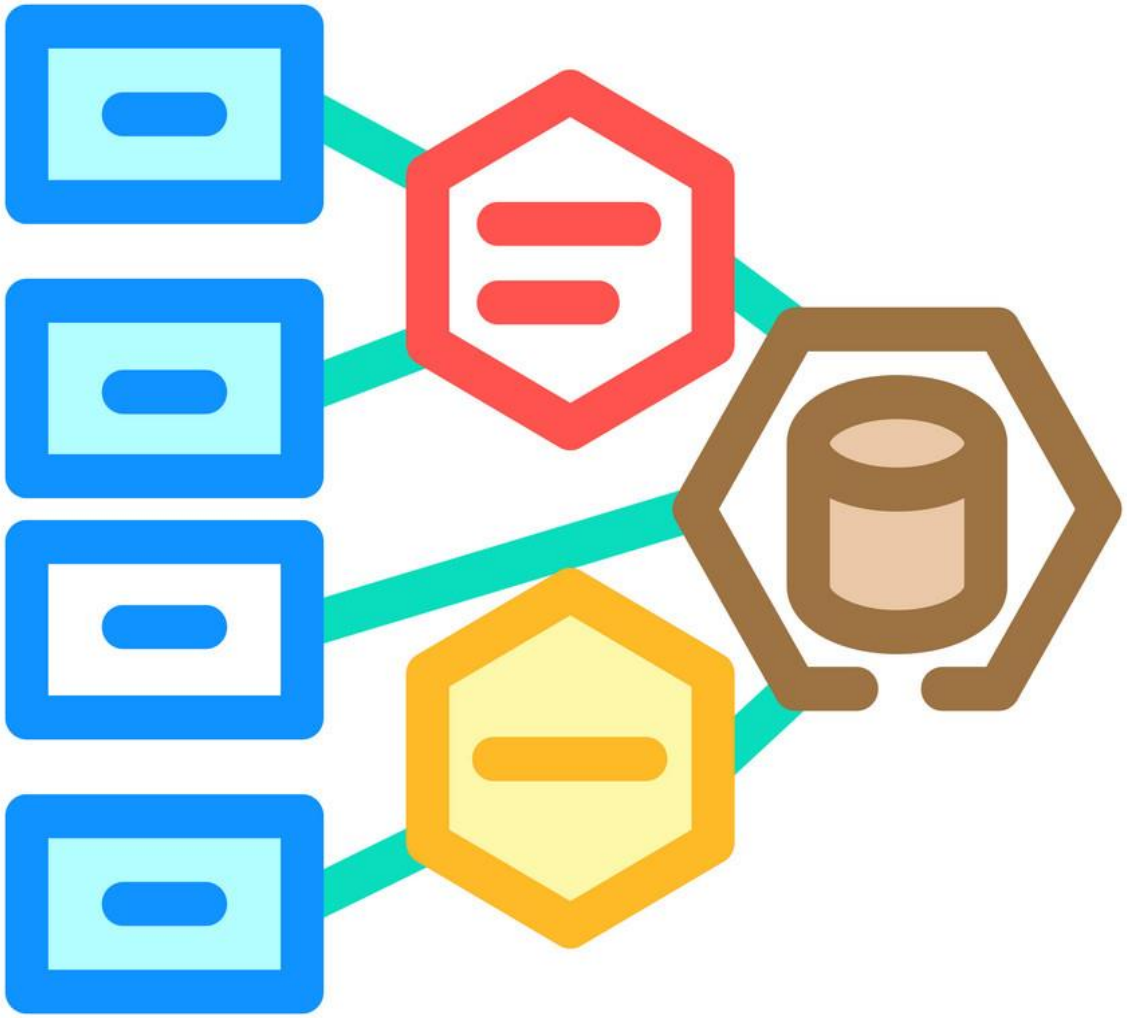
# Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



| Creational patterns | Structural patterns | Behavioral patterns |
| --- | --- | --- |
| Factory | Adapter | Chain of responsibility |
| Abstract factory | Bridge | Command |
| Builder | Composite | Iterator |
| Prototype | Decorator | Mediator |
| Singleton | Façade | Memento |
| | Flyweight | Observer |
| | Proxy | State |
| | | Strategy |
| | | Template method |
| | | Visitor |

