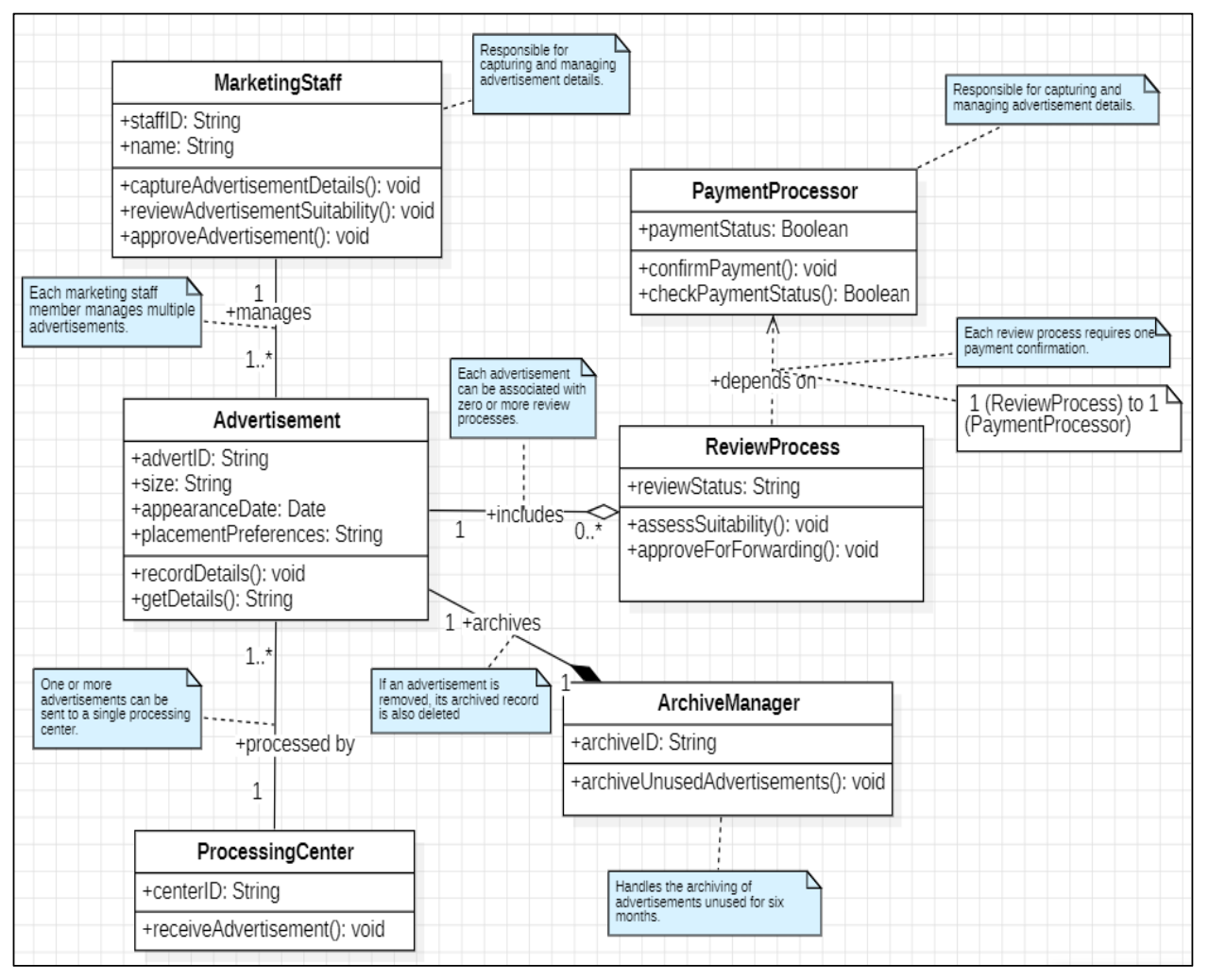
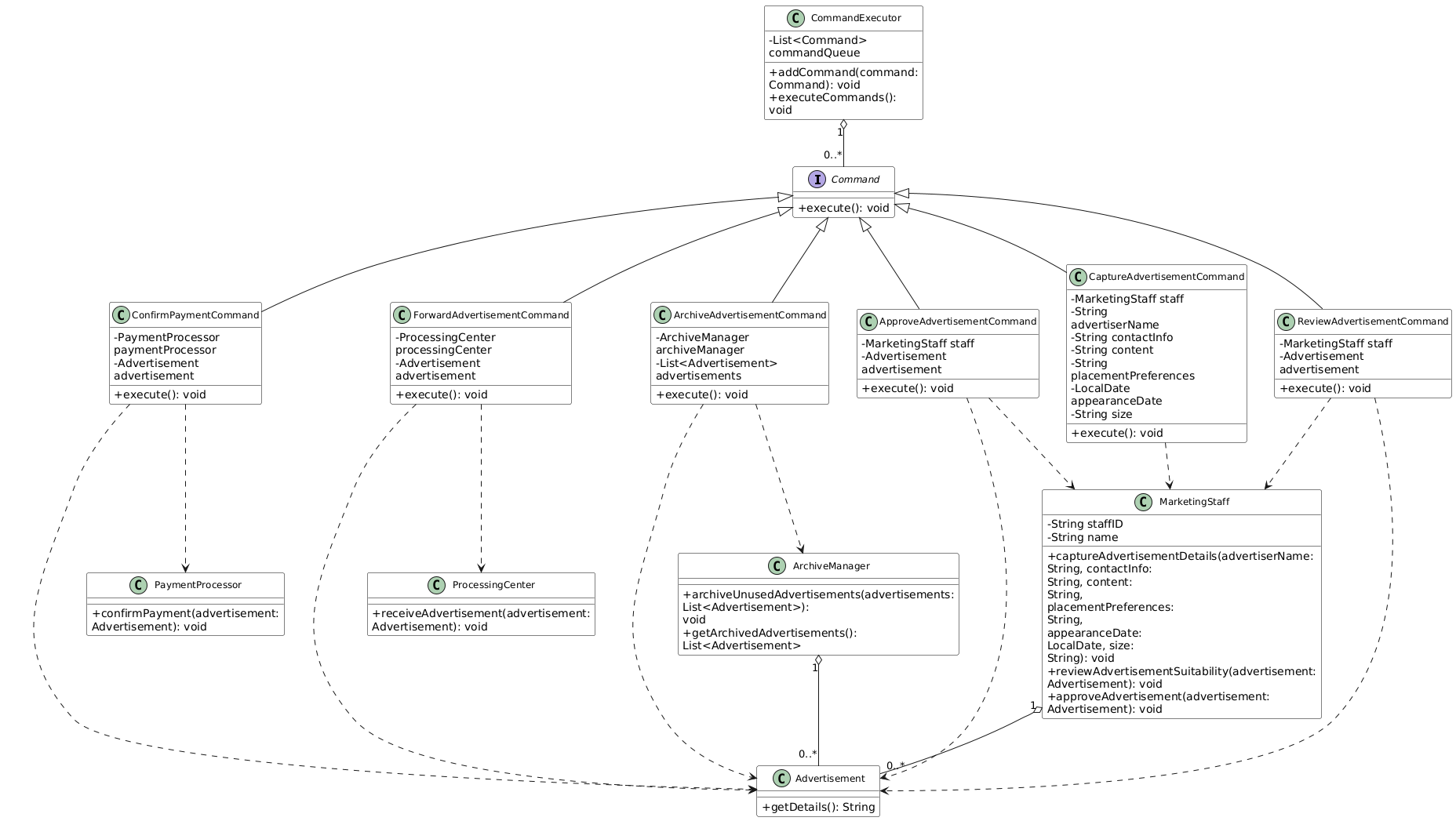
### **General Explanation of the Classes and Functions in the Command Pattern-Based Class Diagram**

