Strategy Design Pattern

Defining a family of algorithms, encapsulating them, and making them interchangeable without altering client code. It's particularly useful in scenarios where you need to switch between multiple algorithms dynamically or add new ones without changing client code.

Key Components:

- **Strategy**: Declares an interface or abstract class for a family of algorithms.
- **Concrete Strategies**: Implement the strategy interface with specific algorithm variations.
- **Context**: Maintains a reference to a strategy and delegates tasks to it.

Advantages:

- **Flexibility**: Allows dynamic selection of algorithms at runtime.
- **Decoupling**: Separates algorithm details from the client, promoting low coupling.
- **Extensibility**: Easily add new algorithms without modifying existing code.
- **Testability**: Simplifies unit testing by replacing strategies with mocks or stubs.

Disadvantages:

- **Complexity**: May introduce additional classes and indirection.
- **Client Responsibility**: Clients need to choose and set the appropriate strategy.
- **Potential Overhead**: Dynamic strategy selection can have some performance overhead.

Use Cases:

- **Sorting Algorithms**: Dynamic selection of sorting algorithms (e.g., quicksort, mergesort).
- **Data Compression**: Compressing data using various algorithms (e.g., gzip, zlib).
- **Routing Algorithms**: Selecting optimal routes for navigation.
- **Text Editors**: Implementing spell checking and autocorrect with different algorithms.
- **Gaming**: AI decision-making strategies for game characters.
- **Image Processing**: Applying filters or transformations using various algorithms.

Template Design Pattern

Defining the skeleton of an algorithm, allowing certain steps to be implemented by subclasses while maintaining the overall structure.

Key Components:

- **Abstract Template**: Declares the structure of the algorithm with placeholder methods.
- **Concrete Templates**: Implement specific versions of the algorithm by filling in the placeholder methods.

Advantages:

- **Code Reuse**: Promotes reuse of common algorithm structure among multiple subclasses.
- **Consistency**: Enforces a consistent algorithm structure across subclasses.
- **Extensibility**: Allows for the customization of specific algorithm steps.
- **Reduced Duplication**: Minimizes duplicated code in similar algorithms.

Disadvantages:

- **Complexity**: Can introduce complexity when there are many optional steps or hook methods.

Use Cases:

- **Report Generation**: Creating report templates with variable sections or data sources.
- **Web Page Generation**: Defining a common structure for web page rendering with customizable content.
- **Database Access**: Executing database queries with consistent connection management.
- **Test Automation**: Implementing test scripts with standardized setup and teardown steps.