**A good link to refer:** https://refactoring.guru/design-patterns

# Structural patterns

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

<<already covered>>

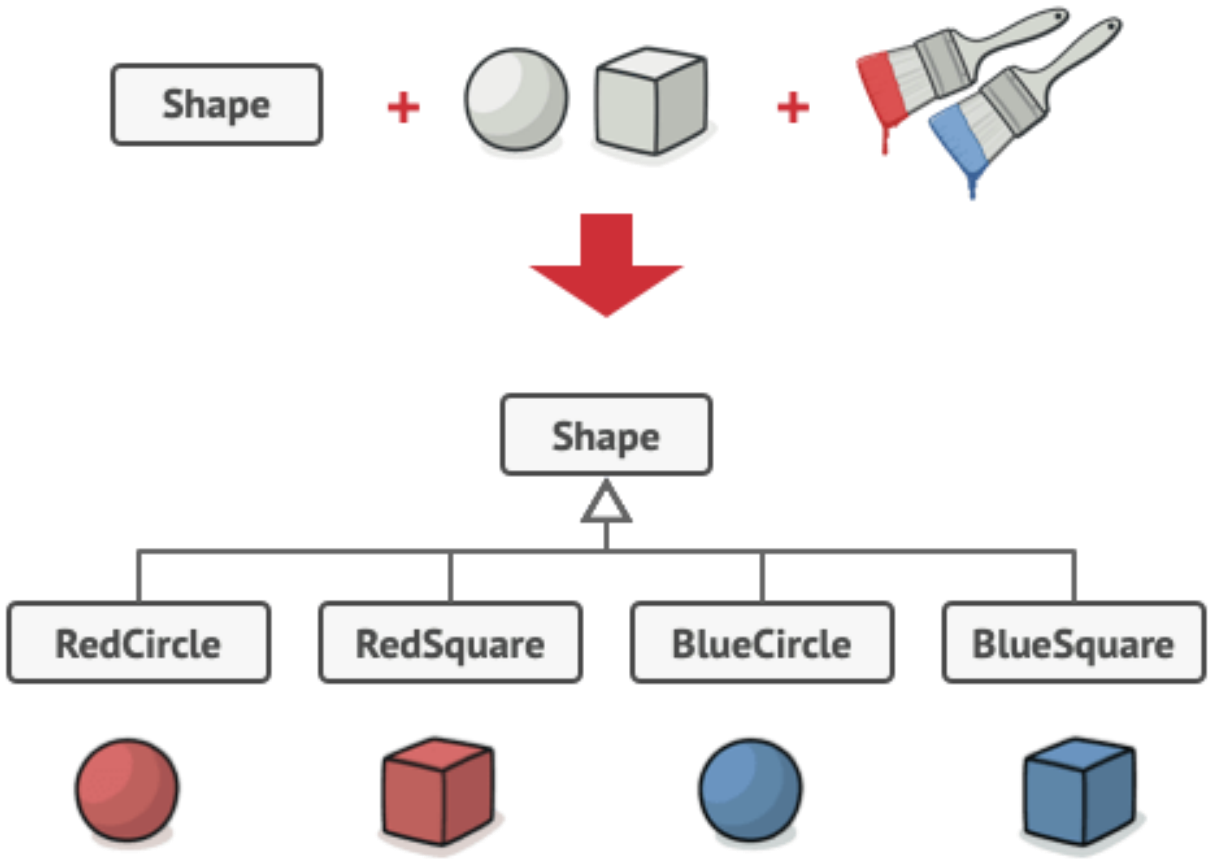
# Structural patterns

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

**Bridge** is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

Say you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses.

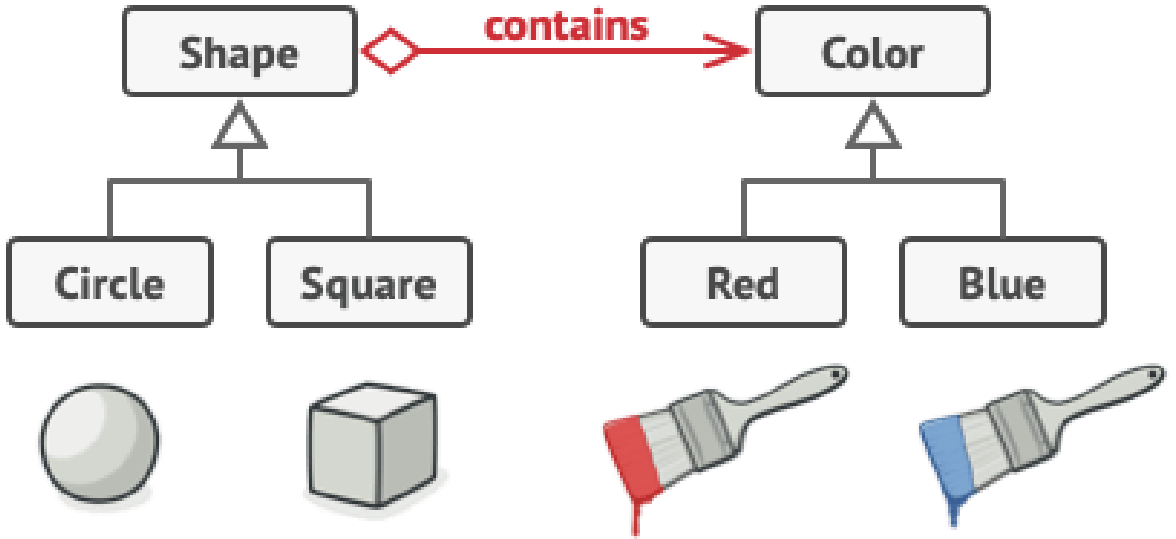
However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare.



Adding new shape types and colors to the hierarchy will grow it exponentially. For example, to add a triangle shape you'd need to introduce two subclasses, one for each color.

And after that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.

- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.
- What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.



