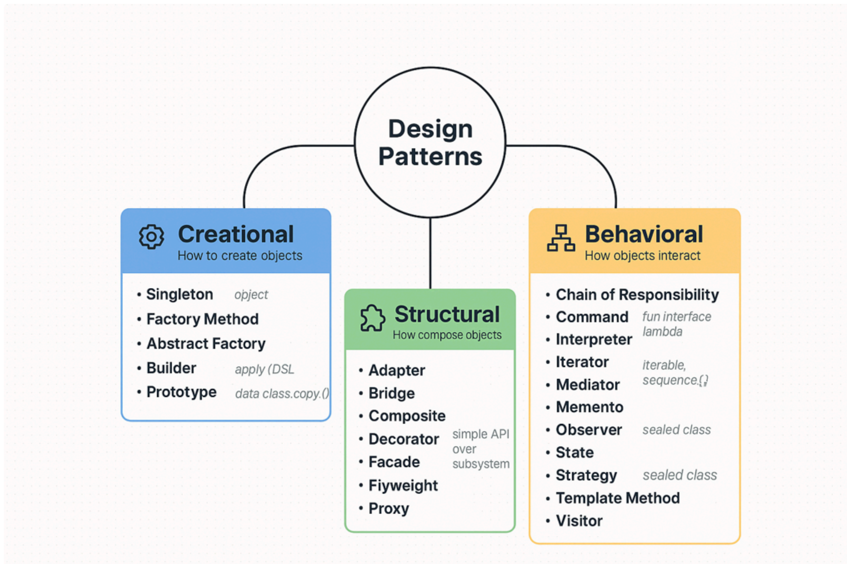
🧐 Design Patterns in Kotlin

Design patterns are proven solutions to recurring problems in software design. They provide a **structured approach** to solve common development challenges, making your code **more maintainable**, **reusable**, and **scalable**.



Why Use Design Patterns?

1. **Code Reusability**: Reuse proven solutions instead of reinventing the wheel.
2. **Improved Readability**: Enhance clarity and structure for better collaboration.
3. **Scalability**: Easily extend or modify your code without introducing bugs.
4. **Industry Standard**: Familiar to developers, making onboarding easier

Creational Patterns

Creational patterns focus on **object creation** mechanisms, ensuring the right objects are created efficiently and appropriately for the situation.

Singleton Pattern

The **Singleton pattern** ensures that a class has only **one instance** throughout the application and provides the global point of access to that instance.

1. constructor private
2. object create with the help of method
3. create field to store the object is private

Problem: You need one shared coordinator (logging, config), and creating many instances would cause conflicts.

Example: Database Connection

```kotlin
object DatabaseConnection {
    fun connect() {
```

```
        println("Connected to the database.")
    }
}

fun main() {
    // Access the Singleton instance
    DatabaseConnection.connect()
}
```

Explanation:

- The `object` keyword in Kotlin automatically creates a **thread-safe Singleton** instance.
- Useful when only one instance of a class is needed, like a **logging service** or a **configuration manager**.

Real-World Use Case

- Managing a single instance of a **database connection** or a **shared preference manager**.

Factory Pattern

The **Factory pattern** provides a way to create objects without specifying their exact class. It **hides the object creation logic from the client**.

Sure! Let's make Factory Method super simple.

- You want to **create objects** but you don't want the main code to know **which exact class** to `new`.
- So you call a **factory method** (a function) that **decides what to create** and gives you back the right object.

Think of a **delivery app**: you press "Send message", and depending on settings it uses **SMS** or **Email**. Your screen shouldn't care which one—it just says "send".

---

1) The common interface (what the app needs)

```
interface MessageSender {
    fun send(text: String)
}

```

2) Two concrete products (different ways to do the job)

```
class SmsSender : MessageSender {
    override fun send(text: String) = println("📱 SMS: $text")
}

class EmailSender : MessageSender {
    override fun send(text: String) = println("📧 Email: $text")
}

```

3) The Factory Method lives in a base class

- The base class knows **when** to send, but not **which** sender to use.

