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.	

Participation 2: Design Pattern Specifications & Comparisons

Colby P. Frison
Emily R. Locklear
James Totah
Grant P. Parkman
Antonio M. Natusch Zarco

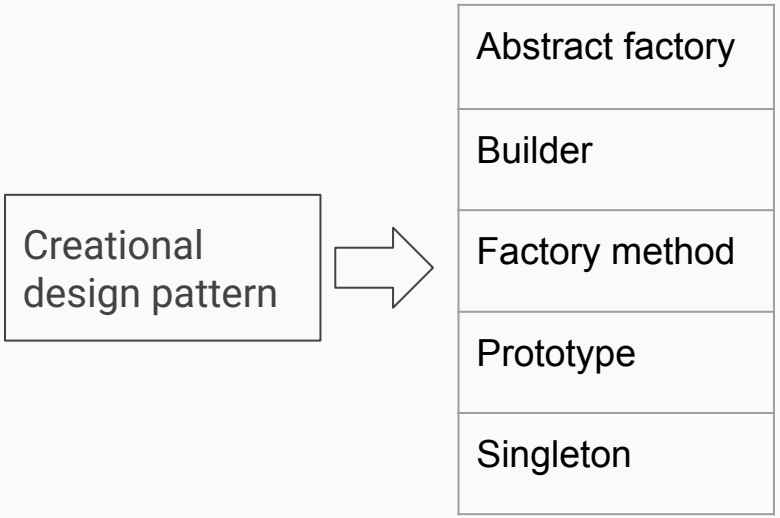
Group H

Overview - Our 15 patterns at a glance

Type	Name				
Creational	**Abstract Factory**	**Builder**	**Factory Method**	**Prototype**	**Singleton**
Structural	**Adapter**	**Bridge**	**Composite**	**Decorator**	**Facade**
Behavioral	**Visitor**	**Chain of Responsibility**	**Command**	**Interpreter**	**Iterator**

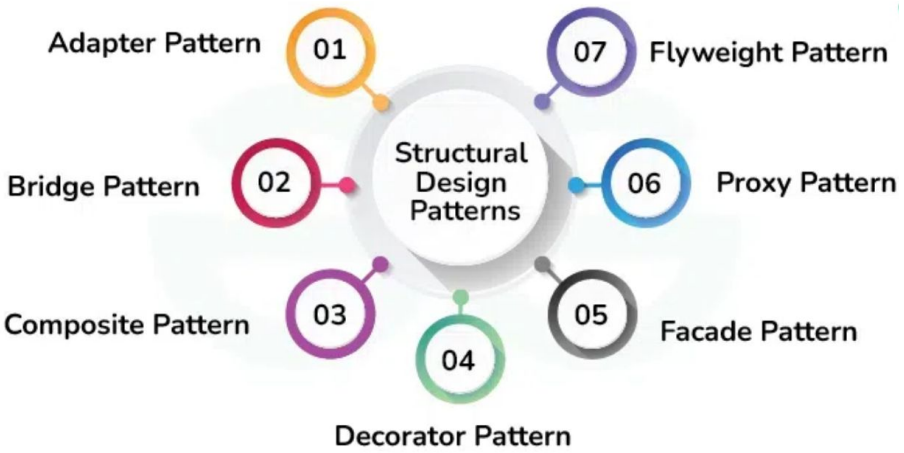
Creational

- **What they do:** Control *how objects are created*.
- **Why use them:** To add flexibility and decouple code from needing to know exactly which object to create.
- **The trade-off:** They make code more flexible but also more complex than just using the new keyword.
- **Key question:** "Is the added flexibility worth the extra code?"
- **Example:** Using a **Factory** to create different types of database connections.



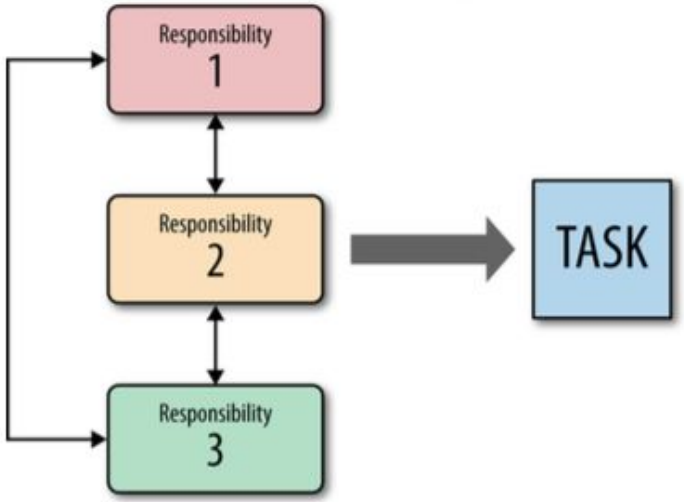
Structural

- **What they do:** Define *how objects fit together* to form larger structures.
- **Why use them:** To simplify relationships between objects and make incompatible interfaces work together.
- **The trade-off:** They add a layer of abstraction ("glue" code) which can slightly hurt performance.
- **Key question:** "Does the benefit of cleaner structure outweigh the cost of indirection?"
- **Example:** Using an **Adapter** to make a new payment service work with your old system's code.



