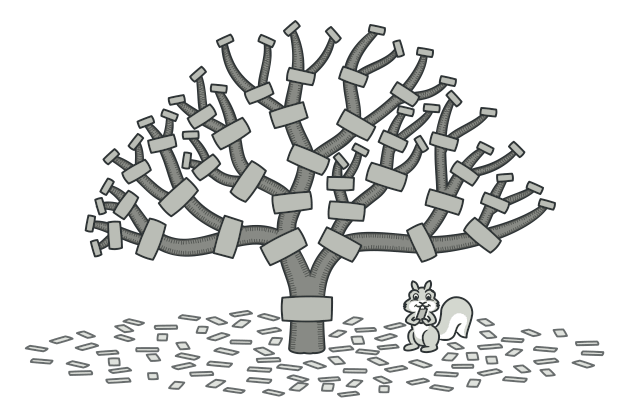
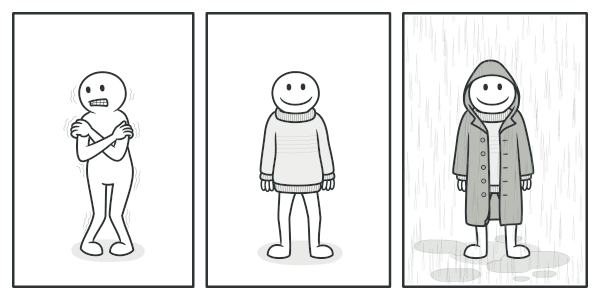
**Composite Patter:**



Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

**When to use:**  Representing a UI component tree (e.g., DOM, React component tree).

**Decorator Pattern:**

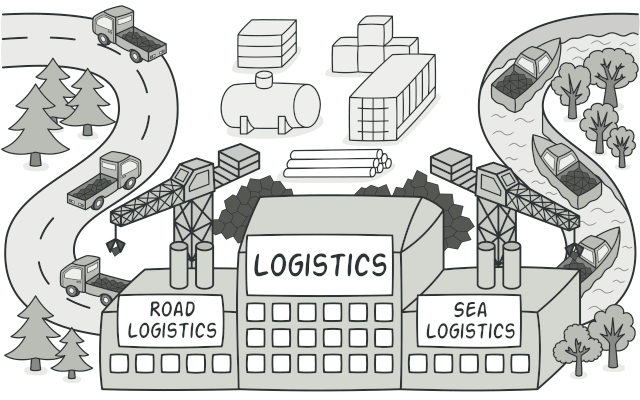


lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

**When to use:**

* Adding additional functionalities to UI components (e.g., adding logging, styling, validation).
* Enhancing React components using Higher-Order Components (HOCs) or custom hooks.
* Middleware functions in frameworks like Express.

**Factory Pattern**:

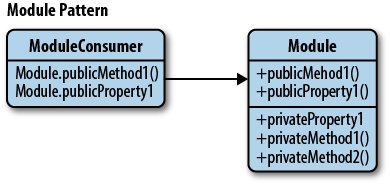


The Factory Pattern provides an interface for creating objects, but allows subclasses to alter the type of objects that will be created.

**When to use:**

* Creating different components dynamically based on input.
* Generating various types of charts, buttons, or UI elements.
* Abstracting the creation of objects with shared interface.

**Module Pattern**:

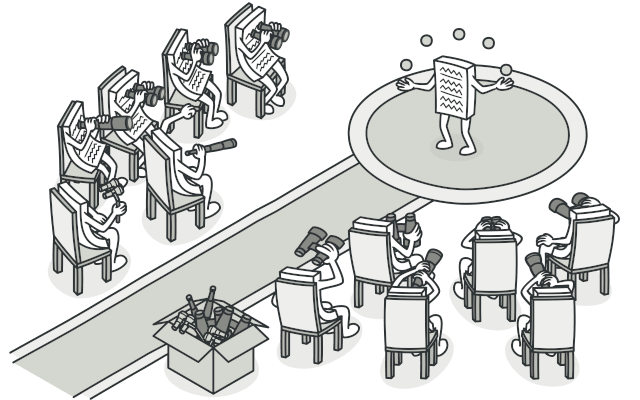


Encapsulate related functionalities into a single object, limiting the scope of variables and methods to avoid polluting the global namespace. It's a way of achieving privacy  and organization, similar to namespaces.

**When to use:**

* Encapsulating utility functions (e.g., date formatting, validation helpers).
* Creating services or APIs in frontend apps (e.g., authentication, data fetching modules).
* Managing application state in vanilla JS apps.
* Organizing configuration and constants.
* Wrapping third-party libraries to provide a consistent interface.

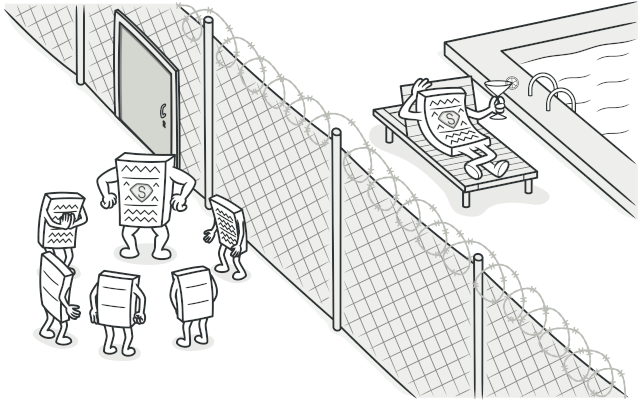
**Observer Pattern**:



Defines a one-to-many relationship between objects   so that when one object changes state, all its dependents are notified and updated automatically.

* Implementing event systems (like the EventEmitter).
* State management libraries (e.g., Redux, Zustand) to reactively update UI components.
* Real-time data updates (e.g., chat applications, notifications).

**The Proxy Pattern**:



* Intercepting requests/responses (e.g., logging, caching, or modifying network requests).
* Implementing lazy loading of images or components.
* Protecting access to sensitive parts of an application (like APIs).
* Virtual scrolling or infinite scroll lists (loading data as needed).
* Data validation and sanitization before updating state or sending to server.
* Creating mock APIs for frontend development/testing.

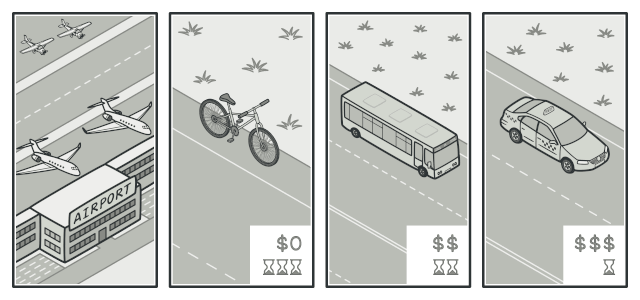
**The Singleton Pattern**:



Ensures a class only has one instance and provides a global point of access to it.

* Managing global application state (like a centralized store, e.g., Redux store).
* Implementing a service for API calls or configuration settings.
* Handling browser storage (localStorage, sessionStorage) through a single access point.
* Managing a single WebSocket connection for real-time features.
* Creating a global event bus for component communication.

**The Strategy Pattern**:



Defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern lets the algorithm vary independently from clients that use it.

* Implementing different sorting or filtering algorithms for data.
* Handling different payment methods or authentication strategies.
* Switching between different ways of rendering components.
* To avoid complex conditional logic (like many if-else or switch statements).
* To follow the Open/Closed Principle (open for extension, closed for modification).