- Subclasses decide **what** to create by overriding `createSender()`.

```kotlin
abstract class Notifier {
    // Factory Method — subclasses will choose the concrete sender
    protected abstract fun createSender(): MessageSender

    // High-level logic stays the same
    fun notifyUser(text: String) {
        val sender = createSender() // ← we don’t know if this is SMS or Email
        sender.send(text)
    }
}

class SmsNotifier : Notifier() {
    override fun createSender(): MessageSender = SmsSender()
}

class EmailNotifier : Notifier() {
    override fun createSender(): MessageSender = EmailSender()
}

```

4) Client code (simple to use)

```kotlin
fun main() {
    val n1: Notifier = SmsNotifier()
    n1.notifyUser("Your OTP is 1234")   // 📱 SMS: Your OTP is 1234

    val n2: Notifier = EmailNotifier()
    n2.notifyUser("Welcome to our app") // 📧 Email: Welcome to our app
}

```

What’s happening (intention behind lines)

- `MessageSender` = **contract** (“I can send a message”).
- `SmsSender` / `EmailSender` = **concrete implementations**.
- `Notifier.notifyUser()` = **fixed workflow** (create → send).
- `createSender()` = **Factory Method**. It’s `abstract` so **subclasses decide** the exact class.
- `SmsNotifier` / `EmailNotifier` = **deciders** that plug in the right product without changing `notifyUser()`.

When should I use Factory Method?

- You have **one workflow**, but the **product type varies** (platform, theme, channel).
- You want to **avoid big** `if/else` in your high-level code.
- You want **easy testing**: inject a subclass that returns a test/dummy product.

Android angle (relatable examples)

- `ViewModelProvider.Factory` creates different **ViewModels** without Activities knowing exact classes.
- `WorkerFactory` in WorkManager creates different **Workers**.
- In your app, you could have `ImageLoaderFactory` that returns **Coil** or **Glide** loaders based on build flavor.

Bonus: “Simple Factory” vs “Factory Method”

- **Simple Factory** = just a function that switches and returns the right object.

```
object SenderFactory {
    fun get(kind: String): MessageSender =
        if (kind == "sms") SmsSender() else EmailSender()
}

```

  Easy, but the **decision lives in one place** (often uses `if/when`).

- **Factory Method** = put the decision in **subclasses** via an overridable method (`createSender()`).
  Better when you expect **new types** and want to avoid editing a big `when` each time.

If you want, I can show the **same example with unit tests** or a **Compose demo** (button chooses SMS/Email at runtime).

Explanation:

- The factory method encapsulates the creation logic.
- This makes your code extensible. If a new type of notification is added, you don’t need to change the client code.

Real-World Use Case

- Creating UI components dynamically based on user input.

Builder Pattern

The **Builder pattern** is used to construct complex objects step by step.

While creating object when object contain many attributes there are many problem exists.

1. We have to pass many arguments to create object.
2. Factory does take responsibilty of creating object . If object is heavy then all complexity is the part of factory class. So in

Example: Car Builder

```
class Car private constructor(
    val engine: String,
    val seats: Int,
    val color: String
) {
```

```kotlin
    data class Builder(
        var engine: String = "Default Engine",
        var seats: Int = 4,
        var color: String = "White"
    ) {
        fun setEngine(engine: String) = apply { this.engine = engine }
        fun setSeats(seats: Int) = apply { this.seats = seats }
        fun setColor(color: String) = apply { this.color = color }
        fun build() = Car(engine, seats, color)
    }
}

fun main() {
    val car = Car.Builder()
        .setEngine("V8 Engine")
        .setSeats(2)
        .setColor("Red")
        .build()
    println("Car built: Engine=${car.engine}, Seats=${car.seats}, Color=${car.color}")
}
```

Explanation:

- The **Builder** separates the object creation process into discrete steps.
- This is particularly useful for creating immutable objects.

Real-World Use Case

- Creating complex objects like **network request configurations** or **UI components**.

---

2. Structural Patterns

Structural patterns focus on organizing classes and objects for **efficient relationships**.

Adapter Pattern

The **Adapter pattern** allows two incompatible interfaces to work together by wrapping an existing class with a new interface.

