**Structural Design Patterns**

Structural patterns shape how classes and objects come together to build larger, efficient systems. They emphasize connections between components, often making designs cleaner by spotting typical relationship patterns.

**1. Adapter Pattern**

**Summary**: Adjusts a class’s interface to fit what a client needs, bridging gaps between incompatible parts.  
**Pros**:

* Lets old classes fit new systems without changes, aiding reuse.
* Frees the client from the adaptee’s details, boosting flexibility.
* Straightforward to adapt new mismatched pieces.  
  **Cons**:
* Adds adapter classes, raising complexity.
* Might slow things slightly with extra steps.
* Too many adapters can bog down maintenance.  
  **When It’s Handy**:
* Use it to blend legacy or external code with clashing interfaces into your work.
* Good for reusing classes with mismatched interfaces.
* Example: A voltage converter fitting a US device into an EU socket.

**2. Bridge Pattern**

**Summary**: Splits abstraction from implementation, letting each grow on its own by dividing their hierarchies.  
**Pros**:

* Keeps abstraction and implementation free to change separately, enhancing adaptability.
* Cuts ties between system parts.
* Works well for cross-platform setups (e.g., UI with different OS engines).  
  **Cons**:
* Layers make it trickier to follow.
* Can muddy purpose if the split isn’t obvious.
* Needs planning to avoid overcomplicating things.  
  **When It’s Handy**:
* Use when abstraction and implementation should evolve apart (e.g., a Car class with gas or electric engines).
* Fits systems expecting separate updates to both.
* Example: A music app with playback (abstraction) using different audio libraries (implementation).

**3. Decorator Pattern**

**Summary**: Adds features to objects at runtime by wrapping them in decorator layers, sidestepping heavy subclassing.  
**Pros**:

* Lets you tweak objects on the fly without touching their core.
* Avoids a mess of subclasses for every feature mix.
* Sticks to the Open/Closed Principle (extendable, not editable).  
  **Cons**:
* Piles on complexity with wrapper objects.
* Can make bugs harder to pin down across layers.
* Not great if object identity matters (e.g., for equality).  
  **When It’s Handy**:
* Perfect for runtime enhancements (e.g., adding a frame to a photo).
* Use when subclassing would spiral out of control.
* Example: C# streams with added compression via decorators.

**4. Facade Pattern**

**Summary**: Gives a clean, single interface to a messy subsystem, keeping clients clear of the chaos.  
**Pros**:

* Trims complexity with one access point.
* Makes APIs easier to grasp and use.
* Keeps clients separate from subsystem guts, easing updates.  
  **Cons**:
* Might hide too much, blocking advanced options.
* Could turn into a bloated hub if overpacked.
* Adds a step that might nudge performance.  
  **When It’s Handy**:
* Use to tame wild subsystems (e.g., a sprawling toolkit).
* Good for cutting client-subsystem links.
* Example: A smart home hub controlling lights, thermostat, and locks.

**5. Proxy Pattern**

**Summary**: Serves as a substitute for an object to manage access, add extras, or hold off creation.  
**Pros**:

* Controls access (e.g., delays or security) without changing the real thing.
* Can speed things up (e.g., lazy loading big files).
* Often invisible to clients.  
  **Cons**:
* Extra steps might drag performance.
* Adds classes, upping complexity.
* Poor design could repeat logic.  
  **When It’s Handy**:
* Use to delay costly object creation (e.g., loading a map only when viewed – Virtual Proxy).
* Great for adding security or tracking (e.g., network proxies).
* Example: A login proxy restricting database access.

**Quick Reference**

| **Pattern** | **Pros** | **Cons** | **When to Apply** |
| --- | --- | --- | --- |
| Adapter | Reuse, versatility | Layers, clutter | Fix interface clashes |
| Bridge | Separate growth, loose ties | Complexity, intent blur | Split abstraction-implementation |
| Decorator | Runtime tweaks, subclass avoidance | Wrappers, debug woes | Flexible enhancements |
| Facade | Clarity, separation | Over-simplification, bloat | Subsystem cleanup |
| Proxy | Control, efficiency | Steps, complexity | Manage object use |

**Behavioral Design Patterns**

Behavioral patterns guide how objects talk and split duties, shaping their teamwork.

**1. Chain of Responsibility**

**Summary**: Sends a request down a line of handlers, each choosing to act or pass it on.  
**Pros**:

* Frees the sender from picking a receiver, adding wiggle room.
* Lets you swap handlers in or out easily.
* Splits jobs cleanly among handlers.  
  **Cons**:
* Requests might fall through if the line’s off.
* Long lines can bog down speed.
* Tracking the flow can get messy.  
  **When It’s Handy**:
* Use when various objects might tackle a task (e.g., app event routing).
* Good for ordered request handling.
* Example: A help desk escalating tickets from basic to expert support.

