By Saung Yu Wady
(Steph)

Software DESIGN PATTERN

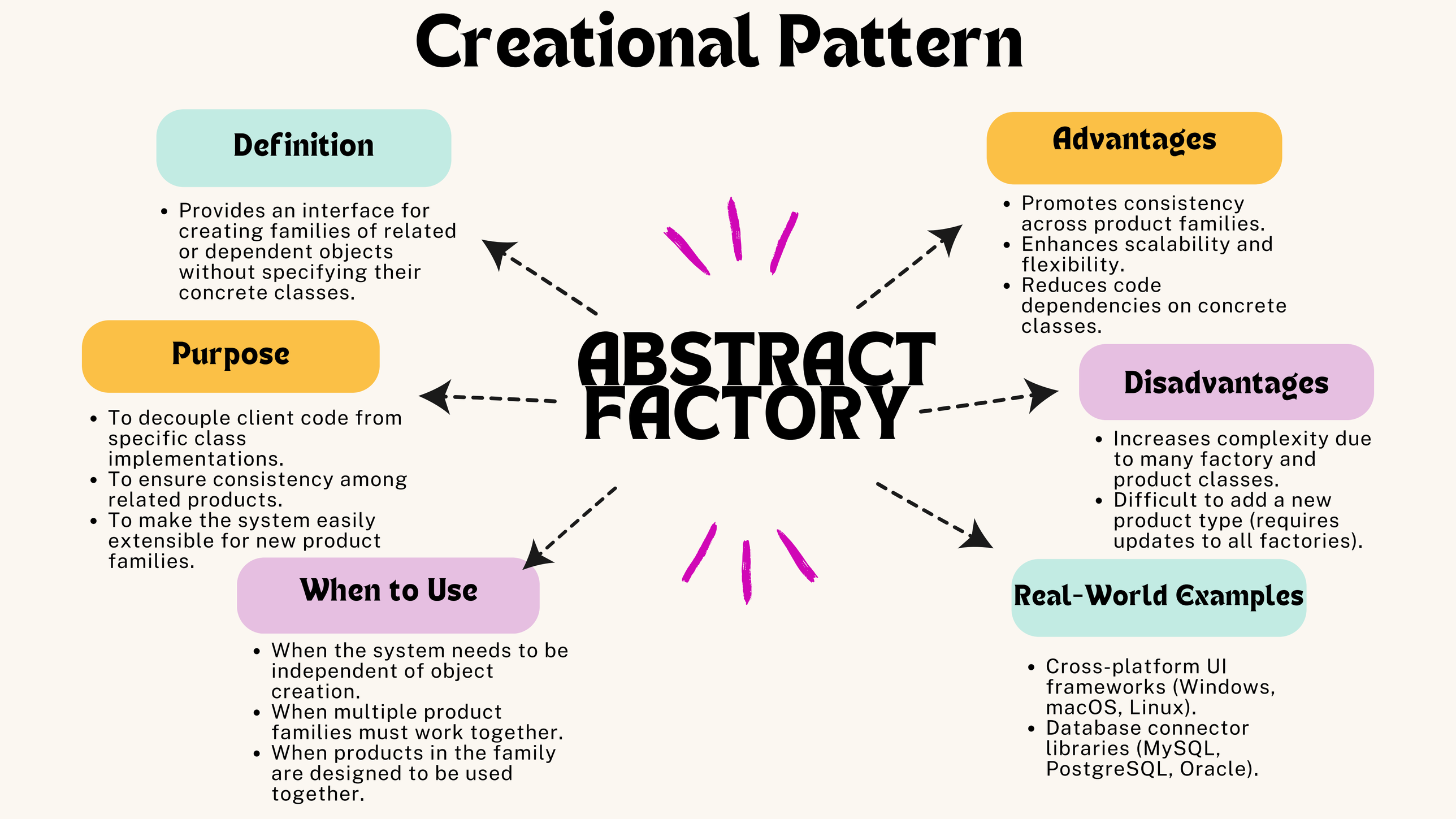
Agenda

1. Introduction to Design Patterns
2. Creational Pattern – Abstract Factory
3. Structural Pattern – Flyweight
4. Behavioral Pattern – Command
5. Comparison & Summary

What Are Design Patterns?

- Definition: Reusable solutions to common software design problems.
- Purpose:
  - Improve code flexibility and maintainability
  - Encourage best practices and reusability
  - Provide common vocabulary for developers
- Types of Design Patterns:
  - Creational – Object creation (e.g., Abstract Factory)
  - Structural – Object composition (e.g., Flyweight)
  - Behavioral – Object interaction (e.g., Command)

Creational Pattern



Definition

- Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Purpose

- To decouple client code from specific class implementations.
- To ensure consistency among related products.
- To make the system easily extensible for new product families.

When to Use

- When the system needs to be independent of object creation.
- When multiple product families must work together.
- When products in the family are designed to be used together.

Advantages

- Promotes consistency across product families.
- Enhances scalability and flexibility.
- Reduces code dependencies on concrete classes.

Disadvantages

- Increases complexity due to many factory and product classes.
- Difficult to add a new product type (requires updates to all factories).

Real-World Examples

- Cross-platform UI frameworks (Windows, macOS, Linux).
- Database connector libraries (MySQL, PostgreSQL, Oracle).

Structural Pattern

FLYWEIGHT PATTERN

Definition

- A pattern that allows programs to support large numbers of similar objects efficiently by sharing common parts of their state instead of duplicating them.

Purpose

- To reduce memory usage by sharing intrinsic (unchanging) data.
- To improve performance in applications with many similar objects.

When to Use

- When you have millions of similar objects.
- When object data can be split into:
  - Intrinsic state: shared, invariant data.
  - Extrinsic state: unique, context-specific data.

Advantages

- Significant memory and performance optimization.
- Centralized object management.
- Encourages better understanding of shared vs unique data.

Disadvantages

- Increased code complexity.
- Requires careful management of extrinsic state by clients.

Real-World Examples

- Text editors (each character shares font and style data).
- Game engines (trees, enemies, or particles sharing graphics data).
- Document rendering systems (shared glyphs).

Behavioral Pattern

COMMAND PATTERN

Definition

- Encapsulates a request as an object, allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations.

Purpose

- To decouple the object that issues a request from the one that executes it.
- To store, queue, and undo/redo operations flexibly.

When to Use

- When requests need to be issued, queued, or undone dynamically.
- When actions must be recorded or logged.
- When a system must decouple senders from receivers.

Advantages

- Promotes loose coupling between invoker and receiver.
- Supports undo/redo and macro commands.
- Makes it easy to add new commands without changing existing code.

Disadvantages

- Introduces many small command classes.
- Slightly increases overall system complexity.

Real-World Examples

- Remote controls (each button triggers a command).
- Text editors (Undo/Redo operations).
- GUI menu systems (each action represented as a command).

CONCLUSIONS

CATEGORY	PATTERN	PURPOSE	KEY BENEFIT
1. Creational	• Abstract Factory	• Create related objects without specifying their classes	• Flexibility in product creation
2. Structural	• Flyweight	• Share common data to reduce memory	• Memory efficiency
3. Behavioral	• Command	• Encapsulate requests as objects	• Decoupled and undoable actions

THANK YOU!