- Following this approach, we can extract the color-related code into its own class with two subclasses: Red and Blue.
- The Shape class then gets a reference field pointing to one of the color objects.
- Now the shape can delegate any color-related work to the linked color object.
- That reference will act as a bridge between the Shape and Color classes.
- From now on, adding new colors won’t require changing the shape hierarchy, and vice versa.

## Steps to Implement:

1. **Abstraction**: Define the abstraction interface.
2. **Implementor**: Define the implementation interface.
3. **Concrete Implementors**: Implement different variants of the implementor.
4. **Refined Abstraction**: Extend the abstraction and delegate work to the implementor.

```
# Implementor Interface
class Color:
    def fill(self):
        pass

# Concrete Implementors
class Red(Color):
    def fill(self):
        return "Red color"

class Blue(Color):
    def fill(self):
        return "Blue color"

# Abstraction
class Shape:
    def __init__(self, color):
        self.color = color

    def draw(self):
        pass
```

```
# Refined Abstraction
class Circle(Shape):
    def draw(self):
        return f"Circle filled with {self.color.fill()}"

class Square(Shape):
    def draw(self):
        return f"Square filled with {self.color.fill()}"

# Usage
red = Red()
blue = Blue()

circle = Circle(red)
square = Square(blue)

print(circle.draw())  # Circle filled with Red color
print(square.draw())  # Square filled with Blue color
```

**Abstraction (Shape):** Links to a Color implementor.

**Implementor (Color):** Provides different color implementations (Red, Blue).

**Refined Abstraction (Circle, Square):** Uses Shape but extends behavior.

- You can create platform-independent classes and apps.
- The client code works with high-level abstractions. It isn’t exposed to the platform details.
- Open/Closed Principle. You can introduce new abstractions and implementations independently from each other.
- Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.

You might make the code more complicated by applying the pattern to a highly cohesive class.

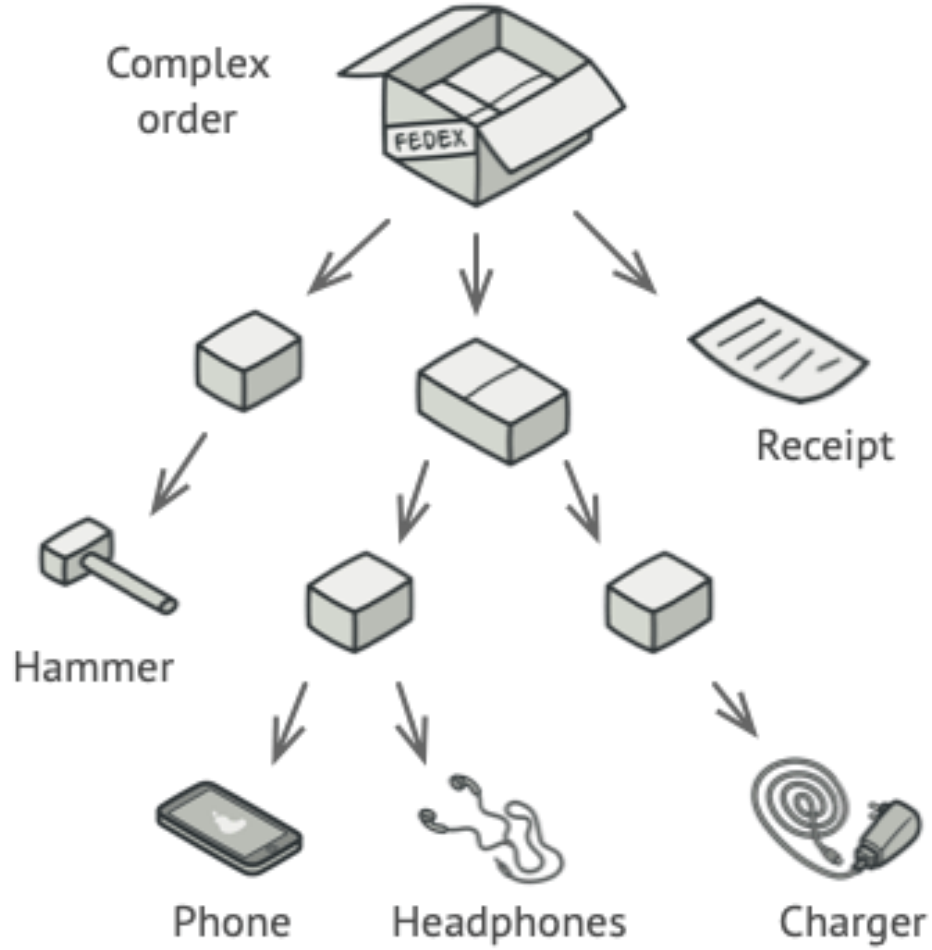
# Structural patterns

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

**Composite** is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

For example, imagine that you have two types of objects: Products and Boxes. A Box can contain several Products as well as a number of smaller Boxes. These little Boxes can also hold some Products or even smaller Boxes, and so on.

Say you decide to create an ordering system that uses these classes. Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes. How would you determine the total price of such an order?



You could try the direct approach: unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world; but in a program, it’s not as simple as running a loop.

