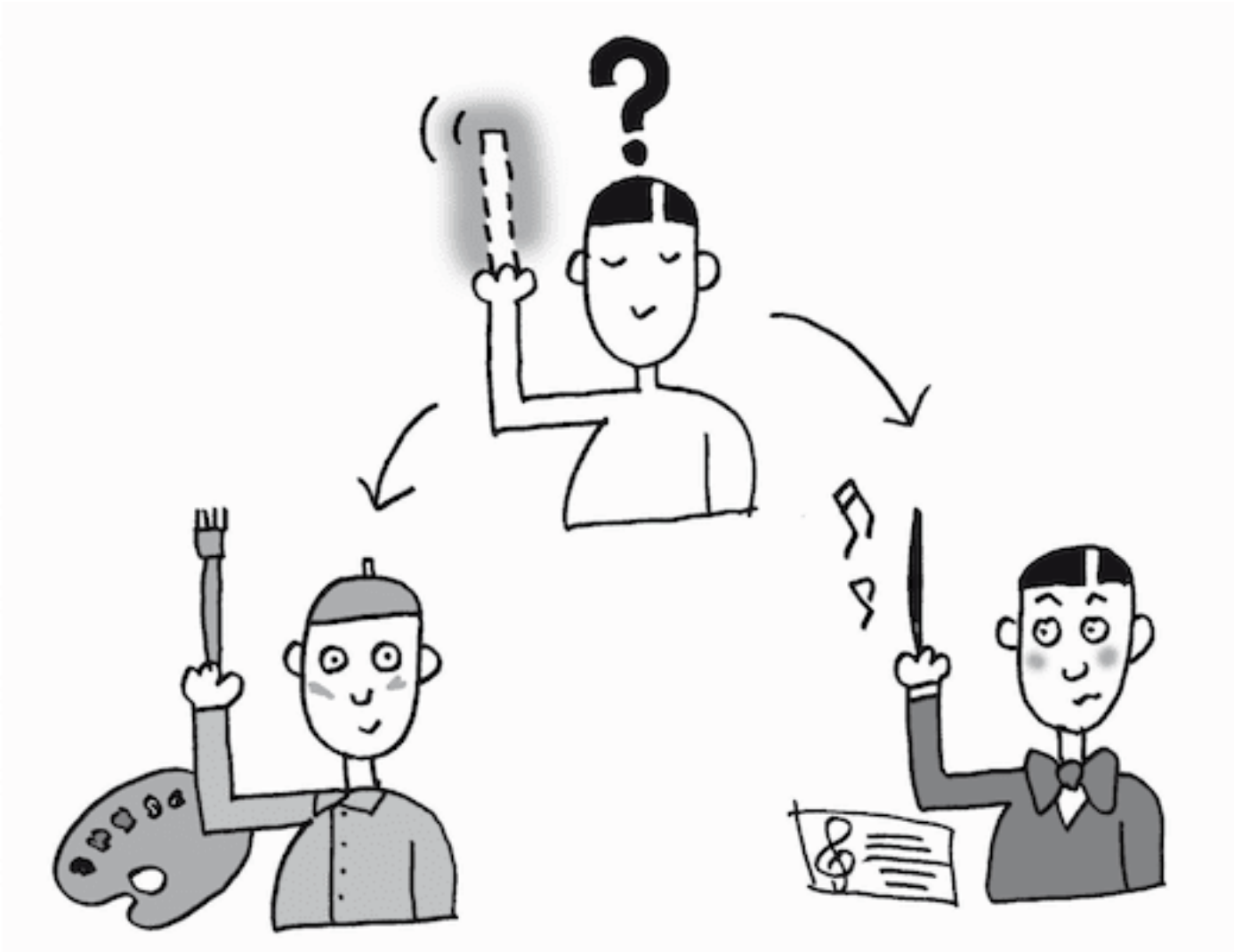
Template Method Pattern

Super classes guide and sub classes implement



Template Method Pattern

Template Method Pattern

When we already have an overall algorithm, we can use it as a template and let subclasses (or specific steps) fill in the details.

- **Cooking Recipe**
  - General steps: *prepare ingredients → cook → serve*
  - Details vary: pasta, curry, or soup.
- **Online Shopping Checkout**
  - Steps: *add to cart → payment → shipping*
  - Payment methods or shipping details differ per store.

