#### Structural Design Patterns

Structural design patterns deal with organizing classes and objects into larger, cohesive systems while ensuring adaptability and efficiency. They focus on relationships among entities, streamlining designs by recognizing recurring ways to connect components.

##### 1. Adapter Pattern

**Overview**: Transforms a class’s interface to align with what a client requires, enabling compatibility between otherwise mismatched components.  
**Advantages**:

* Boosts reuse by integrating existing classes into new contexts without altering them.
* Enhances adaptability by separating the client from the adaptee’s specifics.
* Simplifies expansion to accommodate new incompatible elements.  
  **Disadvantages**:
* Introduces extra complexity with adapter classes.
* May slightly slow performance due to added layers.
* Excessive use could clutter the system with too many adapters.  
  **Use Case**:
* Ideal for merging legacy code or external libraries with incompatible interfaces into your project.
* Useful when reusing a class with an interface that doesn’t fit your needs.
* Example: Converting a UK plug for use in a Japanese outlet with an adapter.

##### 2. Bridge Pattern

**Overview**: Separates an abstraction from its implementation, allowing both to evolve independently by splitting their hierarchies.  
**Advantages**:

* Permits separate development of abstraction and implementation, increasing versatility.
* Lowers dependency between parts of the system.
* Supports cross-platform designs (e.g., UI tools with varying OS support).  
  **Disadvantages**:
* Adds layers that can complicate the design.
* Might confuse intent if the separation lacks clear purpose.
* Demands foresight to prevent unnecessary complexity.  
  **Use Case**:
* Best when abstraction and implementation need distinct evolution (e.g., a Shape class with rendering options like Vulkan or Metal).
* Suited for systems expecting independent updates to both aspects.
* Example: A graphics tool where shapes are drawn using different rendering backends.

##### 3. Decorator Pattern

**Overview**: Dynamically adds features to an object by enclosing it in decorator classes, offering a flexible substitute for inheritance.  
**Advantages**:

* Provides runtime customization without altering the core class.
* Prevents subclass overload by avoiding combinations for every feature.
* Adheres to the Open/Closed Principle (extensible, not modifiable).  
  **Disadvantages**:
* Creates complexity with numerous wrapper objects.
* Can complicate tracing issues across decorator layers.
* Unsuitable if object identity must stay constant (e.g., for comparisons).  
  **Use Case**:
* Perfect for adding capabilities dynamically (e.g., enhancing a text box with borders or scrolling).
* Useful when inheritance would lead to excessive subclassing.
* Example: Python’s file streams with added buffering via decorators.

##### 4. Facade Pattern

**Overview**: Offers a straightforward interface to a complicated subsystem, shielding clients from its details.  
**Advantages**:

* Cuts down complexity with a unified access point.
* Boosts API clarity and ease of use.
* Isolates clients from subsystem internals, aiding upkeep.  
  **Disadvantages**:
* Might overly simplify, restricting access to nuanced features.
* Risks becoming a monolithic hub if overloaded with duties.
* Adds a layer that could marginally affect speed.  
  **Use Case**:
* Great for easing interaction with intricate systems (e.g., a software library).
* Helpful when minimizing ties between clients and subsystem parts.
* Example: A car dashboard simplifying control of engine, radio, and AC.

##### 5. Proxy Pattern

**Overview**: Acts as a stand-in for another object to manage access, enhance functionality, or delay creation.  
**Advantages**:

* Grants control over access (e.g., security or lazy loading) without altering the target.
* Can optimize performance (e.g., deferring costly object setup).
* Often seamless for clients.  
  **Disadvantages**:
* Adds indirection that might slow execution.
* Increases design intricacy with extra classes.
* Risks redundant logic if not well-planned.  
  **Use Case**:
* Ideal for postponing heavy| heavy resource creation (e.g., loading a video only when viewed – Virtual Proxy).
* Useful for adding security or logging layers (e.g., network proxies).
* Example: A caching proxy for a web service.

**Summary Table**

| **Pattern** | **Advantages** | **Disadvantages** | **Use Case** |
| --- | --- | --- | --- |
| Adapter | Reuse, adaptability | Extra layers, clutter | Interface mismatches |
| Bridge | Independent growth, low coupling | Complexity, planning needed | Abstraction split |
| Decorator | Runtime flexibility, no subclass explosion | Wrappers, debug issues | Dynamic enhancements |
| Facade | Ease, isolation | Simplification risks, overhead | Subsystem simplification |
| Proxy | Control, optimization | Indirection, complexity | Access management |

#### Behavioral Design Patterns

Behavioral patterns address how objects collaborate and share responsibilities, shaping their interactions.

##### 1. Chain of Responsibility

**Overview**: Routes a request through a sequence of handlers, each choosing to handle it or pass it along.  
**Advantages**:

* Separates requester from handler, boosting adaptability.
* Supports adding or dropping handlers dynamically.
* Encourages task-specific roles.  
  **Disadvantages**:
* Requests might slip through unprocessed if misconfigured.
* Long chains can slow performance.
* Chain dynamics may hinder debugging.  
  **Use Case**:
* Suited for scenarios where multiple entities could process a request (e.g., UI event systems).
* Best when requests need ordered handling.
* Example: An email filter passing messages through spam, priority, and archive checks.