Example: Audio and Video Player

Explanation:

- The adapter converts the **VideoPlayer** interface into a format that the client expects.

here's the **Adapter pattern** in super-simple terms

---

think of a travel plug 🔌

your phone charger (type-C) doesn't fit the wall socket (type-D).
you use a **plug adapter** that *matches the wall's shape* outside and *your charger's shape* inside.

software adapter = same idea:

- **client expects:** `Target` interface (e.g., `MediaPlayer.play()`)
- **you have:** an existing class with a *different* API (e.g., `VideoPlayer.playVideo()`)

- **adapter:** implements `Target` and **translates** calls to the existing class.

```
Client → Target (MediaPlayer)
           ↑
           │ implements
      Adapter (VideoPlayerAdapter) → Adaptee (VideoPlayer)

```

```kotlin
interface MediaPlayer { fun play() }  // Target the app expects

class AudioPlayer : MediaPlayer {      // Works natively
    override fun play() = println("Playing audio")
}

class VideoPlayer {                        // Incompatible API (no play())
    fun playVideo() = println("Playing video")
}

class VideoPlayerAdapter(               // Adapter: looks like MediaPlayer...
    private val videoPlayer: VideoPlayer
) : MediaPlayer {
    override fun play() {               // ...but translates call
        videoPlayer.playVideo()
    }
}

fun main() {
    val audio: MediaPlayer = AudioPlayer()
    audio.play()                        // ✅ works

    val video: MediaPlayer = VideoPlayerAdapter(VideoPlayer())
    video.play()                        // ✅ also works via adapter
}

```

intention behind lines

- `MediaPlayer` = the **shape** your app understands.
- `VideoPlayer` = **existing** thing with a different method (`playVideo()`).
- `VideoPlayerAdapter` = **wrapper** that **implements** `MediaPlayer` and forwards `play()` to `playVideo()` .
- `main()` uses only `MediaPlayer` → it doesn't care whether it's audio or video.

why this is good

- your app stays **decoupled** from concrete classes.
- you can plug in new players later with **new adapters**, not app rewrites.

---

even easier example: payments (unit conversion)

```kotlin
// App expects this:
interface Payment { fun pay(amountInRupees: Int) }

// Third-party library you can’t change:
class ThirdPartyGateway {
    fun makePayment(paise: Int) = println("Paid ₹${paise / 100.0}")
}

// Adapter converts your interface to theirs:
class GatewayAdapter(private val gateway: ThirdPartyGateway) : Payment {
    override fun pay(amountInRupees: Int) {
```

```
        gateway.makePayment(amountInRupees * 100) // convert rupees → paise
    }
}

fun main() {
    val pay: Payment = GatewayAdapter(ThirdPartyGateway())
    pay.pay(250) // Paid ₹250.0
}

```

---

when to use

- integrating a **third-party/legacy** class with a different API
- avoiding big `if/when` blocks and direct dependency on concrete classes

android notes

- **RecyclerView.Adapter** “adapts” your data to the rows the RecyclerView expects (not the classic GoF adapter exactly, but the spirit is the same: make two sides fit).
- Wrapping a callback-based SDK into a **clean interface** (or `suspend` functions) is a common adapter use.

pitfalls

- too many adapters can hint at the **wrong abstraction**—consider aligning interfaces at the domain boundary.
- keep adapters **thin** (translate names/types/units), not god-classes.

want me to turn either example into a tiny unit-tested snippet or a Compose demo button that switches adapters at runtime?

Real-World Use Case:

- Integrating **third-party libraries** into your app.

Facade

Facade = one simple doorway to a complicated building.
It gives you a tiny, clean method (or a few) that internally calls many subsystem classes in the right order.

Think **hotel reception**: you say “check me in”, and the receptionist coordinates rooms, keys, payments, notifications. You don’t talk to 5 different desks.

---

tiny Kotlin example — “Home Theater” 🎬

Subsystems (complicated but focused):

```
class Amplifier {
    fun on() = println("Amp ON")
    fun off() = println("Amp OFF")
    fun setVolume(level: Int) = println("Amp volume = $level")
}

class DvdPlayer {
    fun on() = println("DVD ON")
    fun off() = println("DVD OFF")
    fun play(movie: String) = println("Playing: $movie")
}

class Lights {
```

```
    fun dim(level: Int) = println("Lights dimmed to $level%")
}

```

Facade (simple API that orchestrates everything):

```
class HomeTheaterFacade(
    private val amp: Amplifier,
    private val dvd: DvdPlayer,
    private val lights: Lights
) {
    fun watchMovie(title: String) {
        println("Get ready for a movie…")
        lights.dim(30)
        amp.on(); amp.setVolume(5)
        dvd.on(); dvd.play(title)
    }

    fun endMovie() {
        println("Shutting movie theater down…")
        dvd.off()
        amp.off()
        lights.dim(100)
    }
}

fun main() {
    val theater = HomeTheaterFacade(Amplifier(), DvdPlayer(), Lights())
    theater.watchMovie("Interstellar")
    theater.endMovie()
}

```

what's happening (intention)

- `watchMovie()` is the **one-step** call your app uses.
- Inside, the Facade **coordinates** several steps across multiple objects in the right sequence.
- Clients don't need to know *how* many parts exist or *which order* to call them.

