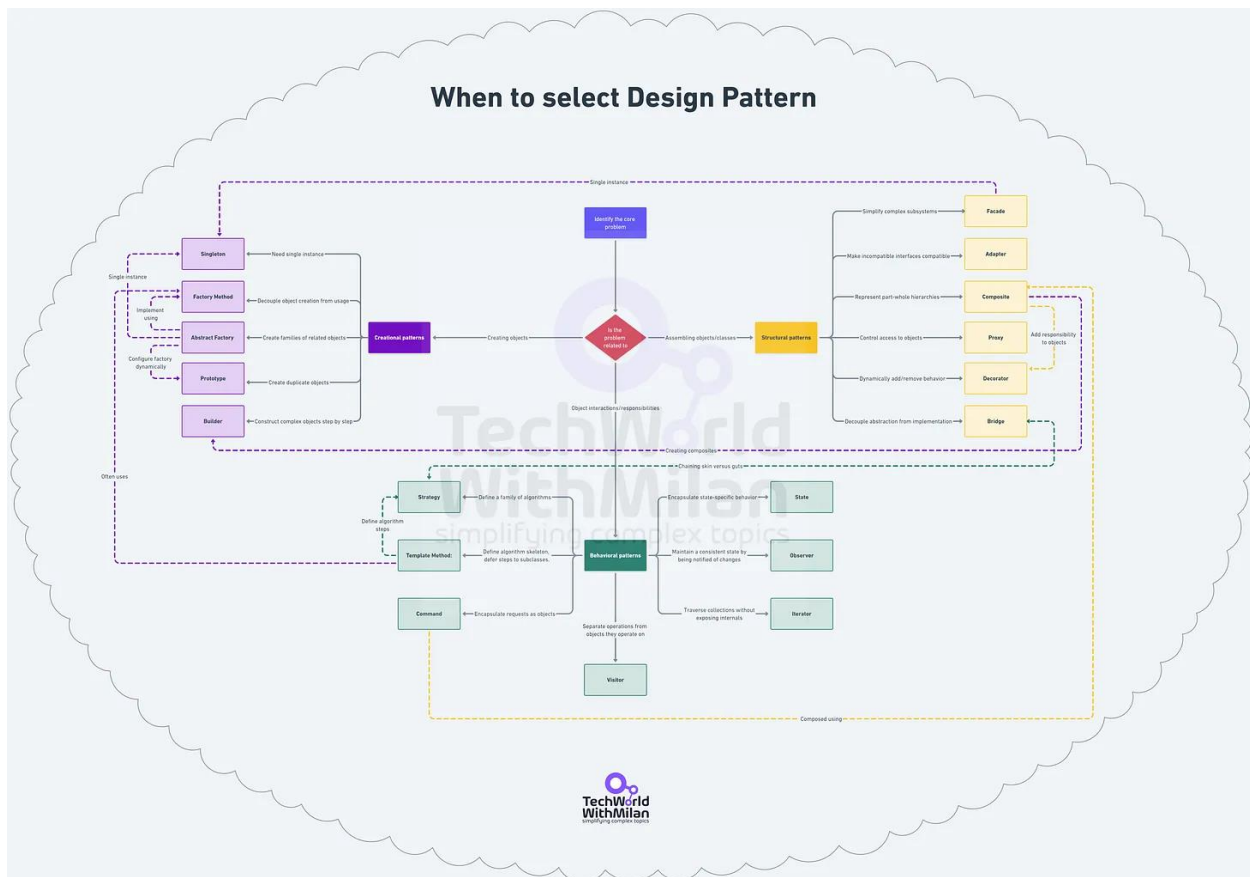


How to select the correct design pattern

Now, when we have analyzed all vital design patterns, we came up with the pattern we needed for our problem:



Here is the summary of the pattern selection:

HOW TO SELECT DESIGN PATTERN		
1. Creational Patterns	→ object creation	<ul style="list-style-type: none"> • Singleton: Use when a single instance of a class is needed. Some examples are logging and database connections. • Factory Method: Decouple object creation from usage. For example, you create different types of database connections based on configuration. • Abstract Factory: Create families of related objects. For example, I build parsers for different file formats (e.g., JSON, XML, CSV). • Builder: Constructing complex objects step by step. For example, if you need to create a complex domain object. • Prototype: Creating duplicate objects and reusing cached objects to reduce database calls.
2. Structural Patterns	→ object assembly	<ul style="list-style-type: none"> • Adapter: Make incompatible interfaces compatible. For example, it integrates a new logging library into an existing system that expects a different interface. • Composite: Represent part-whole hierarchies. For example, graphic objects in a drawing application can be grouped and treated uniformly. • Proxy: Control access to objects. For example, lazy loading of a high-resolution image in a web application. • Decorator: Dynamically add/remove behavior. For example, we are implementing compression or encryption on top of file streams. • Bridge: Decouple abstraction from implementation. For example, I am separating platform-specific code from core logic.
3. Behavioral Patterns	→ object interactions	<ul style="list-style-type: none"> • Strategy: Define a family of algorithms. For example, they allow users to choose different sorting or compression algorithms. • Observer: Maintain a consistent state by being notified of changes and, for example, notifying subscribers of events in a messaging system. • Command: Encapsulate a request as an object. For example, I implement undo/redo functionality in text or image editor. • State: Encapsulate state-specific behavior. For example, we are handling different states of a user interface element (e.g., enabled, disabled, selected). • Template Method: Define the skeleton of an algorithm in operation, deferring some steps to subclasses and implementing a base class for unit testing with customizable setup and teardown steps.

Design patterns are not the only kind of patterns we have. To learn more about other types of patterns, [check here](#). ([architecture-styles-patterns-and-design-patterns.pdf](#)/ [Top 10 Architectural Patterns.pdf](#))

BONUS: Design Patterns Cheat Sheet

Most Used Design Patterns Cheat Sheet

Creational Patterns

Used to construct objects

Structural Patterns

Used to form large object structures

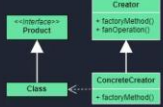
Behavioral Patterns

Used to manage algorithms and relationships

Factory Method

Use when you want to delegate object creation to subclasses.

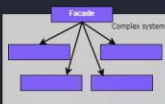
Example: create GUI component



Facade

Use when you want to provide a simplified interface to a complex subsystem

Example: Providing a simple interface to a complex subsystem



Proxy

Use for object access control

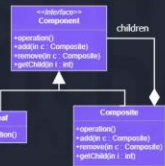
Example: Controlling access to sensitive resources



Composite

Represent part-whole hierarchies

Example: Graphic object in a drawing can be grouped



Template Method

Use when you want to break down an algorithm into a series of steps

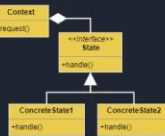
Example: Common behavior should be located in one class



State

Encapsulate state-specific behavior

Example: Handling different states of a user interface



Singleton

Use when you want to have one instance of a class.

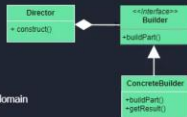
Example: logging, db connections.



Builder

Constructing complex objects, step by step

Example: create complex domain object



Adapter

Use when you need to convert an interface to another interface

Example: make incompatible classes work together



Decorator

Use when you need to wrap objects to modify their behaviors

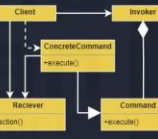
Example: make object behaviors dynamically modifiable



Command

Use for encapsulating requests with parameters

Example: Implementing operations



Strategy

Use for interchangeable algorithms that can be swapped at runtime

Example: Implement different sorting algorithms



Observer

Use for automatic updates of dependant objects

Example: Implement subscribers