The Problem

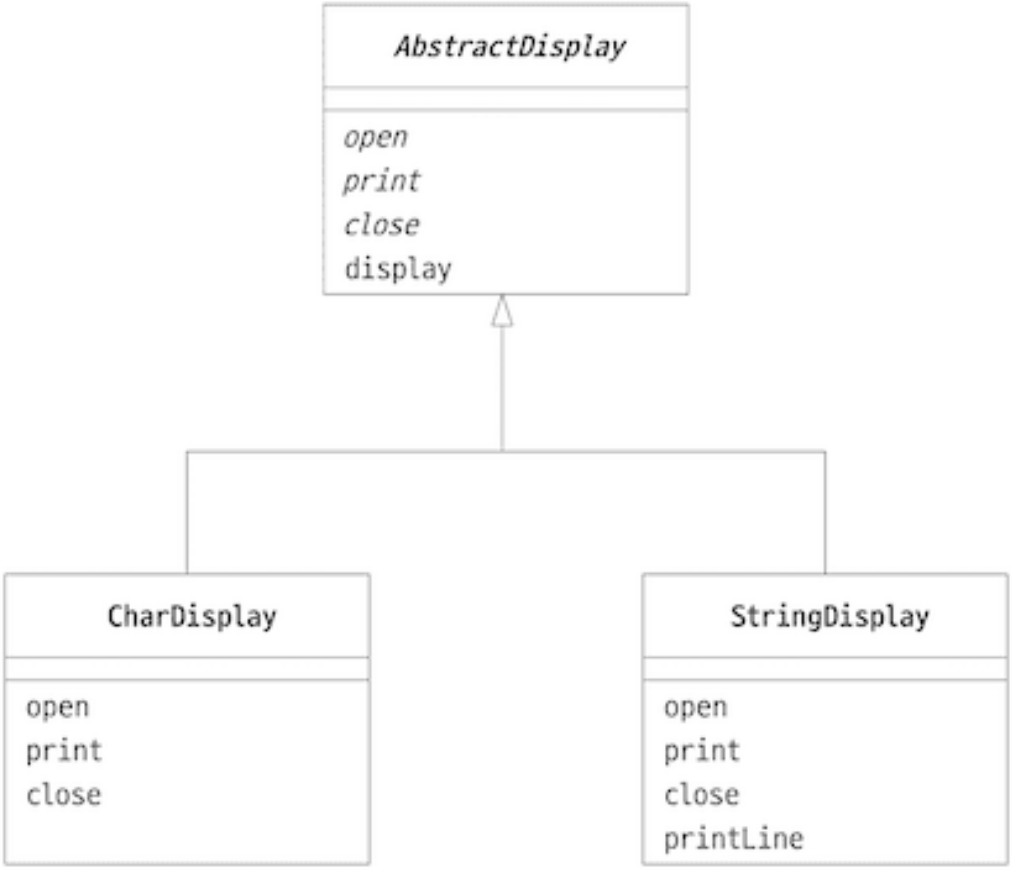
- We already defined overall algorithm that works fine.
- We want to use the same algorithm for different implementations.
    - We have different ways to display content
    - But the algorithm is always the same.

The challenge: how to reuse the same algorithm structure while allowing different implementations for specific steps?

The *Template Method* as the Solution

- Define the skeleton of an algorithm in a method, deferring some steps to subclasses.
- *Template Method* lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- The *abstract class* `controls` the flow, while **concrete classes** `implement` the details.

The Design

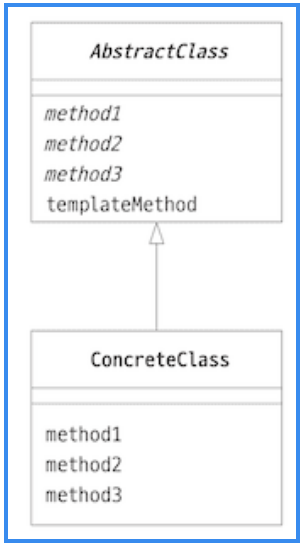

AbstractDisplay
open
print
close
display
CharDisplay
open
print
close
StringDisplay
open
print
close
printLine

Step 1: Understand the Players

In this design, we have players:

- *AbstractClass*
  - Defines the template Method that contains the algorithm skeleton.
  - Provides the abstract methods.
- **ConcreteClass**
  - Implements each abstract method.

Step 2: Separation of *abstraction* and concretion



- The template Method is implemented in the *AbstractClass*.
  - It defines the `algorithm skeleton` by calling abstract methods in a specific order.
- **Concrete classes** cannot change the algorithm structure, only the step implementations.