You have to know the classes of Products and Boxes you’re going through, the nesting level of the boxes and other nasty details beforehand. All of this makes the direct approach either too awkward or even impossible.

- The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.

- How would this method work? For a product, it’d simply return the product’s price.
- For a box, it’d go over each item the box contains, ask its price and then return a total for this box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated.
- A box could even add some extra cost to the final price, such as packaging cost.

**Implementation: File System Example**

*Classes:*

- Component: Abstract class/interface (e.g., FileSystemItem).
- Leaf: Represents individual objects (e.g., File).
- Composite: Represents a collection of objects (e.g., Folder)

```
from abc import ABC, abstractmethod

# Component
class FileSystemItem(ABC):
    @abstractmethod
    def display(self, indent=0):
        pass

# Leaf
class File(FileSystemItem):
    def __init__(self, name):
        self.name = name

    def display(self, indent=0):
        print(" " * indent + self.name)
```

```
# Composite
class Folder(FileSystemItem):
    def __init__(self, name):
        self.name = name
        self.children = []

    def add(self, item):
        self.children.append(item)

    def remove(self, item):
        self.children.remove(item)

    def display(self, indent=0):
        print(" " * indent + self.name)
        for child in self.children:
            child.display(indent + 2)
```

```
# Client Code
if __name__ == "__main__":
    # Create Files
    file1 = File("File1.txt")
    file2 = File("File2.txt")
    file3 = File("File3.txt")

    # Create Folders
    folder1 = Folder("Folder1")
    folder2 = Folder("Folder2")

    # Build the structure
    folder1.add(file1)
    folder1.add(file2)
    folder2.add(file3)
    folder1.add(folder2)

    # Display the structure
    folder1.display()
```

1. **Component**: A general "blueprint" for objects in the system.
    - In the example, FileSystemItem is this blueprint, defining the display() method.
2. **Leaf**: Represents basic objects that do not contain other objects.
    - Example: File (a single file like File1.txt).
3. **Composite**: Represents objects that *can contain* other objects (both Leaf and Composite).
    - Example: Folder (it can contain files or other folders).

In short:

- **File** = individual file (smallest unit, no children).
- **Folder** = group of files or folders (can have children).
- **FileSystemItem** = ensures both File and Folder can be treated the same way.

- You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

- It might be difficult to provide a common interface for classes whose functionality differs too much.
- In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

# Structural patterns

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

**Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Imagine that you’re working on a notification library which lets other programs notify their users about important events.

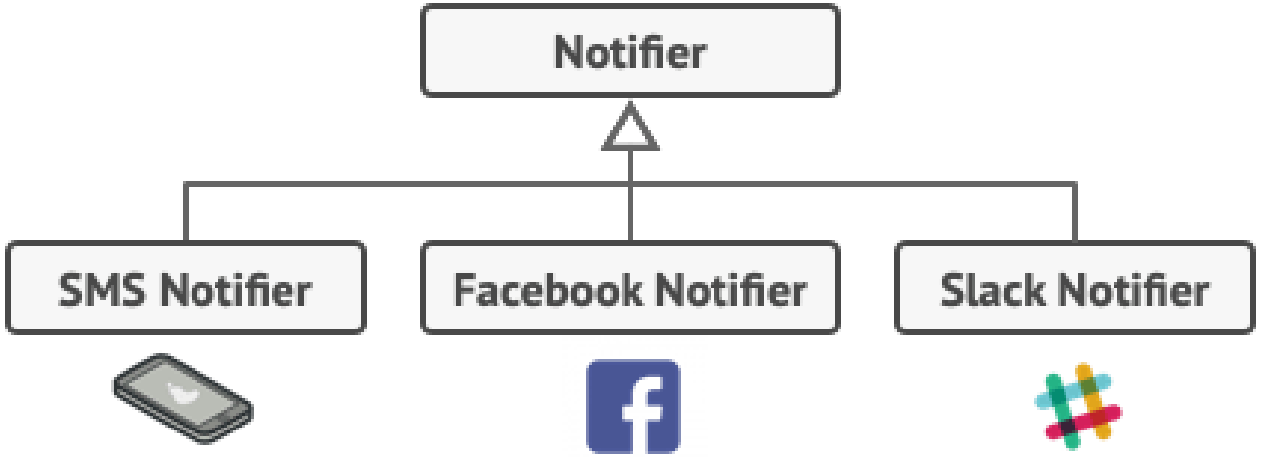
The initial version of the library was based on the Notifier class that had only a few fields, a constructor and a single send method.

The method could accept a message argument from a client and send the message to a list of emails that were passed to the notifier via its constructor.

A third-party app which acted as a client was supposed to create and configure the notifier object once, and then use it each time something important happened.

At some point, you realize that users of the library expect more than just email notifications.