---

when to use

- You have a **complex subsystem** (many classes, specific order) but most callers just want a **simple action**.
- You're exposing a **clean entry point** (like a `Service` or `Repository`) to the rest of your app.

pitfalls

- Don't make your Facade a **god object**; keep it a thin coordinator.
- Keep business rules inside the **right** classes; the Facade should mainly **orchestrate**.

---

how it's different from Adapter / Mediator / Proxy

- **Adapter:** makes *incompatible interfaces fit* (translates A → B).
- **Facade:** provides a *simpler face* over *many* classes (A, B, C..., in order).

---

3. Behavioral Patterns

Behavioral patterns focus on communication between objects.

Observer Pattern

The **Observer pattern** is used when you want to notify multiple objects of changes in one object.

Example: News Update System

```
interface Observer {
    fun update(news: String)
}

class NewsSubscriber(private val name: String) : Observer {
    override fun update(news: String) {
        println("$name received news: $news")
    }
}
class NewsPublisher {
    private val subscribers = mutableListOf<Observer>()
    fun subscribe(observer: Observer) {
        subscribers.add(observer)
    }
    fun notifySubscribers(news: String) {
        subscribers.forEach { it.update(news) }
    }
}
fun main() {
    val newsPublisher = NewsPublisher()
    val subscriber1 = NewsSubscriber("Alice")
    val subscriber2 = NewsSubscriber("Bob")
    newsPublisher.subscribe(subscriber1)
    newsPublisher.subscribe(subscriber2)
    newsPublisher.notifySubscribers("Breaking News: Kotlin 2.0 Released!")
}
```

Explanation:

- The **publisher** notifies all its subscribers when new data is available.

Real-World Use Case:

- Live updates in **chat applications** or **news apps**.

Strategy Pattern

The **Strategy pattern** allows you to define multiple algorithms and choose one dynamically at runtime.

Example: Payment System

```
interface PaymentStrategy {
    fun pay(amount: Double)
}

class CreditCardPayment : PaymentStrategy {
    override fun pay(amount: Double) {
        println("Paid $$amount using Credit Card")
    }
}
class PayPalPayment : PaymentStrategy {
    override fun pay(amount: Double) {
        println("Paid $$amount using PayPal")
    }
}
class ShoppingCart(private val paymentStrategy: PaymentStrategy) {
    fun checkout(amount: Double) {
        paymentStrategy.pay(amount)
    }
}
fun main() {
    val cart = ShoppingCart(CreditCardPayment())
    cart.checkout(100.0)
```

```
    val anotherCart = ShoppingCart(PayPalPayment())
    anotherCart.checkout(50.0)
}
```

Explanation:

- You can **swap strategies** without altering the client code.

Real-World Use Case:

- Implementing dynamic **sorting algorithms** or **payment methods**.

Command — "wrap an action as an object"

Idea: Turn a button click or menu action into an object so you can queue, log, undo.

```
fun interface Command { fun execute() }

class Light { fun on() = println("on"); fun off() = println("off") }
class TurnOn(private val light: Light) : Command { override fun execute() = light.on() }

class Button(private val cmd: Command) { fun click() = cmd.execute() }

```

Intention:

- `Command` standardizes "do it".
- `Button` doesn't know the action; it just calls `execute()` → decoupled UI logic.

Android: toolbar actions, work manager tasks, undo stacks.

Let's break the **Command pattern** down in plain English, then walk through your code step-by-step, and finally show a few powerful upgrades (macro commands, redo stack, and a tiny lambda version).

Command wraps "*do this action*" into an **object**. Because it's an object, you can pass it around, queue it, log it, undo it, or run it later — without the caller knowing the details.

