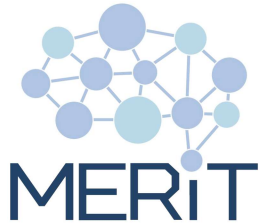
Design 3

Design Patterns

Design Patterns

Definition



- Design patterns are **proven**, **reusable solutions** to **common problems** that **occur repeatedly** in software design.
- They are **abstract templates**, **not code**, but **guidelines** that can be implemented in different ways depending on the context.

Purpose

- Solve problems **faster** with **known solutions**
- Avoid **reinventing the wheel**
- Improve communication through **shared terminology**
- **Design principles** in action

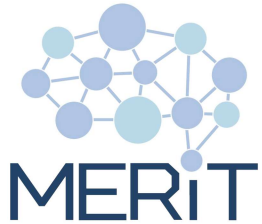
Categories

- Creational
- Structural
- Behavioral

1. Creational

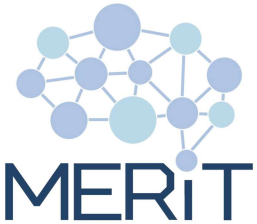
Singleton

Definition



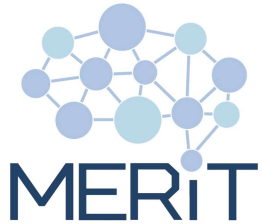
- The Singleton Pattern ensures that **only one instance** of a class exists within the entire system and provides a **global access point** to that instance.

Core Idea



- You want exactly one object managing some global state, resource, or configuration.
- Instead of allowing multiple instances, you control instance creation and store it internally.

When to Use



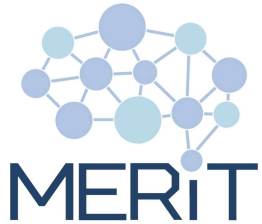
- You need a single configuration manager
- You want a global cache, logger, or metrics collector
- You manage shared hardware resources (e.g., printer manager, thread pool)

Code

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance
```

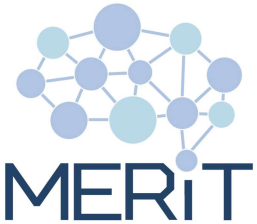
Factory

Definition



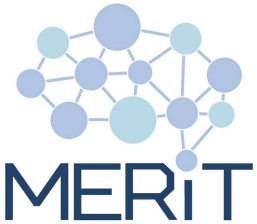
- The Factory Pattern defines an interface for creating objects, but **allows subclasses to decide which class to instantiate**.
- Delegate the creation of objects to a factory method, instead of calling constructors (new) directly.

Core Idea



- You want to create objects without exposing the instantiation logic to the client.
- Let subclasses or strategies decide which concrete class to create based on runtime context.

When to Use

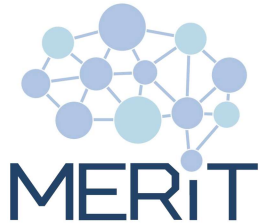


- You need to create objects from a family of related types
- Instantiation depends on logic, config, or environment
- Code should depend on abstraction, not concrete classes

Code – Without Factory

```
if os == "Windows":
    button = WindowsButton()
else:
    button = MacButton()
```

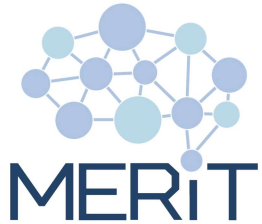
Code – With Factory




```
class Dialog:
    def create_button(self):  # Factory Method
        pass

class WindowsDialog(Dialog):
    def create_button(self):
        return WindowsButton()

class MacDialog(Dialog):
    def create_button(self):
        return MacButton()
```

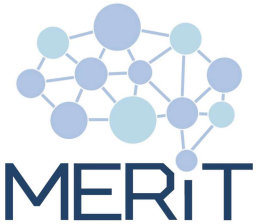
Abstract Factory

Definition



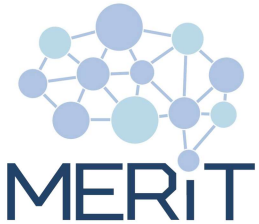
- The Abstract Factory Pattern provides an interface for **creating families of related or dependent objects without specifying their concrete classes**.
- In contrast to Factory Method (which returns one product), Abstract Factory returns a full set of related products, ensuring compatibility and consistency.

Core Idea



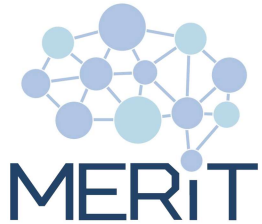
- You want to create multiple related objects, but don’t want to hardcode their classes.
- Each "product family" is created via its own factory — you just choose the factory.

When to Use



- You need to create related groups of objects that must match or work together
- You want to enforce consistency across families of components
- You need to support multiple variants of a product (e.g., themes, platforms, vendors)
- You want to decouple client code from concrete class implementations

Code



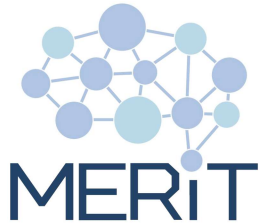