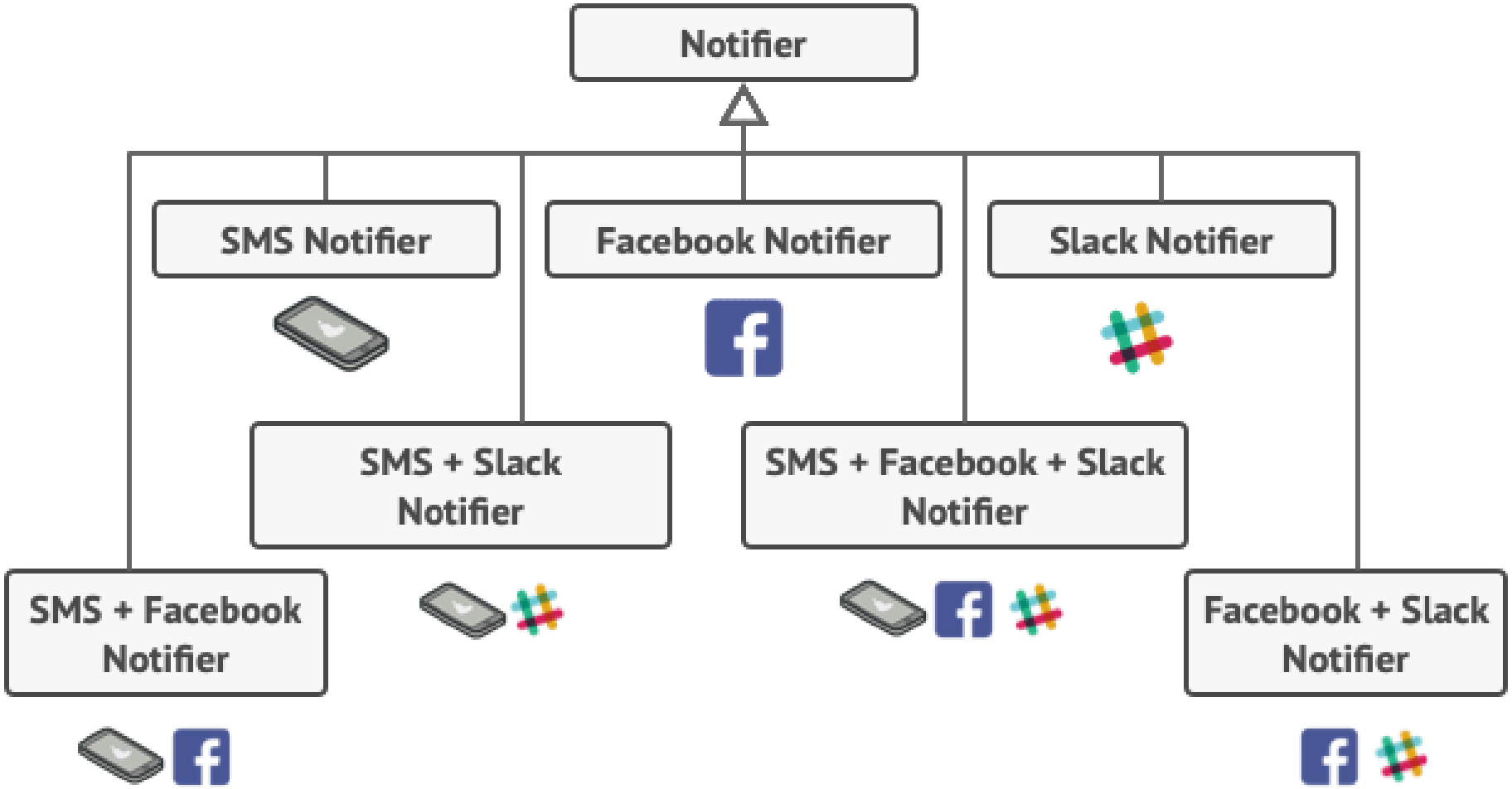
Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.



How hard can that be? You extended the Notifier class and put the additional notification methods into new subclasses. Now the client was supposed to instantiate the desired notification class and use it for all further notifications.

But then someone reasonably asked you, “Why can’t you use several notification types at once? If your house is on fire, you’d probably want to be informed through every channel.”

You tried to address that problem by creating special subclasses which combined several notification methods within one class. However, it quickly became apparent that this approach would bloat the code immensely, not only the library code but the client code as well.