Think TV remote:

- Button (Invoker) doesn't know electronics.
- Each button has a tiny **command** object that knows **what to do** on the device (Receiver).

```
Client → Invoker (RemoteControl) → Command (TurnOnLight) → Receiver (Light)

```

Your code, explained line-by-line

1) Command interface — the contract

```
interface Command {
    fun execute()
    fun undo()
}

```

- Every command **must** know how to **do** (`execute`) and **undo** (`undo`) its action.

2) Receiver — the thing that actually works

```kotlin
class Light {
    fun turnOn()  = println("Light is ON")
    fun turnOff() = println("Light is OFF")
}

```

- `Light` has the **real operations**. Commands will call these.

3) Concrete commands — tiny wrappers around receiver actions

```kotlin
class TurnOnLightCommand(private val light: Light) : Command {
    override fun execute() = light.turnOn()
    override fun undo()    = light.turnOff()
}

class TurnOffLightCommand(private val light: Light) : Command {
    override fun execute() = light.turnOff()
    override fun undo()    = light.turnOn()
}

```

- Each command **holds** a `Light` (composition).
- `execute()` calls the forward action; `undo()` calls the **inverse** action.
- This is where the **knowledge** of “how to do it” lives.

4) Invoker — triggers commands & remembers history

```kotlin
class RemoteControl {
    private val commandHistory = mutableListOf<Command>()

    fun executeCommand(command: Command) {
        command.execute()
        commandHistory.add(command)     // keep for undo
    }

    fun undoLastCommand() {
        if (commandHistory.isNotEmpty()) {
            val last = commandHistory.removeLast()
            last.undo()
        } else {
            println("No commands to undo.")
        }
    }
}

```

- `RemoteControl` doesn’t know about `Light` at all.
- It just **runs** the command and **stores** it so it can call `undo()` later.

5) Main — how it flows

```kotlin
fun main() {
    val light = Light()
    val turnOn = TurnOnLightCommand(light)
    val turnOff = TurnOffLightCommand(light)
    val remote = RemoteControl()

    remote.executeCommand(turnOn)   // → Light is ON
    remote.executeCommand(turnOff)  // → Light is OFF
    remote.undoLastCommand()        // undo last (TurnOff) → Light is ON
}

```

Timeline:

1. `execute turnOn` → `Light.turnOn()` → “Light is ON”
2. `execute turnOff` → `Light.turnOff()` → “Light is OFF”
3. `undo last` (which was **turnOff**) → `Light.turnOn()` → “Light is ON”

That’s exactly your output.

Why this pattern is useful

- **Parameterize actions:** pass any command to the same invoker method.
- **Queue & schedule:** store commands to run later.
- **Logging & replay:** record commands, replay them to rebuild state.
- **Undo/redo:** keep stacks of commands to reverse/redo actions.
- **Decoupling:** UI code doesn’t know device details; new devices or actions don’t change the invoker.

Useful upgrades

1) Macro command (run many actions as one “scene”)

```
class MacroCommand(private val commands: List<Command>) : Command {
    override fun execute() = commands.forEach { it.execute() }
    override fun undo()    = commands.asReversed().forEach { it.undo() } // reverse order!
}

```

Use it:

```
val movieTime = MacroCommand(listOf(
    TurnOnLightCommand(light),
    // Add more: LowerBlindsCommand(blinds), StartProjectorCommand(projector), ...
))
remote.executeCommand(movieTime)
remote.undoLastCommand() // undoes the scene safely in reverse order

```

2) Add a redo stack (common in editors)

```
class RemoteControl {
    private val undoStack = ArrayDeque<Command>()
    private val redoStack = ArrayDeque<Command>()

    fun executeCommand(c: Command) {
        c.execute()
        undoStack.addLast(c)
        redoStack.clear()           // new action kills redo history (like editors)
    }

    fun undo() {
        val c = undoStack.removeLastOrNull() ?: return println("Nothing to undo")
        c.undo()
        redoStack.addLast(c)
    }

    fun redo() {
        val c = redoStack.removeLastOrNull() ?: return println("Nothing to redo")
        c.execute()
        undoStack.addLast(c)
    }
}

```

3) Kotlin-y lambda version (when you don't need classes)

For quick UIs, you can model a command as two functions:

```
data class SimpleCommand(val doIt: () -> Unit, val undoIt: () -> Unit)

class SimpleRemote {
    private val history = ArrayDeque<SimpleCommand>()
    fun run(cmd: SimpleCommand) { cmd.doIt(); history.addLast(cmd) }
    fun undo() { history.removeLastOrNull()?.undoIt() ?: println("Nothing to undo") }
}

val remote2 = SimpleRemote()
remote2.run(SimpleCommand(
    doIt   = { light.turnOn() },
    undoIt = { light.turnOff() }
))
remote2.undo()
```

> This is great for small apps. For large systems, the interface-based approach scales better (types, testing, composition).

