Factory Design Patterns - Table

Feature	**Simple**	**Factory**	**Abstract**
Creation	Single factory class	Delegated to subclasses via polymorphism	Interface for families of related objects
Inheritance	None	Extends the factory class	Each concrete is a varient
Flexibility	Less flexible, modifications needed	Highly flexible, follows Open/Closed	Compatibility and consistency
Complexity	Simple	More complex, requires inheritance and polymorphism	Complex, involves multiple factories

Simple Factory Characteristics - Creational

Centralized Factory: Single method/class creates all objects.

Simple Structure: Used for systems with limited number of object types.

e.g. Switch statement to run each constructor.

Use when you have a small, fixed number of object types and won't need to extend.

Abstract Factory Elements - Creational

Abstract Factory (interface): Set of creation methods. Each for different abstract product.

Concrete Factory: Implements creation methods. Each CF corresponds to variant of product.

Abstract Product: Interface for set of distinct but related products.

Concrete Product: Implements abstract products, *grouped by variant*. Each abstract product (e.g. sofa/chair) must be *implemented in all variants* (e.g. Victorian/Modern).

Use when code needs to work with families of related products but don't want to depend on concrete classes.

Singleton - Creational

Only one class instance (e.g. db connection, logs) created.

Race Condition: Two threads change same data at same time causes weird results.

Thread-Safe: keyword *Synchronized*.

Double-Checked Locking: check criteria before getting lock.

Disadvantages: Global state, tight-coupling (anti-pattern).

Design Patterns

Solutions to *common problems*. Guideline, not strict.

Provides common vocab: helps collaboration.

Creational: Right objects created in right situations. *Instantiation optimization*.

Structural: How classes/objects create larger structures. *System robustness*.

Behavioral: Interaction/communication between objects. *Efficient & flexible*.

OOP Terms

Encapsulation: Data + methods.

Abstraction: Information hiding.

Inheritance: Sub/super classes.

Polymorphism: Treat all subs like super.

OOP Principles

Single Responsibility: One job = one reason to change.

Dependency Injection: Receive dependencies from external source.

UML Use Case Diagrams

What a system does, **not how**.

Interactions between *actors and system*.

Components: Actors, use cases (functions), and relationships.

Context: Clarifications, constraints, exceptions, references, annotations.

Factory Method Elements - Creational

Creator ***(abstract)***: Declares factory method.

Concrete Creator: Implements factory method.

Product ***(interface)***: For objects created by factory method.

Concrete Product: Implements product interface.

Use when you need to create objects without specific class or expect product types to expand later.

Builder - Creational

Separated construction from representation. Useful for objects that have many optional components. Allows method call chaining.

Builder Interface: Define methods for building.

Concrete Builders: Implement interface.

Director ***(optional)***: Order and use of build instructions.

Product: Object that is constructed.

DAO - Structural

From Core J2EE: separates business/domain logic.

Primary Functions: Create, read, update, delete *(CRUD)*.

Data Source: Connection.

Domain/Business Object:

Data Access Object: CRUD operations. Abstracts access to DS.

Data Transfer Object: Models data (row). Follows Java Bean.

Java Bean Class: Default constractor, private vars with get/set. Implements Serializable.

Strategy - Behavioral

Defines family of algorithms **(intent)**, puts each in separate class resulting in *interchangeable objects*.

Can switch between algorithms at **runtime** without altering code.

Promotes flexibility, extensibility, and separation of concerns.

Strategy Interface: Defines set of behaviors all CS must implement.

Concrete Strategies: Implements SI. Each CS provides specific behavior (algorithm).

Context: Class that uses CS (delegates actual work). Contains reference to SI.

SOLID Principles

Purpose: To make software more *understandable, flexible*, and *maintainable*.

Modular: Reduces bugs when creating new code.

Single Responsibility: Classes.

Open Closed: New features don't alter code.

Liskov Substitution: Sub substitutable for super.

Interface Substitution: Divide large interface into smaller.

Dependency Inversion: Depend on abstractions, not concretes.

UML Sequence Diagrams

Interaction diagram. How objects interact in particular scenario over time.

Same diagrams from Web last term.