##### 2. Observer

**Overview**: Establishes a one-to-many link where a subject’s state changes trigger updates to all observers.  
**Advantages**:

* Keeps subject and observers loosely tied.
* Allows observer addition/removal at runtime.
* Fits event-based designs.  
  **Disadvantages**:
* Risks memory issues if observers linger.
* High observer counts increase update costs.
* Unchecked updates might cascade.  
  **Use Case**:
* Ideal when state shifts affect multiple parties (e.g., GUI refreshes).
* Perfect for subscription-based systems.
* Example: A weather app alerting users to temperature drops.

##### 3. Visitor

**Overview**: Detaches operations from an object structure, enabling new functions without altering it.  
**Advantages**:

* Simplifies adding operations (Open/Closed Principle).
* Groups related logic in a visitor class.
* Handles complex structures well.  
  **Disadvantages**:
* Adds visitor-related complexity.
* May expose private data, weakening encapsulation.
* Tough to adapt if the structure shifts often.  
  **Use Case**:
* Best for applying operations to fixed structures (e.g., parsing code trees).
* Useful when operations change more than the structure.
* Example: A payroll system calculating bonuses across employee types.

#### Creational Design Patterns

Creational patterns streamline object instantiation, making it adaptable and resource-efficient.

##### 1. Abstract Factory

**Overview**: Supplies an interface to produce sets of related objects without tying to specific classes.  
**Advantages**:

* Guarantees cohesive object sets (e.g., themed UI controls).
* Shields clients from concrete implementations.
* Simplifies switching product families.  
  **Disadvantages**:
* Layers on complexity with factory classes.
* Overkill for basic creation needs.
* Hard to expand for new object types.  
  **Use Case**:
* Great for generating related object groups (e.g., app widgets for iOS vs. Android).
* Fits when creation details should stay hidden.
* Example: A toy factory making matching dolls and accessories.

##### 2. Builder

**Overview**: Breaks down complex object creation into steps, allowing varied outputs from one process.  
**Advantages**:

* Eases building intricate objects incrementally.
* Offers precise construction control.
* Reuses a single process for multiple results.  
  **Disadvantages**:
* Adds builder-related complexity.
* Needs distinct builders per product.
* Less useful for simple objects.  
  **Use Case**:
* Ideal for objects with many options (e.g., a custom PC).
* Best when separating creation from the object’s core.
* Example: Assembling a burger with selectable toppings.

##### 3. Factory Method

**Overview**: Sets a creation interface, letting subclasses pick the instantiated class.  
**Advantages**:

* Isolates clients from concrete types.
* Eases adding new types via subclasses.
* Ensures uniform creation.  
  **Disadvantages**:
* May spawn many subclasses.
* More complex than direct creation.
* Struggles with diverse products.  
  **Use Case**:
* Suited when object types aren’t predictable (e.g., a report generator).
* Useful for subclass-driven creation.
* Example: A game spawning enemy types via subclasses.

##### 4. Prototype

**Overview**: Generates objects by duplicating a prototype, bypassing traditional class instantiation.  
**Advantages**:

* Cuts subclassing by cloning.
* Speeds up complex object setup.
* Offers runtime creation flexibility.  
  **Disadvantages**:
* Cloning gets tricky with nested references.
* Deep copies add overhead.
* Less effective for simple setups.  
  **Use Case**:
* Best when cloning beats creating anew (e.g., game NPCs).
* Fits when tweaking existing objects.
* Example: Copying a base character model for variations.

##### 5. Singleton

**Overview**: Limits a class to one instance, providing universal access to it.  
**Advantages**:

* Ensures one instance for shared use.
* Streamlines access globally.
* Delays creation for efficiency.  
  **Disadvantages**:
* Global state tightens coupling.
* Testing is harder with statics.
* Needs thread-safe design.  
  **Use Case**:
* Perfect for single-instance needs (e.g., a settings manager).
* Suited for centralized control points.
* Example: A printer queue with one manager.

**Summary Tables**  
Behavioral Patterns

| **Pattern** | **Advantages** | **Disadvantages** | **Use Case** |
| --- | --- | --- | --- |
| Chain of Resp. | Adaptability, task split | Unhandled cases, slowdowns | Multi-handler requests |
| Observer | Loose ties, event support | Leaks, update costs | State notifications |
| Visitor | Easy operations, focus | Complexity, exposure | Stable structure ops |

Creational Patterns

| **Pattern** | **Advantages** | **Disadvantages** | **Use Case** |
| --- | --- | --- | --- |
| Abstract Factory | Cohesion, isolation | Complexity, rigidity | Related object sets |
| Builder | Incremental, versatile | Extra classes, niche | Complex builds |
| Factory Method | Isolation, expansion | Subclass bloat | Subclass creation |
| Prototype | Speed, adaptability | Cloning issues | Copy-based creation |
| Singleton | Unity, ease | Coupling, testing | Single instance control |