Participation 2 - Design Pattern Specifications & Comparisons

Comparison Sheet

Frison, Colby
Locklear, Emily
Natusch Zarco, Antonio
Parkman, Grant
Totah, James

Group H

CS 4213 - Fall 2025

	Intent	When to Use	Contrasts
Creational			
Factory Method	To vary the type of object used by an algorithm	When you need to create a single type of object, and you want to allow subclasses to determine which concrete class to instantiate.	1. **Scope of Control: Abstract Factory** and **Factory Method** control *which* object is created, while **Builder** controls *how* a complex object is constructed step-by-step. 2. **Object Management**: **Prototype** creates new objects by cloning an existing instance, whereas **Singleton** restricts instantiation to ensure only *one* instance of a class ever exists.
Abstract Factory	To create objects of a selected family of classes but you don't want the client to be affected by the selection.	When you need to create families of related or dependent objects, and you want to ensure that these objects are compatible with each other. Say, a UML diagramming tool. Once the user selects either UML 1.0 or UML 2.0, it is not necessary to test the user's choice each time a diagram element is created.	
Singleton	To design a class that has only a limited number of globally accessible instances	When you need to allow an application to connect to a database, for system configuration, system logs, etc.	
Prototype	To reduce the number of classes that share the same behavior and relationships	When you need your system to support Dynamically Loaded Classes, when cloning is more efficient than creating the object, etc.	
Builder	To separate the construction of a complex object from its representation.	When object creation involves multiple steps, many parameters, or different representations.	
Structural			
Composite	To treat individual objects and object compositions uniformly within a tree structure.	To represent part-whole hierarchies where clients can ignore the difference between single and composite objects.	1. **Relationship vs. Responsibility: Adapter** and **Bridge** focus on *relationships* between classes, Adapter makes existing interfaces work together, while Bridge decouples an abstraction from its implementation. **Decorator** focuses on dynamically adding *responsibilities* to an individual object. 2. **Complexity Hiding: Facade** provides a *simplified*, unified interface to a *complex* subsystem, while **Composite** builds *complex* tree structures from *simple* components, treating individuals and groups uniformly.
Adapter	To allow objects with incompatible interfaces to collaborate by **converting the interface of one class into another** that a client expects.	When **integrating existing classes with incompatible interfaces** into a system, particularly when modifying the source code of those classes is not possible.	
Facade	To simplify the interface for the client and abstract components using the simplified facade to interact with each feature.	When you need to access multiple separate components from one central simple interface.	
Decorator	To dynamically add responsibilities to an object by wrapping it, without altering its structure	When you need to add functionality to an object at runtime without using subclassing.	
Bridge	To decouple an abstraction from its implementation, using composition to allow both to evolve independently.	When an abstraction and its implementation must be extensible through subclassing without causing a combinatorial explosion of classes	
Behavioral			
Command	To encapsulate a request as an object, enabling parameterization, queuing, and undoable operations.	To decouple an operation's invoker from its executor, or to support features like undo/redo and command queues.	1. **Request Handling: Chain of Responsibility** passes a request along a chain of potential handlers until one processes it, promoting *loose coupling* between sender and receiver. In contrast, **Command** *encapsulates a request as an object*, allowing for parameterization and queuing of operations. 2. **Traversal vs. Interpretation**: **Iterator** provides a way to access the elements of a collection sequentially without exposing its underlying structure. **Interpreter** defines a grammar for a language and uses it to interpret and evaluate expressions, often leveraging a tree structure built from that grammar.
Visitor	To separate an algorithm from an object structure, allowing **new operations** to be added to that structure **without modifying its classe**s.	When you need to perform operations on a stable object structure **without altering its classes**, especially when those operations require type-specific behavior for each element in the structure.	
Iterator	To access elements of an aggregate without exposing the underlying data structure.	When multiple traversal options are required, and when more than one traversal on an aggregate at the same time is needed.	
Interpreter	To turn a set of language rules into objects that can be combined to check or evaluate commands.	When your program needs to understand and process simple commands or expressions, and you want a flexible way to define the rules for those commands	
Chain of responsibility	To give multiple objects a chance to handle a request by passing it along a chain.	When the specific handler for a request isn't known upfront and must be determined dynamically at runtime.	