Code

- Main Method
- AbstractDisplay (Template Method)
- CharDisplay/StringDisplay (Concrete Implementations)

Main Method

```
from char_display import CharDisplay
from string_display import StringDisplay

def main():
    print("=== Template Method Pattern Example ===\n")

    d1 = CharDisplay('H')
    d2 = StringDisplay("Hello, world.")

    print("1. CharDisplay with character 'H':")
    d1.display()  # Uses template method

    print("\n2. StringDisplay with 'Hello, world.':")
    d2.display()  # Uses same template method
```

Step 1: Create concrete instances

```
d1 = CharDisplay('H')
d2 = StringDisplay("Hello, world.")
```

- Both classes inherit from **AbstractDisplay**.
  - It contains the `algorithm skeleton`.
- They implement the abstract methods differently.
  - CharDisplay & StringDisplay implement details differently.

Step 2: Call the template method

```
d1.display()  # Template method in AbstractDisplay
d2.display()  # Same template method, different implementations
```

- The `display()` method is the *template method* (`algorithm skeleton`).
- It calls `open()`, `print()` (5 times), then `close()`.
    - Each method is implemented in the subclasses differently.

Abstract Template Class

```
from abc import ABC, abstractmethod

class AbstractDisplay(ABC):
    @abstractmethod
    def open(self): pass

    @abstractmethod
    def print(self): pass

    @abstractmethod
    def close(self): pass

    def display(self):  # Algorithm!
        self.open()
        for i in range(5):
            self.print()
        self.close()
```

Character Display Implementation

```
class CharDisplay(AbstractDisplay):
    def __init__(self, ch: str):
        self._ch = ch

    def open(self):
        print("<<", end="")

    def print(self):
        print(self._ch, end="")

    def close(self):
        print(">>")
```

String Display Implementation

```
class StringDisplay(AbstractDisplay):
    def __init__(self, string: str):
        self._string = string
        self._width = len(string)

    def open(self):
        self._print_line()

    def print(self):
        print(f"|{self._string}|")

    def close(self):
        self._print_line()

    def _print_line(self):
        print("+" + "-" * self._width + "+")
```

Output Example

CharDisplay('H'):

```
<<HHHHH>>
```

StringDisplay("Hello, world."):

```
+--------------+
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
|Hello, world.|
+--------------+
```

Discussion

- The template method in the should typically be declared as final (non-overridable) to prevent subclasses from changing the algorithm structure.
- A hook method is an optional method with a default (usually empty) implementation that subclasses can override to extend the algorithm at specific points.

- Hollywood Principle(**Don't call us, we'll call you**): the template method calls the subclass methods, not the other way around.
- The Template Method pattern depends on collaboration between superclass and subclass: isolation is not the goal — **controlled extension** is

Template in C++ vs. Template Method (DP)

- **C++ Template** = *language feature* for **generic code** (compile-time, type parameters)
- **Template Method (DP)** = *object-oriented pattern* where a **base class fixes the algorithm** and **subclasses fill steps**

C++ Templates and Template Method solve different problems: One is language-level generics, the other is an OO behavioral pattern.

- "Different types, same algorithm" -> C++ template
- "Same workflow, different steps" -> Template Method (DP)

Template vs Strategy (not discussed yet)

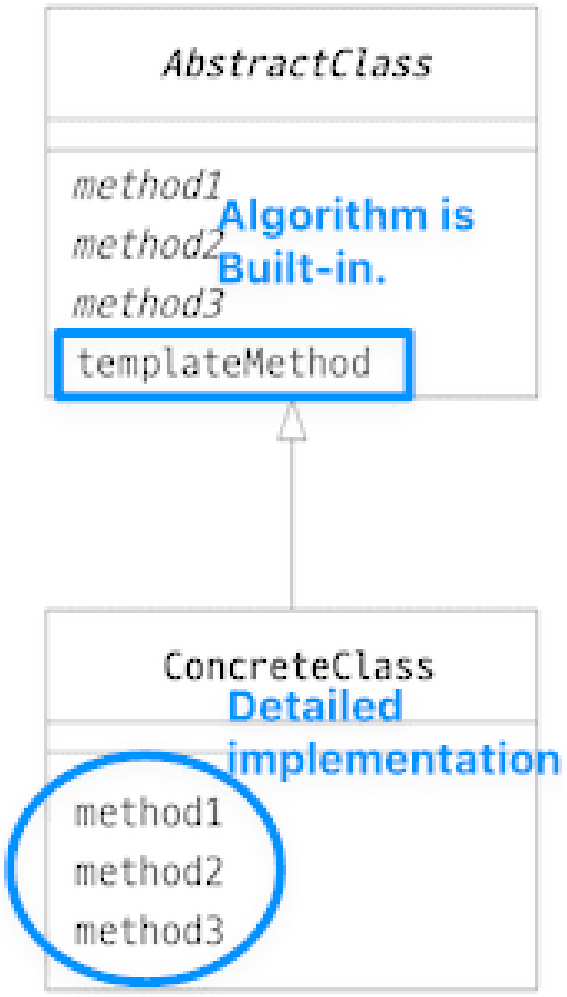
When Template Method does not fit for certain problems, you should consider using the Strategy design pattern.

