Creational Design Patterns

Creational patterns abstract the process of object creation, improving flexibility and testability. Below, each pattern includes a conceptual overview and how this repository implements it with class and function names.

Builder

Description

Separates the construction of a complex object from its representation so that the same construction process can create different representations.

Usage

Use when you need to construct complex objects step-by-step, or when construction must allow different configurations.

Advantages

- Same construction, different representations
- Readable step-wise assembly
- Encapsulates construction logic in builders

Disadvantages

- More classes (builders, director)
- May be overkill for simple objects

Key Components

- Director – orchestrates the build steps
- Builder – interface for constructing parts
- ConcreteBuilder – actual build logic
- Product – the complex object

Implementation in this project

Classes: Car, ICarBuilder, CarBuilder, CarDirector.

Key methods in CarBuilder: SetEngine, SetTyres, SetColor, Build.

Director method: ConstructSportsCar.

Flow: CarDirector.ConstructSportsCar() → CarBuilder.SetEngine() → SetTyres() → SetColor() → Build() → returns Car.

Factory

Description

Defines an interface for creating objects, letting subclasses decide which class to instantiate.

Usage

Use when object creation logic varies or depends on input, and you want to avoid direct 'new' usage in client code.

Advantages

- Encapsulates creation logic

- Reduces coupling to concrete classes
- Centralizes object selection

Disadvantages

- Adds extra factory class
- Can become a large conditional if not evolved to Abstract Factory

Key Components

- Creator (Factory) – exposes creation method
- Product interface – contract for created objects
- Concrete Products – implementations

Implementation in this project

Classes: IAnimal, Dog, Cat, AnimalFactory.

Product operations: IAnimal.Speak(); concrete: Dog.Speak(), Cat.Speak().

Flow: AnimalFactory.CreateAnimal("Dog") → returns Dog → client calls Speak().

Fluent Builder

Description

A builder that returns itself from each step to enable readable method chaining.

Usage

Use for constructing objects with many optional parameters and when readability is important.

Advantages

- Fluent, readable chained calls
- Keeps object immutable until Build()
- Clear separation of configuration vs. creation

Disadvantages

- Can hide mandatory steps if not validated
- Long chains may obscure control flow

Key Components

- FluentBuilder – builder with chainable setters
- Product – the built object

Implementation in this project

Classes: CarVehicle, CarVehicleBuilder.

Typical call chain: CarVehicleBuilder.SetEngine(...).SetColor(...).SetTyres(...).Build().

Responsibility split: chain collects configuration; Build() constructs the final CarVehicle.

Memento

Description

Captures and externalizes an object’s internal state so the object can be restored to that state later, without exposing its internals.

Usage

Use when you need undo/redo or checkpoints in object state.

Advantages

- Undo/redo support without exposing internals
- Separation of concerns: Originator vs. Caretaker

Disadvantages

- Potential memory cost for many snapshots
- Careful lifecycle management of mementos

Key Components

- Originator – the object with state (Editor)
- Memento – an immutable snapshot (EditorMemento)
- Caretaker – stores mementos (History)

Implementation in this project

Editor methods: Save, Restore.

History methods: SaveState, RestoreState.

Flow: Editor.Save() → EditorMemento created → History.SaveState(memento).

Restore: var m = History.RestoreState(); Editor.Restore(m) → state rolled back.

Structural Design Patterns

Structural patterns explain how to assemble classes and objects into larger structures while keeping them flexible, testable, and efficient.

Bridge

Description

Decouples an abstraction from its implementation so that the two can vary independently.

Usage

Use when you want to avoid a permanent binding between an abstraction and its implementation.

Advantages

- Improved extensibility
- Greater flexibility
- Clear separation of concerns

Disadvantages

- Adds indirection and complexity
- Takes time to understand roles

Key Components

- Abstraction – high-level control (RemoteControl)
- Refined Abstraction – variants (StandardRemote)
- Implementor – device interface (ITV)
- Concrete Implementors – device implementations (SamsungTV, SonyTV)

Implementation in this project

Classes: ITV, RemoteControl, StandardRemote, SamsungTV, SonyTV.