Behavioral

- **What they do:** Manage *communication and responsibility* between objects.
- **Why use them:** To make object interaction more flexible and decoupled, making it easier to change how things work.
- **The trade-off:** They can make the flow of the program harder to follow and debug.
- **Key question:** "Is the decoupling worth making the program's behavior less obvious?"
- **Example:** Using an **Observer** to let multiple parts of an app update when data changes.



A Mini Case

SmartWrite

Brief overview

- **About SmartWrite:** Document processing and AI-powered workspace that helps in the analysis, editing, and interaction with various document types.
- **Features:**
  a. Supports PDF and Markdown file editing.
  b. AI-driven text analysis
  c. Model management system
  d. AI assistant for summarizing text, asking questions, etc.

- **Code Architecture:** Event-Driven Architecture
  - Components interact primarily through events rather than direct method calls.

About AI models…

- The application is intended for supporting multiple AI models (like GPT-4, Claude 3, Gemini Pro), allowing users to choose between them for document analysis and chat.
- Currently, the system uses conditional logic and direct method calls to handle model selection, error recovery, and switching between models.
- While this setup works, it can become difficult to extend or modify as new models and features are added.

```
simulateModelResponse(message, model, signal) {
    return new Promise((resolve, reject) => {
        const delay = Math.random() * 2000 + 1000; // 1-3 second delay

        this.pendingRequest = setTimeout(() => {
            this.pendingRequest = null;

            // Call the appropriate model-specific handler based on model ID
            // This makes it easy to replace with real API calls later
            let response;
            switch (model.id) {
                case 'gpt-4':
                    response = this.handleGPT4Request(message);
                    break;
                case 'gpt-3.5-turbo':
                    response = this.handleGPT35Request(message);
                    break;
                case 'claude-3-opus':
                    response = this.handleClaudeRequest(message);
                    break;
                case 'gemini-pro':
                    response = this.handleGeminiRequest(message);
                    break;
                case 'unstable-ai':
                    response = this.handleUnstableAIRequest(message);
                    break;
                default:
                    response = {
                        text: `AI response to: "${message}"`,
                        model: model.id,
                        timestamp: new Date().toISOString()
                    };
            }
```

Command Pattern

- Encapsulates each model request as its own object, making the system easier to extend and maintain.
- Removes the need for large switch/case statements and repetitive conditional logic.
- Enables features like queuing, logging, and undoing actions in the future.
- Improves code organization by decoupling the request logic from the invoker.
- Makes it simple to add new AI models or request types without changing existing code.

```
// Command Interface (with optional undo/redo as per GoF spec)
class ModelCommand {
  execute() {
    throw new Error('execute() must be implemented');
  }
  undo() {
    // Default: not supported
    throw new Error('undo() not implemented for this command');
  }
  redo() {
    // Default: not supported
    throw new Error('redo() not implemented for this command');
  }
  reversible() {
    // Default: commands are not reversible unless overridden
    return false;
  }
}

// Concrete commands for each model
class GPT4Command extends ModelCommand {
  constructor(modelManager, message) {
    super();
    this.modelManager = modelManager;
    this.message = message;
    this.result = null;
  }
  execute() {
    this.result = this.modelManager.handleGPT4Request(this.message);
    return this.result;
  }
  // Example: GPT4Command is not reversible
  reversible() {
    return false;
  }
}
```

Code Snippets

```
// Example command with undo/redo support
class ToggleModelSettingCommand extends ModelCommand {
  constructor(modelManager, settingKey, newValue) {
    super();
    this.modelManager = modelManager;
    this.settingKey = settingKey;
    this.newValue = newValue;
    this.previousValue = null;
  }
  execute() {
    this.previousValue = this.modelManager.getSetting(this.settingKey);
    this.modelManager.setSetting(this.settingKey, this.newValue);
    return `Set ${this.settingKey} to ${this.newValue}`;
  }
  undo() {
    this.modelManager.setSetting(this.settingKey, this.previousValue);
    return `Reverted ${this.settingKey} to ${this.previousValue}`;
  }
  redo() {
    return this.execute();
  }
  reversible() {
    return true;
  }
}
```

```
class Client {
  constructor() {
    this.history = [];
    this.redoStack = [];
  }
  executeCommand(command) {
    const result = command.execute();
    if (command.reversible()) {
      this.history.push(command);
      this.redoStack = [];
    }
    return result;
  }
  undo() {
    if (this.history.length === 0) return 'Nothing to undo';
    const command = this.history.pop();
    const result = command.undo();
    this.redoStack.push(command);
    return result;
  }
  redo() {
    if (this.redoStack.length === 0) return 'Nothing to redo';
    const command = this.redoStack.pop();
    const result = command.redo();
    this.history.push(command);
    return result;
  }
}

const manager = ModelManager.getInstance();
const client = new Client();

client.executeCommand(new GPT4Command(manager, 'Summarize this PDF'));
client.executeCommand(new SuperAICommand(manager, 'Analyze this document'));

client.executeCommand(new ToggleModelSettingCommand(manager, 'darkMode', true));
client.undo(); // will revert the darkMode change
client.redo(); // will re-apply the darkMode chang
```

Thank you!