upGrad ENTERPRISE

Design Patterns & SOLID Principles in C#

By

Narasimha Rao T

Corporate Trainer and Mentor

Professional Development Trainer

tnrao.trainer@gmail.com

1. Introduction to Design Patterns

Definition:

Design Patterns are proven, reusable solutions to common software design problems. They represent best practices evolved over time.

Origin:

Popularized by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) in *Design Patterns: Elements of Reusable Object-Oriented Software*.

Categories:

1. **Creational Patterns** – Object creation mechanisms.
2. **Structural Patterns** – Class and object composition.
3. **Behavioral Patterns** – Communication between objects.

2. Why Do We Use Design Patterns?

- **Code Reusability** – Write once, use in multiple projects.
- **Maintainability** – Easier to modify and extend.
- **Scalability** – Design adapts as the application grows.
- **Communication** – Patterns give a common vocabulary to developers.
- **Best Practices** – Avoid reinventing the wheel.

3. Advantages of Design Patterns

- Reduces development time.
- Improves code readability.
- Promotes loose coupling.
- Encourages separation of concerns.
- Makes onboarding easier for new team members.

4. Creational Patterns in C#

4.1. Factory Method Pattern

Definition / Intent:
A creational design pattern that provides an interface for creating objects but lets subclasses decide which class to instantiate.
Instead of calling a constructor directly, you call a factory method.

Why We Use It:

- To encapsulate object creation logic.
- To decouple client code from concrete classes.
- To follow the Open/Closed Principle.

Advantages:

- Loose coupling between client and object creation code.
- Centralized creation logic, easier maintenance.
- Can easily add new types without modifying existing code.

Real-World Analogy:

Think of a restaurant kitchen — you don’t make your own dish; you place an order, and the kitchen (factory) decides how to prepare it.

Common Use Cases:

- Creating different parsers (XMLParser, JSONParser).
- Logging systems (FileLogger, DatabaseLogger).
- Game objects (different types of enemies).

4.2. Abstract Factory Pattern

Definition / Intent:

A creational pattern that provides an interface to create families of related or dependent objects without specifying their concrete classes.

Why We Use It:

- When a system should be independent of how its products are created.
- When we need to ensure that products in a family work together.

Advantages:

- Ensures consistency among products in a family.
- Promotes loose coupling.
- Makes it easy to swap entire product families.

Disadvantages:

- Can be more complex than needed for small systems.
- Adding new product families may require large changes.

Real-World Analogy:

Think of buying furniture from a specific brand. A single “factory” produces a whole set (sofa, table, chair) that match in style.

Common Use Cases:

- Cross-platform UI components (Windows vs Mac buttons/checkboxes).
- Database connectors (MySQL vs PostgreSQL client objects).
- Theming systems (Light Theme vs Dark Theme UI widgets).

4.3. Singleton Pattern

Definition / Intent:

A creational pattern that ensures a class has only one instance and provides a global point of access to it.

Why We Use It:

- When exactly one object is needed to coordinate actions across a system.
- To manage shared resources (configurations, logging, cache).

Advantages:

- Controlled access to the single instance.
- Reduces memory usage when only one instance is needed.
- Can be lazy-loaded.

Disadvantages:

- Can be misused as a global variable substitute.
- Harder to test due to hidden dependencies.
- In multi-threaded environments, must handle thread safety.

Real-World Analogy:

Like having one president for a country — there's only one official representative.

Common Use Cases:

- Logging services.
- Configuration managers.
- Connection pools.

4.4. Builder Pattern

Definition / Intent:

A creational pattern that separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Why We Use It:

- When an object needs to be constructed in multiple steps.
- When an object can have many optional parts or configurations.

Advantages:

- Step-by-step object creation.
- Avoids telescoping constructors (many constructor parameters).
- More readable and maintainable code.

Disadvantages:

- More code and complexity compared to directly creating objects.
- Might be unnecessary for simple objects.

Real-World Analogy:

- Building a custom burger at a fast-food place — you decide step-by-step (bun, patty, cheese, sauces) and then get the final product.

Common Use Cases:

- Creating complex documents (Word, PDF).
- Configuring HTTP requests.
- Constructing game characters with multiple optional attributes.

Comparison Table

Pattern	Focus	Key Benefit	Common Use Case
Factory	Create objects via interface	Decouple creation from use	Parsers, loggers
Abstract Factory	Create related object families	Consistency among products	Cross-platform UI
Singleton	Single instance of a class	Global access point, resource sharing	Logging, config
Builder	Step-by-step object creation	Flexible object construction	Complex models