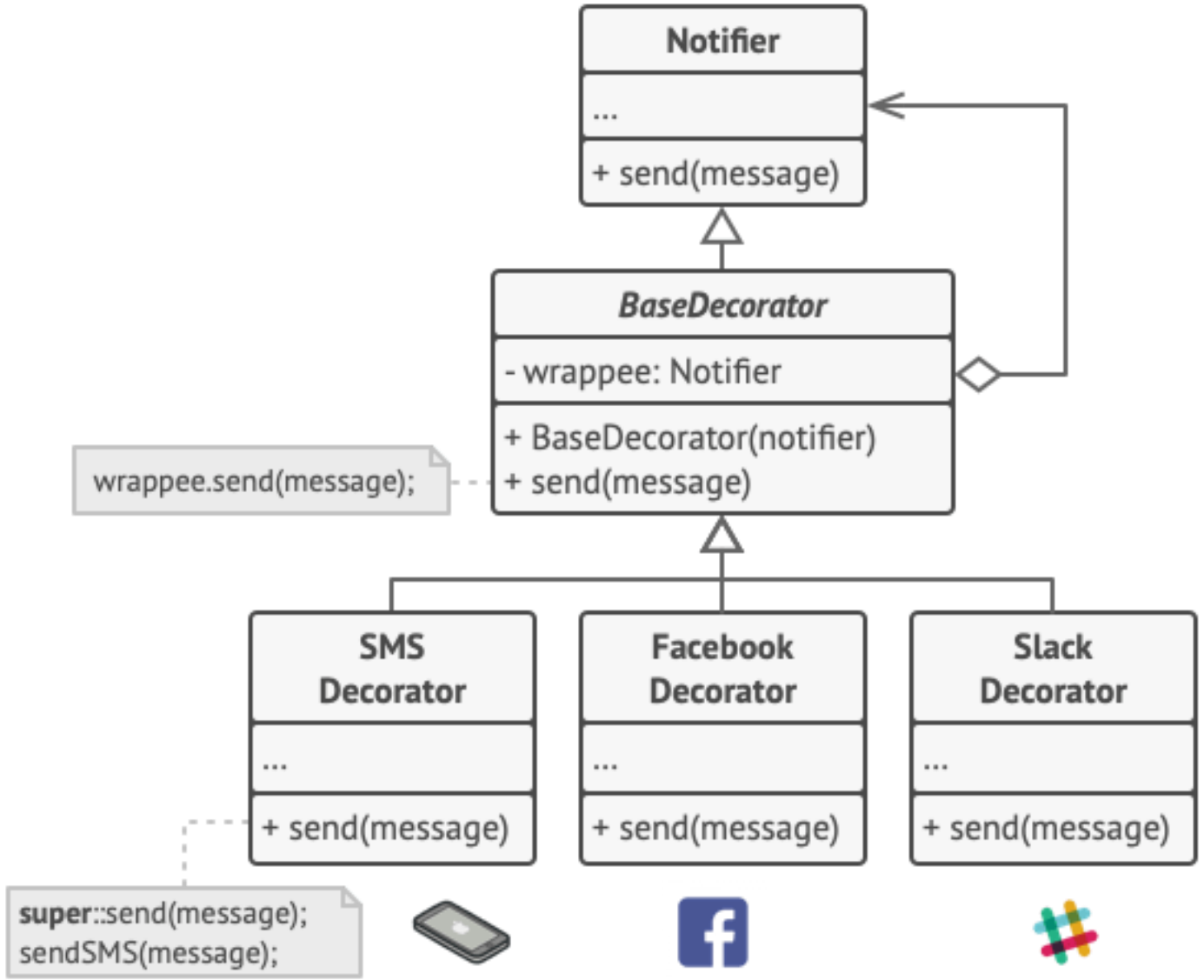

Notifier
SMS Notifier
Facebook Notifier
Slack Notifier
SMS + Slack Notifier
SMS + Facebook + Slack Notifier
SMS + Facebook Notifier
Facebook + Slack Notifier

- Extending a class is the first thing that comes to mind when you need to alter an object's behavior. However, inheritance has several serious caveats that you need to be aware of.
- **Inheritance is static**. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.
- **Subclasses can have just one parent class**. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.

- One of the ways to overcome these is by using Aggregation or Composition  instead of Inheritance.
- Both of the alternatives work almost the same way: one object has a reference to another and delegates it some work, whereas with inheritance, the object itself is able to do that work, inheriting the behavior from its superclass.

- With this new approach you can easily substitute the linked “helper” object with another, changing the behavior of the container at runtime.
- An object can use the behavior of various classes, having references to multiple objects and delegating them all kinds of work.

## *Various notification methods become decorators.*



## Steps to Implement:

1. **Define a component interface** that declares a method(s) that can be extended.
2. **Implement a concrete component** that implements the component interface.
3. **Create an abstract decorator** that implements the same interface and has a reference to a component.
4. **Create concrete decorators** that extend the abstract decorator and add functionality.

## A Simple Coffee Example

coffee shop where we can decorate a basic coffee with different add-ons (like milk or sugar).

```
# Step 1: Component interface
class Coffee:
    def cost(self):
        pass

# Step 2: ConcreteComponent
class SimpleCoffee(Coffee):
    def cost(self):
        return 5  # Basic cost of a simple coffee
```

```
# Step 3: Abstract Decorator
class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self._coffee = coffee

    def cost(self):
        return self._coffee.cost()

# Step 4: Concrete Decorators
class MilkDecorator(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 2  # Adding milk cost
```

```
class SugarDecorator(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 1  # Adding sugar cost

# Using the decorators
simple_coffee = SimpleCoffee()
print(f"Simple Coffee Cost: ${simple_coffee.cost()}")

milk_coffee = MilkDecorator(simple_coffee)
print(f"Milk Coffee Cost: ${milk_coffee.cost()}")

sugar_milk_coffee = SugarDecorator(milk_coffee)
print(f"Sugar and Milk Coffee Cost:
${sugar_milk_coffee.cost()}")
```

- You can extend an object’s behavior without making a new subclass.
- You can add or remove responsibilities from an object at runtime.
- You can combine several behaviors by wrapping an object into multiple decorators.
- Single Responsibility Principle. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

- It’s hard to remove a specific wrapper from the wrappers stack.
- It’s hard to implement a decorator in such a way that its behavior doesn’t depend on the order in the decorators stack.
- The initial configuration code of layers might look pretty ugly.