```
class GUIFactory:
    def create_button(self): pass
    def create_scrollbar(self): pass

class LightThemeFactory(GUIFactory):
    def create_button(self): return LightButton()
    def create_scrollbar(self): return LightScrollbar()

class DarkThemeFactory(GUIFactory):
    def create_button(self): return DarkButton()
    def create_scrollbar(self): return DarkScrollbar()
```

Builder

Definition

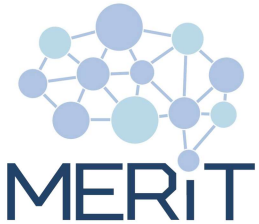
- The Builder Pattern separates the construction of a complex object from its representation, allowing the same construction process to create **different representations**.
- Use step-by-step construction logic, and then assemble the final object, especially useful when the object has many optional parts or configurations.

Core Idea



- Building a complex object is not a one-step job.
- Instead of having a huge constructor with many parameters, break the construction into small, manageable steps.
- The client does not need to know how the object is built internally, only how to trigger the steps.

When to Use

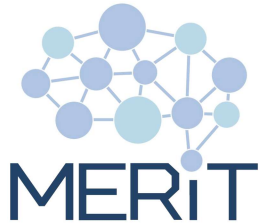


- You need to create complex objects with many configuration options or steps
- Object creation requires multiple stages or validation
- Objects need to support multiple representations or formats (e.g., HTML vs. JSON reports)
- You want to avoid telescoping constructors (new Product(a, b, c, d, e))
- Immutability or stepwise control is required

Code – Without Builder

```
soup = Soup(
    base="chicken",
    has_noodles=True,
    has_carrots=True,
    has_celery=False,
    spicy=False
)
```

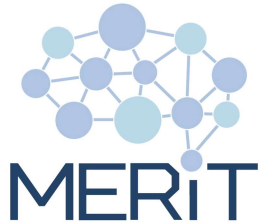
Code – With Builder




```
builder = SoupBuilder()
pizza = (
    builder.set_base("chicken")
        .add_noodles()
        .add_carrots()
        .make_spicy()
        .build()
)
```

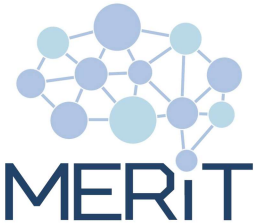
Prototype

Definition



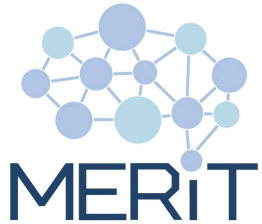
- The Prototype Pattern creates new objects by **cloning an existing object** (the prototype), instead of instantiating classes directly.
- It focuses on copying complex, preconfigured objects, especially when creating them from scratch is expensive or complicated.

Core Idea



- Instead of constructing new instances from scratch, use an existing object as a template.
- The system doesn’t need to know the exact class of the object, it just clones from a prototype.
- Especially useful when object creation is costly, slow, or involves complex setup.

When to Use



- The cost of creating a new object is high (time, memory, setup)
- You need to create many identical or similar objects
- Object configuration is complex or dynamic
- You want to decouple instantiation from the class hierarchy
- You need to add or remove prototypes at runtime

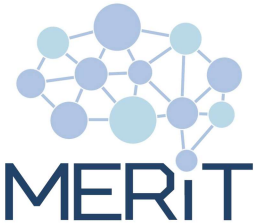
Code