RemoteControl methods: —.

StandardRemote methods: —.

Flow: RemoteControl delegates calls like SetChannel(...) to ITV; SamsungTV/SonyTV provide brand-specific behavior.

Composite

Description

Composes objects into tree structures to represent part–whole hierarchies, letting clients treat individual objects and compositions uniformly.

Usage

Use to model hierarchical structures (files/folders) with uniform operations.

Advantages

- Uniform treatment of leaf and composite
- Simplifies client logic

- Supports recursive operations

Disadvantages

- Can make type constraints looser
- May complicate object ownership/lifecycles

Key Components

- Component – common interface (FileSystemItem)
- Leaf – indivisible object (File)
- Composite – container of components (Directory)

Implementation in this project

Classes: FileSystemItem, File, Directory.

Directory methods: Add.

File methods: —.

Flow: Directory.Add(FileSystemItem) builds the tree; client calls Display()/GetSize() uniformly on both.

Decorator

Description

Allows behavior to be added to individual objects dynamically without affecting other objects of the same class.

Usage

Use to extend object responsibilities at runtime instead of subclassing.

Advantages

- Greater flexibility than inheritance
- Supports combining features dynamically
- Open/Closed-friendly

Disadvantages

- Many small classes can increase complexity
- Debugging wrapped chains can be harder

Key Components

- Component – interface (Coffee)
- Concrete Component – base (SimpleCoffee)
- Decorator – base wrapper (CoffeeDecorator)
- Concrete Decorators – features (MilkDecorator, SugarDecorator)

Implementation in this project

Classes: Coffee, SimpleCoffee, CoffeeDecorator, MilkDecorator, SugarDecorator.

Coffee methods: —.

Decorators’ methods (Milk/Sugar): MilkDecorator: —; SugarDecorator: —.

Flow: new MilkDecorator(new SugarDecorator(new SimpleCoffee())).GetCost()/GetDescription() adds features layer by layer.

Facade

Description

Provides a simplified interface to a complex subsystem.

Usage

Use to reduce coupling and simplify usage of multiple collaborating classes.

Advantages

- Simpler API for clients
- Hides subsystem complexity
- Reduces dependencies

Disadvantages

- Risk of becoming a God Object
- May hide useful subsystem features

Key Components

- Facade – unified entry (HomeTheaterFacade)
- Subsystem classes – DvdPlayer, Projector, SoundSystem

Implementation in this project

Classes: HomeTheaterFacade, DvdPlayer, Projector, SoundSystem.
HomeTheaterFacade methods: WatchMovie, EndMovie.
Flow: WatchMovie() orchestrates DvdPlayer.On(), Projector.On(), SoundSystem.On() and related operations.

Flyweight

Description

Minimizes memory usage by sharing as much data as possible with similar objects.

Usage

Use when you have many similar objects and memory is a concern.

Advantages

- Lower memory footprint
- Potential performance gains via sharing

Disadvantages

- More complex state management (intrinsic vs extrinsic)
- Factory/registry adds indirection

Key Components

- Flyweight – shared intrinsic state (TreeModel)
- Flyweight Factory – manages instances (TreeModelFactory)
- Client – supplies extrinsic state at use-time

Implementation in this project

Classes: TreeModel, TreeModelFactory.
TreeModel methods: Render.

Factory methods: GetTreeModel.

Flow: TreeModelFactory.GetTreeModel("Oak") returns a shared model reused by many tree instances.

Behavioral Design Patterns

Behavioral patterns focus on algorithms and the assignment of responsibilities between objects.

Command

Description

Encapsulates a request as an object, allowing parameterization of clients and supporting undo/redo, queuing, and logging.

