Request) -> bool:

return req.payload is not None and req.payload != ""

**Common Pre-processing Steps**

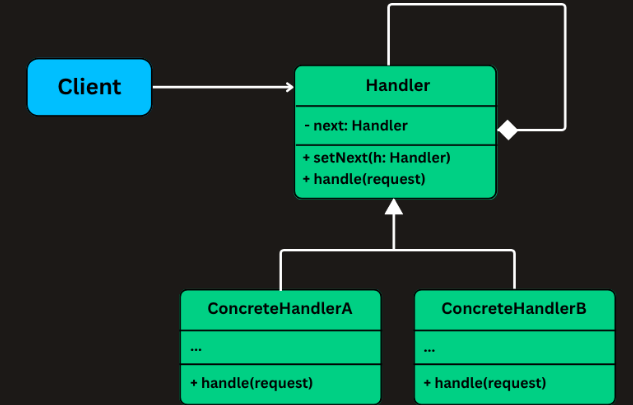
1. **Authentication** – Is the user properly authenticated (e.g., via token or session)?
2. **Authorization** – Is the authenticated user allowed to perform this action?
3. **Rate Limiting** – Has the user exceeded their allowed number of requests?
4. **Data Validation** – Is the request payload well-formed and valid?

**Why This Approach Breaks Down**

**1. Hard to Extend or Modify**

If you want to change the order of checks or add a new one (e.g., logging, metrics, caching), you must **modify the existing handler** — violating the **Open/Closed Principle**.

**2. What is the Chain of Responsibility Pattern**



**3. Implementing Chain of Responsibility**

**1. Define the Common Handler Interface**

class RequestHandler(ABC):

def set\_next(self, next\_handler: 'RequestHandler') -> None:

def handle(self, request: 'Request') -> None:

**2. Create the Abstract Base Handler**

class BaseHandler(RequestHandler):

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

self.\_next\_handler: Optional[RequestHandler] = None

def set\_next(self, next\_handler: RequestHandler) -> None:

self.\_next\_handler = next\_handler

def handle(self, request: 'Request') -> None:

if self.\_next\_handler:

self.\_next\_handler.handle(request)

**3. Create Concrete Handlers**

class AuthHandler(BaseHandler):

def handle(self, request: 'Request') -> None:

if request.user is None:

print("AuthHandler: ❌ User not authenticated.")

return

print("AuthHandler: ✅ Authenticated.")

super().handle(request)

class AuthorizationHandler(BaseHandler):

class RateLimitHandler(BaseHandler):

class ValidationHandler(BaseHandler):

class BusinessLogicHandler(BaseHandler):

**4. Assemble the Chain in Client Code**