```
archer_prototype = Enemy(
    health=100,
    weapon="bow",
    speed=5
)

new_archer = archer_prototype.clone()
```

2. Structural

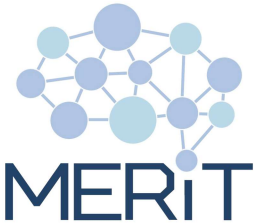
Adapter

Definition



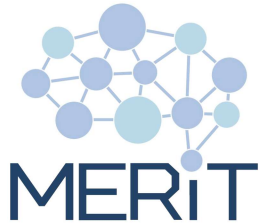
- The Adapter Pattern allows incompatible interfaces to **work together**.
- It converts the interface of a class into one that the client expects.

Core Idea



- Sometimes, you want to use a class that does the job, but its interface doesn't match what your system expects.
- Instead of changing that class, you **wrap it in an adapter** that translates between interfaces.

When to Use

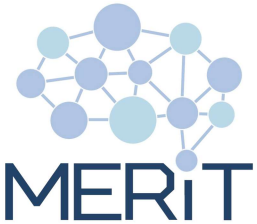


- Integrating a third-party, legacy, or incompatible component
- Two classes can't communicate due to an interface mismatch
- Migrating or refactoring without breaking existing clients
- You need to bridge new code with old systems

Code – Without Adapter

```
class LegacyPrinter:
    def print_text(self, msg):
        print(msg)

# Developer expects a .print() method
printer = LegacyPrinter()
printer.print()  # Error!
```

Code – With Adapter



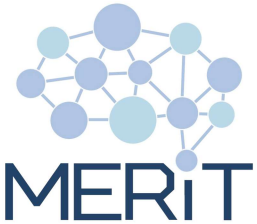

```
class PrinterAdapter:
    def __init__(self, legacy_printer):
        self.legacy = legacy_printer

    def print(self):
        self.legacy.print_text("Hello")

printer = PrinterAdapter(LegacyPrinter())
printer.print() # Now it works
```

Bridge

Definition

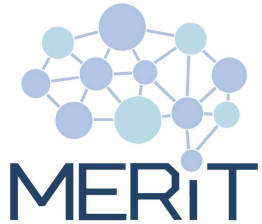
- The Bridge Pattern **decouples an abstraction from its implementation**, so the two can evolve independently.

Core Idea



- Sometimes, you want to **separate high-level logic** (abstraction) from **low-level platform-specific implementation**.
- Instead of binding them tightly through inheritance, use composition, the abstraction “bridges” to an implementation via an interface.

When to Use

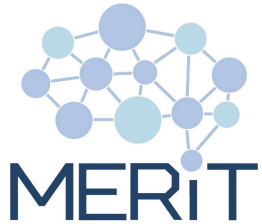


- Abstraction and implementation may vary independently
- You want to avoid a class explosion from combinatorial subclasses
- You want to switch implementations at runtime
- Porting logic across multiple platforms or APIs

Code

```
# Implementor
class DrawingAPI:
    def draw_circle(self, x, y, radius): pass

# ConcreteImplementor
class DrawingAPI1(DrawingAPI):
    def draw_circle(self, x, y, radius):
        print(f"API1.circle at ({x}, {y}) with radius {radius}")

class DrawingAPI2(DrawingAPI):
    def draw_circle(self, x, y, radius):
        print(f"API2.circle at ({x}, {y}) with radius {radius}")
```

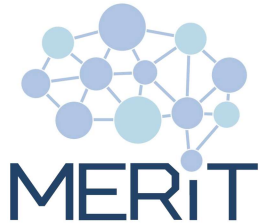
Code




```
class Shape:
    def __init__(self, drawing_api):
        self.drawing_api = drawing_api

    def draw(self): pass

class Circle(Shape):
    def __init__(self, x, y, radius, drawing_api):
        super().__init__(drawing_api)
        self.x, self.y, self.radius = x, y, radius

    def draw(self):
        self.drawing_api.draw_circle(self.x, self.y, self.radius)
```

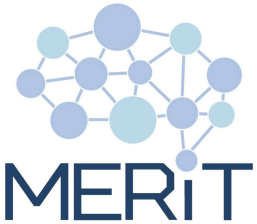
Decorator

Definition



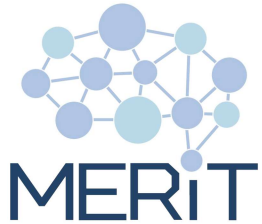
- The Decorator Pattern **allows behavior to be added to an individual object** dynamically, without affecting the behavior of other objects from the same class.
- It wraps the object in a chain of decorators, each adding functionality, while keeping the same interface.

Core Idea



- You want to extend an object's behavior at runtime, without changing its class.
- Instead of inheritance, you use composition: each decorator wraps the object and adds something.
- Think layered behavior: each decorator adds something before/after delegating.

When to Use

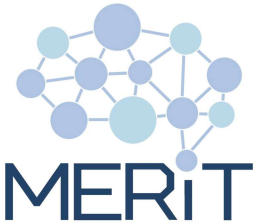


- You want to dynamically add responsibilities to an object
- Subclassing would lead to a combinatorial explosion
- You need to add cross-cutting concerns (e.g., logging, caching, validation)
- You want to follow the Open/Closed Principle (extend without modifying)

Code

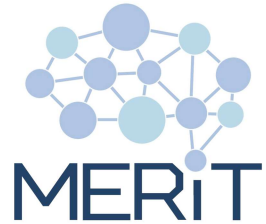
```
# Component
class Text:
    def render(self): pass

