Participation 2: Design Pattern Specifications & Comparisons

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

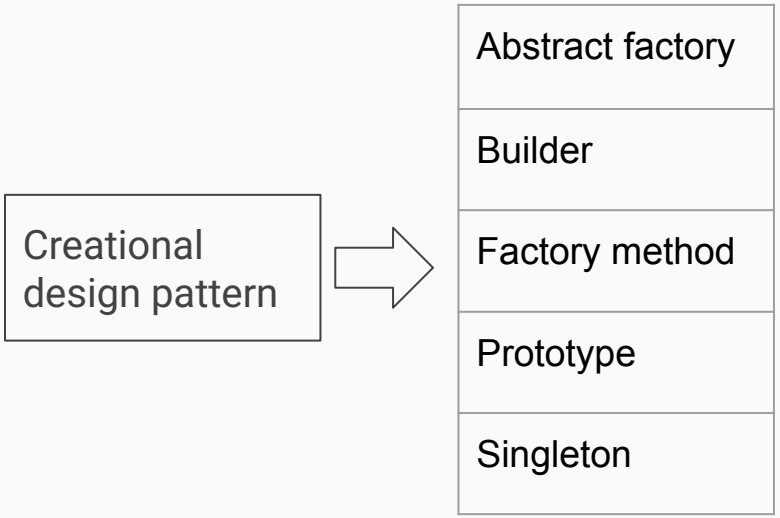
Group H

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**

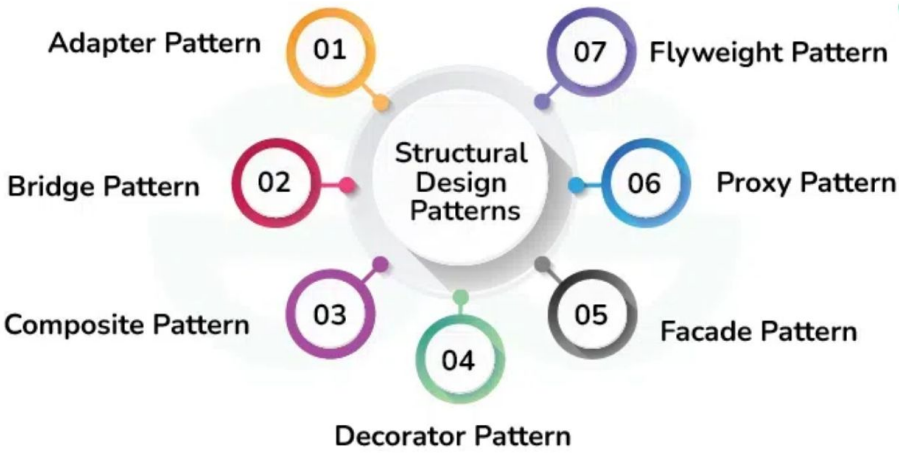
Creational

- **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.



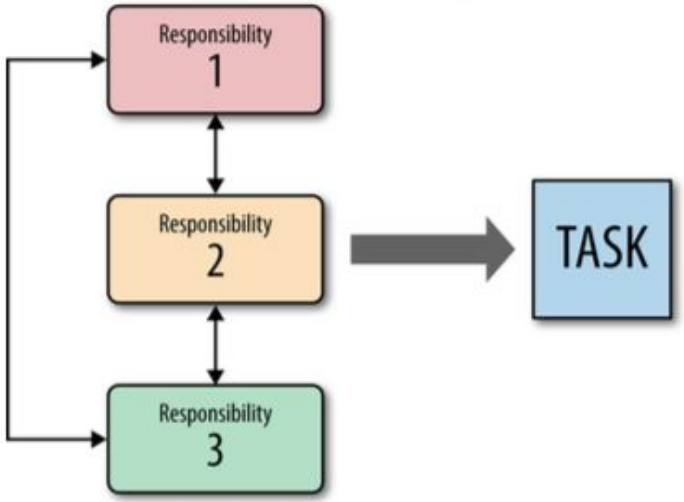
Structural

- **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!