# Structural patterns

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

**Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Essentially, it acts as a "front door" to the subsystem, allowing users to interact with it without needing to understand the internal details.

Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.

As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

- A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

- Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

- For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method encode(filename, format). After creating such a class and connecting it with the video conversion library, you'll have your first facade.

**Steps to Implement the Facade Pattern:**

1. **Identify the Subsystems**: Break down the complex system into smaller subsystems.
2. **Create a Facade Class**: This class will provide a simplified interface to the subsystems, consolidating multiple operations into a single method.
3. **Client Uses the Facade**: The client interacts only with the facade, unaware of the subsystems it manages.

**Example: Home Theater System (Facade)**

**Subsystems:**

- **Projector**: Turns on and off, adjusts settings.
- **SoundSystem**: Manages volume and power.
- **Lights**: Controls room lighting.
- **DVDPlayer**: Plays movies.

**Facade:**

The HomeTheaterFacade simplifies interaction by providing a method to start the movie, which automatically manages interactions with the subsystems

```
# Subsystems (complex operations)
class Projector:
    def on(self):
        print("Projector is on.")

    def off(self):
        print("Projector is off.")

    def set_input(self, input_device):
        print(f"Projector input set to {input_device}.")

class SoundSystem:
    def on(self):
        print("Sound system is on.")

    def off(self):
        print("Sound system is off.")

    def set_volume(self, level):
        print(f"Sound system volume set to {level}.")
```

```
class Lights:
    def dim(self, level):
        print(f"Lights dimmed to {level}%.")

    def on(self):
        print("Lights are on.")

class DVDPlayer:
    def on(self):
        print("DVD player is on.")

    def off(self):
        print("DVD player is off.")

    def play(self, movie):
        print(f"Playing {movie} on the DVD player.")

    def stop(self):
        print("DVD player stopped.")
```

```
# Facade Class
class HomeTheaterFacade:
    def __init__(self, projector, sound_system, lights, dvd_player):
        self.projector = projector
        self.sound_system = sound_system
        self.lights = lights
        self.dvd_player = dvd_player

    # Simplified interface to start the movie
    def watch_movie(self, movie):
        print("Get ready to watch a movie...")
        self.lights.dim(10)
        self.projector.on()
        self.projector.set_input("DVD Player")
        self.sound_system.on()
        self.sound_system.set_volume(5)
        self.dvd_player.on()
        self.dvd_player.play(movie)
```

```
    # Simplified interface to stop the movie
    def end_movie(self):
        print("Shutting down the home theater...")
        self.lights.on()
        self.sound_system.off()
        self.projector.off()
        self.dvd_player.off()

# Client Code
if __name__ == "__main__":
    # Create subsystems
    projector = Projector()
    sound_system = SoundSystem()
    lights = Lights()
    dvd_player = DVDPlayer()

    # Create facade
    home_theater = HomeTheaterFacade(projector, sound_system, lights, dvd_player)

    # Client interacts only with the facade
    home_theater.watch_movie("The Matrix")
    print("\nMovie is over.")
    home_theater.end_movie()
```

In a home theater system, there might be several components like a projector, a sound system, a DVD player, and lights. The

Facade could provide a simplified interface like turnOnMovieMode() that automatically handles the interactions between these components.

Without the facade, the client would have to call individual methods on each component, which could be more complex and error-prone.

```
class Subsystem1:
    def operation1(self):
        print("Subsystem1: operation1")

class Subsystem2:
    def operation2(self):
        print("Subsystem2: operation2")

class Subsystem3:
    def operation3(self):
        print("Subsystem3: operation3")
```

```
class Facade:
    def __init__(self):
        self.subsystem1 = Subsystem1()
        self.subsystem2 = Subsystem2()
        self.subsystem3 = Subsystem3()

    def simplifiedOperation(self):
        print("Facade: Simplified operation started")
        self.subsystem1.operation1()
        self.subsystem2.operation2()
        self.subsystem3.operation3()
        print("Facade: Simplified operation finished")

# Client code
facade = Facade()
facade.simplifiedOperation()
```

In this example:

- The client interacts with the Facade class, calling simplifiedOperation(), instead of dealing with the individual subsystems (Subsystem1, Subsystem2, and Subsystem3).
- The facade handles the coordination between subsystems, making it easier for the client to perform the required operations.

- You can isolate your code from the complexity of a subsystem.

A facade can become a god object coupled to all classes of an app.

a god object is an object that references a large number of distinct types, has too many unrelated or uncategorized methods, or some combination of both

The god object is an example of an anti-pattern and a code smell.

# Structural patterns

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

**Flyweight** is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

**Key Concepts:**

- **Intrinsic State:** Shared, unchanging data (e.g., shape, color).
- **Extrinsic State:** Unique, context-specific data (e.g., position).
- **Flyweight Object:** The shared object containing intrinsic state.