# Concrete Component
class PlainText(Text):
    def __init__(self, content):
        self.content = content
    def render(self):
        return self.content
```

Code




```
# Decorator
class TextDecorator(Text):
    def __init__(self, inner):
        self.inner = inner

# Concrete Decorators
class BoldDecorator(TextDecorator):
    def render(self):
        return f"<b>{self.inner.render()}</b>"

class ItalicDecorator(TextDecorator):
    def render(self):
        return f"<i>{self.inner.render()}</i>"
```

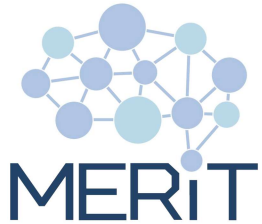
Code




```
text = PlainText("Hello")
decorated = ItalicDecorator(BoldDecorator(text))
print(decorated.render())  # <i><b>Hello</b></i>
```

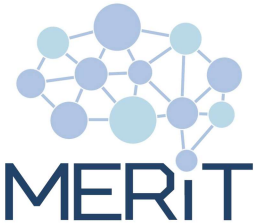
Proxy

Definition



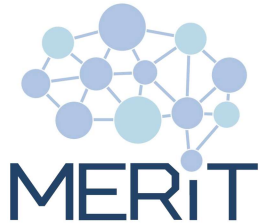
- The Proxy Pattern provides a **surrogate or placeholder** for another object to control access to it.
- It acts as an intermediary between the client and the real object, adding extra behavior like access control, lazy initialization, logging, and security.

Core Idea



- You want to protect, optimize, or mediate access to another object.
- Instead of the client accessing the object directly, the proxy wraps it and decides how and when to delegate.

When to Use



- You need to add access control or logging to an object
- You want to implement lazy loading or caching
- The actual object is expensive or remote (e.g., networked, large file)
- You need to hide implementation details or security rules

Code

```
class RealImage:
    def __init__(self, filename):
        print(f"Loading {filename} from disk...")
        self.filename = filename

    def display(self):
        print(f"Displaying {self.filename}")
```

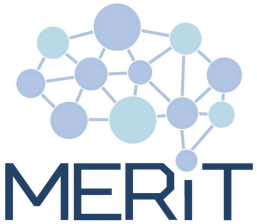
Code

```
class ImageProxy:
    def __init__(self, filename):
        self.filename = filename
        self.real_image = None

    def display(self):
        if not self.real_image:
            self.real_image = RealImage(self.filename)
        self.real_image.display()
```

3. Behavioral

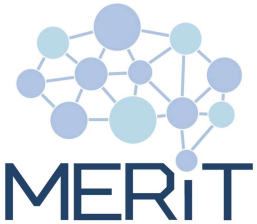
Observer

Definition



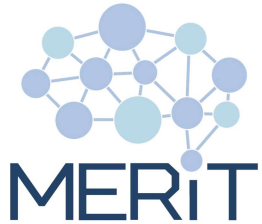
- The Observer Pattern defines a **one-to-many dependency** between objects, so that when one object (the subject) changes state, **all its dependents** (observers) are automatically notified and updated.

Core Idea



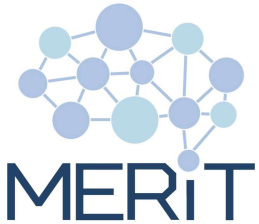
- You have a source of truth (subject) and multiple dependent views or handlers (observers).
- You want changes in the subject to automatically propagate to observers.
- Observers should not be tightly coupled to the subject.

When to Use



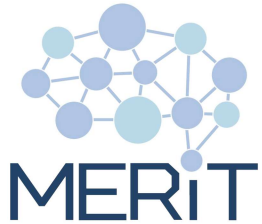
- You need to update multiple objects when a single object changes
- You want to implement event broadcasting
- Building GUIs, live dashboards, pub-sub systems, or data bindings
- Different parts of your system need to react independently to changes

Code