Usage

Use when you need to decouple a sender from a receiver or support macro/undo operations.

Advantages

- Decouples invoker and receiver
- Supports undo/redo and macros
- Commands are composable

Disadvantages

- Adds extra classes per command
- Simple actions can feel over-engineered

Key Components

- Command – request interface (ICommand)
- Concrete Commands – actions (TurnOnCommand, TurnOffCommand)
- Receiver – does the work (Light)
- Invoker – triggers command (RemoteControl)

Implementation in this project

Classes: ICommand, TurnOnCommand, TurnOffCommand, Light, RemoteControl.

RemoteControl methods: Submit.

Light methods: TurnOn, TurnOff.

Flow: RemoteControl.SetCommand(new TurnOnCommand(light)); RemoteControl.PressButton() → Light.On().

Observer

Description

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

Usage

Use when multiple views depend on the state of one subject, or when you need event-style notifications.

Advantages

- Loose coupling between subject and observers
- Dynamic subscription/unsubscription
- Supports multiple observers

Disadvantages

- Order-of-notification pitfalls
- Careful memory management for long-lived observers

Key Components

- Subject – state owner (ISubject/WeatherStation)
- Observer – reacts to changes (IObserver/WeatherDisplay)

Implementation in this project

Classes: ISubject, IObserver, WeatherStation, WeatherDisplay.

WeatherStation methods: RegisterObserver, RemoveObserver, NotifyObservers, MeasurementsChanged, SetMeasurements.

WeatherDisplay methods: Subscribe, Unsubscribe, Update, Display.

Flow: WeatherStation.Register(WeatherDisplay); SetTemperature(...) → notifies WeatherDisplay.Update(...).

Strategy

Description

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Usage

Use when you need to switch algorithms at runtime, or avoid conditional branches selecting behaviors.

Advantages

- Algorithm interchangeability
- Open/Closed-friendly for new strategies
- Removes conditionals from clients

Disadvantages

- More objects to manage
- Clients must be aware of strategy semantics

Key Components

- Strategy – algorithm interface (ISortStrategy)
- Concrete Strategies – BubbleSortStrategy, QuickSortStrategy
- Context – selects and uses strategy (SortedList)

Implementation in this project

Classes: ISortStrategy, BubbleSortStrategy, QuickSortStrategy, SortedList.

SortedList methods: Sort.

Flow: SortedList.SetStrategy(new QuickSortStrategy()); SortedList.Sort() delegates to QuickSortStrategy.Sort(...).

Program.cs Explanation

Program.cs serves as the *driver* for demonstrating all implemented design patterns in this project. It creates objects, invokes methods, and prints outputs to show how each pattern works in practice. The file is structured into sections, each dedicated to a design pattern category (Creational, Structural, Behavioral).

Creational Patterns in Program.cs

- Factory: Demonstrates AnimalFactory creating a Dog object and invoking Speak().
- Builder: Uses CarBuilder and CarDirector to build a sports car.
- Fluent Builder: Constructs CarVehicle with chained WithEngine/WithTyres/WithColor calls.
- Memento: Shows saving and restoring Editor state with History.

Structural Patterns in Program.cs

- Bridge: Uses StandardRemote with a SonyTV to turn on, switch channel, and turn off.
- Composite: Builds a directory tree of files and subdirectories and prints it.
- Decorator: Wraps SimpleCoffee with MilkDecorator and SugarDecorator to add responsibilities.
- Facade: Uses HomeTheaterFacade to simplify movie watching operations.
- Flyweight: Creates TreeModel instances via TreeModelFactory, demonstrating object sharing.

Behavioral Patterns in Program.cs

- Command: Encapsulates Light actions into TurnOnCommand and TurnOffCommand, triggered by RemoteControl.
- Observer: WeatherStation notifies WeatherDisplay observers when SetMeasurements is called.
- Strategy: Switches between BubbleSortStrategy and QuickSortStrategy at runtime for sorting.