4) Async commands with coroutines (Android-friendly)

```
interface SuspendCommand {
    suspend fun execute()
    suspend fun undo()
}
```

Now your invoker can `launch` or `await` commands without blocking the UI thread.

Android/Compose ideas

- Toolbar buttons as **commands** (e.g., Save, Share, Delete) → easy to disable/undo.
- Work queues: each task is a command → feed into `WorkManager`.
- Text editor: each edit is a command capturing **previous state** → robust undo/redo.
- "Scenes" (MacroCommand): e.g., Home automation routines or multi-step form operations.

Pitfalls & tips

- **Undo must be well-defined.** Some actions aren't trivially reversible (e.g., sending an email). Consider **compensating actions** (e.g., "send follow-up cancel").
- **Partial failure** in MacroCommand? Use `try/finally` and attempt to undo executed parts on error.
- **State capture:** Commands should capture enough state to undo reliably (e.g., old brightness level).
- **Don't overuse:** For simple one-offs, a direct function call is fine. Use Command when you need **queue/undo/log/decouple**.

Iterator (loop without knowing internals)

Idea: give a standard way to loop: `hasNext()` + `next()`, hiding how data is stored.

Tiny Kotlin example:

```kotlin
class Stepper(private val n: Int) : Iterable<Int> {
    override fun iterator() = object : Iterator<Int> {
        var cur = 0
        override fun hasNext() = cur < n
        override fun next() = cur++
    }
}

fun main() {
    for (i in Stepper(3)) println(i)   // prints: 0, 1, 2
}
```

You can `for` -loop `Stepper` without knowing it uses `cur` inside.

Template Method (fixed recipe, pluggable steps)

Idea: base class defines the **order of steps**; subclasses fill/override the pieces.

Tiny Kotlin example:

```kotlin
abstract class DrinkMaker {
    fun make() { boilWater(); brew(); pour() }   // fixed order
    private fun boilWater() = println("Boil water")
    private fun pour()      = println("Pour into cup")
    protected abstract fun brew()
}

class TeaMaker : DrinkMaker() {
    override fun brew() = println("Steep tea bag")
}

fun main() {
    TeaMaker().make()
    // Output:
    // Boil water
    // Steep tea bag
    // Pour into cup
}
```

`make()` is the **template**; `brew()` is the customizable step.

---

here's the super-short, super-simple version 👇

Proxy (same face, extra control)

Idea: A stand-in object with the **same interface** as the real one that adds control (lazy load, cache, auth) before calling the real thing.

Tiny Kotlin example (lazy image load):

```kotlin
interface Image { fun display() }

class RealImage(private val path: String) : Image {
    init { println("Loading file: $path") }   // expensive
    override fun display() = println("Showing $path")
}

class LazyImage(private val path: String) : Image {
    private var real: RealImage? = null
```

```
    override fun display() {
        if (real == null) real = RealImage(path)  // load only when needed
        real!!.display()
    }
}

fun main() {
    val img: Image = LazyImage("pic.png")
    println("Created proxy")   // no load yet
    img.display()              // loads + shows
}

```

Use when: objects are expensive/remote, or you need access control or caching.

Prototype (copy to create)

Idea: Make a new object by **cloning** an existing one, then tweak a few fields.

Tiny Kotlin example (data class `copy`)

```
data class ButtonStyle(val color: String, val radius: Int, val shadow: Boolean)

fun main() {
    val base = ButtonStyle(color = "#2196F3", radius = 8, shadow = true)
    val danger = base.copy(color = "#F44336")   // cloned with a change
    println(base)    // ButtonStyle(color=#2196F3, radius=8, shadow=true)
    println(danger)  // ButtonStyle(color=#F44336, radius=8, shadow=true)
}

```

Use when: you have presets/templates and need fast variations without rebuilding from scratch.