**Game Development:** In games like chess or checkers, pieces of the same type (e.g., pawns) share their shape or color but have unique board positions.

**Map Applications:** Reuse icons (e.g., for trees, buildings) while storing their unique GPS locations externally.

**GUI Elements:** Buttons, checkboxes, or icons in a UI toolkit reuse common graphical data, storing unique properties like position or label separately.

The **Flyweight Pattern** involves the following steps:

1. **Flyweight Interface:** Defines common methods for shared objects.
2. **Concrete Flyweight Class:** Implements the flyweight interface and stores intrinsic (shared) state.
3. **Flyweight Factory:** Manages and returns shared flyweight objects.
4. **Client Code:** Maintains extrinsic (unique) state and interacts with flyweight objects.

```
# Flyweight Class
class CharacterFlyweight:
    def __init__(self, character):
        self.character = character  # Intrinsic (shared) state

    def display(self, font_size):
        print(f"Character: {self.character}, Font size: {font_size}")  # 
Extrinsic state


# Flyweight Factory
class FlyweightFactory:
    _flyweights = {}

    @staticmethod
    def get_flyweight(character):
        if character not in FlyweightFactory._flyweights:
            FlyweightFactory._flyweights[character] = 
CharacterFlyweight(character)
        return FlyweightFactory._flyweights[character]
```

```
# Client Code
if __name__ == "__main__":
    factory = FlyweightFactory()

    # Create shared flyweights
    a1 = factory.get_flyweight('A')
    a2 = factory.get_flyweight('A')
    b1 = factory.get_flyweight('B')

    # Verify that 'A' flyweights are the same instance
    print(a1 is a2)  # Output: True

    # Display characters with different font sizes (extrinsic state)
    a1.display(font_size=12)
    a2.display(font_size=18)
    b1.display(font_size=10)
```

- You can save lots of RAM, assuming your program has tons of similar objects.

- You might be trading RAM over CPU cycles when some of the context data needs to be recalculated each time somebody calls a flyweight method.
- The code becomes much more complicated. New team members will always be wondering why the state of an entity was separated in such a way.

# Structural patterns

- ✓ Adapter
- ✓ Bridge
- ✓ Composite
- ✓ Decorator
- ✓ Façade
- ✓ Flyweight
- ✓ Proxy

The **Proxy Pattern** is a structural design pattern that provides a substitute or placeholder to control access to another object. It acts as an intermediary, adding extra functionality like lazy initialization, logging, or access control.

The **Proxy Pattern** solves the problem of controlling access to an object that is **Expensive to create** (e.g., large objects or resources).

Imagine a **high-resolution image viewer** app. Loading high-res images is **slow** and **memory-intensive**, but the app needs to display thumbnails instantly.

**Without Proxy:**
The app directly loads the image object. Users face delays as the app waits for full image loading.

**With Proxy:**

A **proxy object** represents the image. Initially, it shows a placeholder or low-res version (thumbnail). The high-res image is **loaded lazily** only when the user views it in full screen.

This improves performance and user experience while managing resource usage.

In the **Proxy Pattern** context:

**1.Service Object**:

The actual object that performs the real operations or provides the core functionality.

- Example: RealImage in the example, which loads and displays high-resolution images.

**2.Client**:

The code or user interacting with the service object, often via the proxy.

- Example: The code invoking image.display() in the example.

The **proxy** acts as a middleman between the client and the service object.

```
from time import sleep

# Real object
class RealImage:
    def __init__(self, filename):
        self.filename = filename
        self.load_image()

    def load_image(self):
        print(f"Loading high-resolution image: {self.filename}...")
        sleep(2)  # Simulate a slow load
        print(f"Image {self.filename} loaded!")

    def display(self):
        print(f"Displaying high-resolution image: {self.filename}")
```

```
# Proxy object
class ImageProxy:
    def __init__(self, filename):
        self.filename = filename
        self.real_image = None  # Real object is not loaded yet

    def display(self):
        if self.real_image is None:
            print(f"Using proxy for {self.filename}. Loading image on
demand...")
            self.real_image = RealImage(self.filename)  # Load real image
lazily
        self.real_image.display()

# Client code
print("App starts")
image = ImageProxy("photo.jpg")

print("\nDisplaying thumbnail (no high-res load yet)")
# Proxy handles the thumbnail (not implemented here)

print("\nDisplaying high-res image")
image.display()  # High-res image is loaded now

print("\nDisplaying high-res image again")
image.display()  # No reload; uses cached real image
```

- You can control the service object without clients knowing about it.
- You can manage the lifecycle of the service object when clients don’t care about it.
- The proxy works even if the service object isn’t ready or is not available.
- Open/Closed Principle. You can introduce new proxies without changing the service or clients.

- The code may become more complicated since you need to introduce a lot of new classes.
- The response from the service might get delayed.

  **Network Latency**: If the service object is remote (e.g., on a server), network delays can occur.

  Example: Accessing a cloud-based file storage.

  **Resource-Intensive Operations**: If the service performs complex computations or large data loads, it may take time.

  Example: Rendering a high-resolution image.

# That’s it