**2. Observer**

**Summary**: Sets up a one-to-many link where a subject’s changes ping all watchers.  
**Pros**:

* Keeps subject and watchers loosely linked.
* Handles watcher changes on the fly.
* Suits event-driven setups.  
  **Cons**:
* Watchers left hanging can leak memory.
* Lots of watchers slow notifications.
* Updates might snowball if unchecked.  
  **When It’s Handy**:
* Use when state shifts need to alert others (e.g., app UI updates).
* Fits subscription-style systems.
* Example: A news feed pushing updates to readers.

**3. Visitor**

**Summary**: Pulls operations off an object setup, letting you add new tasks without tweaking it.  
**Pros**:

* Adds tasks easily (Open/Closed Principle).
* Keeps related logic in one visitor spot.
* Works with tricky structures.  
  **Cons**:
* Visitor classes pile on complexity.
* Might peek at private data, risking encapsulation.
* Tough if the setup keeps shifting.  
  **When It’s Handy**:
* Use for tasks on steady structures (e.g., code analysis trees).
* Good when operations evolve more than the base.
* Example: A stats tool tallying metrics across file types.

**Creational Design Patterns**

Creational patterns handle object birth, aiming for flexibility and smart resource use.

**1. Abstract Factory**

**Summary**: Offers a way to craft related object sets without pinning down their exact types.  
**Pros**:

* Keeps object groups in sync (e.g., app skins).
* Hides concrete classes from clients.
* Easy to switch sets.  
  **Cons**:
* Factory layers add heft.
* Too much for simple needs.
* New types mean big updates.  
  **When It’s Handy**:
* Use for crafting matched object batches (e.g., game assets for PC vs. console).
* Fits when creation specifics should stay under wraps.
* Example: A gadget maker producing paired phones and chargers.

**2. Builder**

**Summary**: Splits complex object assembly from its form, letting one process yield different results.  
**Pros**:

* Breaks down tricky builds into steps.
* Gives tight control over the process.
* Reuses steps for varied outputs.  
  **Cons**:
* Builder classes bulk up the design.
* Needs a builder per output type.
* Overkill for basic stuff.  
  **When It’s Handy**:
* Use for objects with lots of pieces (e.g., a custom bike).
* Good for isolating build logic.
* Example: Crafting a salad with pick-your-own ingredients.

**3. Factory Method**

**Summary**: Provides a creation hook, letting subclasses choose the object type.  
**Pros**:

* Keeps clients clear of specific classes.
* Simplifies new type additions via subclasses.
* Standardizes creation.  
  **Cons**:
* Can flood with subclasses.
* Heavier than straight instantiation.
* Struggles with wild variety.  
  **When It’s Handy**:
* Use when object types aren’t set (e.g., a chat app’s message styles).
* Fits subclass-led creation.
* Example: A tool spawning image or text editors.

**4. Prototype**

**Summary**: Makes new objects by copying a sample, dodging class-based creation.  
**Pros**:

* Skips subclassing with copies.
* Quickens complex object births.
* Flexible at runtime.  
  **Cons**:
* Copying gets messy with linked objects.
* Deep copies cost more.
* Weak for light setups.  
  **When It’s Handy**:
* Use when copying trumps building (e.g., game props).
* Good for tweaking samples.
* Example: Cloning a base email template.

**5. Singleton**

**Summary**: Caps a class at one instance, offering universal access.  
**Pros**:

* Locks in one instance for shared use.
* Easy global reach.
* Lazy setup saves resources.  
  **Cons**:
* Global scope ties things tight.
* Static nature hampers testing.
* Needs thread care.  
  **When It’s Handy**:
* Use for one-off needs (e.g., a user session).
* Fits single control hubs.
* Example: A sound manager with one output.

**Quick Reference**  
*Behavioral Patterns*

| **Pattern** | **Pros** | **Cons** | **When to Apply** |
| --- | --- | --- | --- |
| Chain of Resp. | Versatility, job split | Drops, slowdowns | Multi-step requests |
| Observer | Loose links, event fit | Leaks, notify costs | Change alerts |
| Visitor | Task ease, logic grouping | Complexity, data risks | Fixed structure tasks |

*Creational Patterns*

| **Pattern** | **Pros** | **Cons** | **When to Apply** |
| --- | --- | --- | --- |
| Abstract Factory | Sync, hiding | Heft, rigidity | Object family creation |
| Builder | Stepwise, control | Extra heft, niche | Complex assemblies |
| Factory Method | Isolation, growth | Subclass sprawl | Subclass picks |
| Prototype | Quick, flexible | Copy woes | Clone-based birth |
| Singleton | One-off, access | Ties, test issues | Sole instance needs |