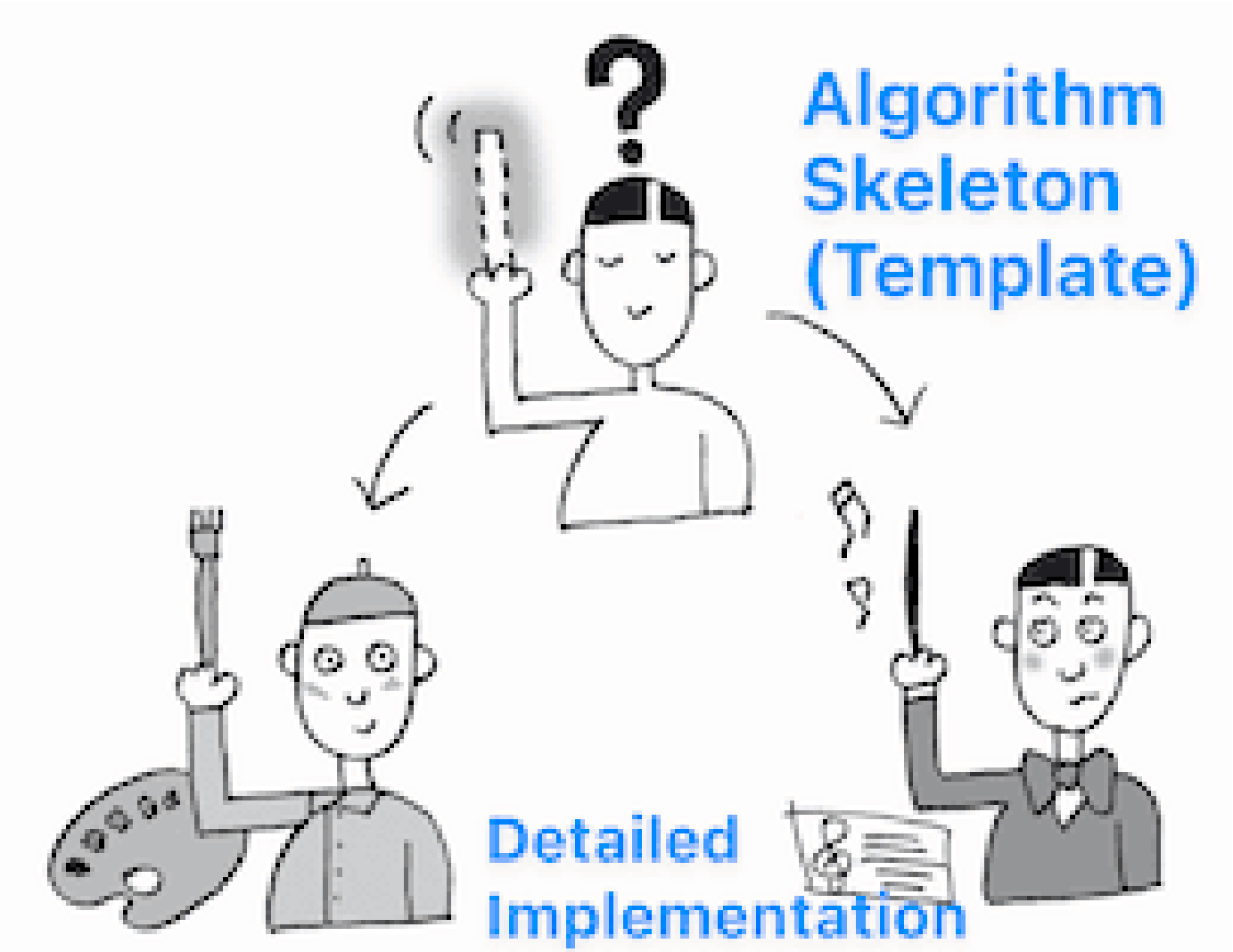
- Template Method: Uses inheritance, defines algorithm skeleton in parent class, subclasses override specific steps.
- Strategy: Uses composition, entire algorithms are interchangeable through different strategy objects.

Rule of Thumb

- Use **Template Method** → when the structure must stay fixed
- Use **Strategy** → when you need full flexibility in choosing algorithms

UML


AbstractClass
method1
method2
method3
templateMethod
Algorithm is
Built-in.
ConcreteClass
Detailed
implementation
method1
method2
method3


Algorithm
Skeleton
(Template)
Detailed
Implementation