```
class Subject:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self, data):
        for observer in self._observers:
            observer.update(data)
```

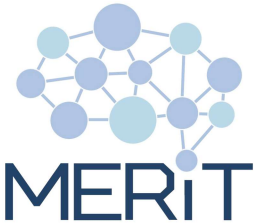
Code




```
class ConcreteObserver:
    def update(self, data):
        print(f"Observer received: {data}")

subject = Subject()
observer1 = ConcreteObserver()
observer2 = ConcreteObserver()

subject.attach(observer1)
subject.attach(observer2)

subject.notify("New event!")  # Both observers get notified
```

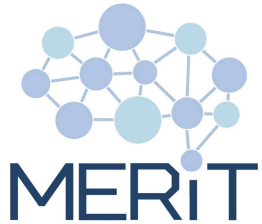
Strategy

Definition



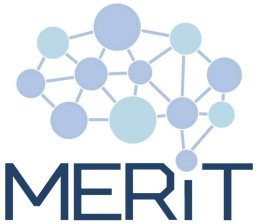
- The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them **interchangeable**.
- It lets you select the **algorithm at runtime**, keeping the context logic **agnostic** to which algorithm is used.

Core Idea



- Instead of hardcoding logic into a class, you extract it into separate strategy classes.
- These strategy classes implement a common interface, so the behavior can be changed without touching the context class.
- Clients can choose or inject different strategies dynamically.

When to Use



- Multiple algorithms or policies can apply to the same task
- You want to avoid large if-else or switch statements
- You want to dynamically switch algorithms
- You want to keep the logic of the decision separate from the implementation

Code

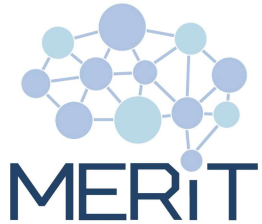
```
class SortStrategy:
    def sort(self, data): pass

class BubbleSort(SortStrategy):
    def sort(self, data):
        print("Sorting with Bubble Sort")
        return sorted(data)  # Simulate it

class QuickSort(SortStrategy):
    def sort(self, data):
        print("Sorting with Quick Sort")
        return sorted(data)  # Simulate it
```

Code

```
class Sorter:
    def __init__(self, strategy: SortStrategy):
        self.strategy = strategy

    def sort(self, data):
        return self.strategy.sort(data)

data = [4, 1, 3]

sorter = Sorter(BubbleSort())
sorter.sort(data)  # Uses BubbleSort

sorter.strategy = QuickSort()
sorter.sort(data)  # Switches to QuickSort
```

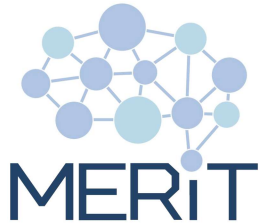
Command

Definition



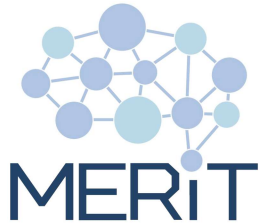
- The Command Pattern encapsulates **a request as an object**, thereby allowing you to:
  - ✔ Parameterize clients with queues, logs, and callbacks
  - ✔ Queue, undo, and redo operations
  - ✔ Decouple the sender of a request from its receiver

Core Idea



- You want to turn actions into objects.
- These objects can be stored, passed around, executed, or undone.
- The invoker (caller) does not need to know who or what executes the action — just holds and triggers the command.

When to Use



- You need to queue, delay, or undo operations
- You want to log and replay actions (macros)
- The sender shouldn’t know the specifics of the receiver
- You need flexible command execution (e.g., scheduling, scripting)

Code

```
class Command:
    def execute(self): pass

class Light:
    def turn_on(self):
        print("Light is ON")
    def turn_off(self):
        print("Light is OFF")
```

Code

```
class TurnOnCommand(Command):
    def __init__(self, light):
        self.light = light
    def execute(self):
        self.light.turn_on()

class TurnOffCommand(Command):
    def __init__(self, light):
        self.light = light
    def execute(self):
        self.light.turn_off()
```

Code

```
class RemoteControl:
    def submit(self, command):
        command.execute()

light = Light()
remote = RemoteControl()

remote.submit(TurnOnCommand(light))  # Light is ON
remote.submit(TurnOffCommand(light)) # Light is OFF
```