The Command Pattern-based class diagram introduces a structured design for managing actions (e.g., capturing advertisements, reviewing suitability) by encapsulating each action in separate Command objects. Here's a breakdown of the key classes and their responsibilities:

Previous Class Diagram



Command pattern task diagram

#### **1. Command Interface**

* **Purpose:** Defines a standard interface (execute()) for all commands.
* **Functionality:**
  + Ensures all commands implement the execute() method, making it easy for the invoker to execute any command without knowing its specifics.

#### **2. Concrete Command Classes**

Each command class represents a specific action in the advertisement management system:

1. **CaptureAdvertisementCommand**
   * Encapsulates the logic for capturing advertisement details.
   * Calls MarketingStaff.captureAdvertisementDetails() to perform the action.
2. **ReviewAdvertisementCommand**
   * Encapsulates the logic for reviewing an advertisement's suitability.
   * Calls MarketingStaff.reviewAdvertisementSuitability().
3. **ConfirmPaymentCommand**
   * Handles payment confirmation for an advertisement.
   * Calls PaymentProcessor.confirmPayment().
4. **ApproveAdvertisementCommand**
   * Approves an advertisement after review and payment confirmation.
   * Calls MarketingStaff.approveAdvertisement().
5. **ForwardAdvertisementCommand**
   * Forwards approved advertisements to the processing center.
   * Calls ProcessingCenter.receiveAdvertisement().
6. **ArchiveAdvertisementCommand**
   * Archives unused or outdated advertisements.
   * Calls ArchiveManager.archiveUnusedAdvertisements().

#### **3. Invoker: CommandExecutor**

* **Purpose:** Manages a queue of commands and executes them sequentially.
* **Functionality:**
  + addCommand(Command command): Adds a command to the queue.
  + executeCommands(): Executes all queued commands and clears the queue.
* **Advantage:** Decouples the workflow execution logic from the specific command implementations, making the system more flexible.

#### **4. Receivers**

These classes perform the actual operations associated with commands:

1. **MarketingStaff**
   * Captures advertisement details, reviews suitability, and approves advertisements.
   * Functions like captureAdvertisementDetails() and reviewAdvertisementSuitability() are called by specific command objects.
2. **PaymentProcessor**
   * Confirms payment for advertisements.
   * confirmPayment() ensures that payment is completed before further processing.
3. **ProcessingCenter**
   * Receives approved advertisements for publication.
   * receiveAdvertisement() handles the transfer of advertisement details.
4. **ArchiveManager**
   * Manages archiving of unused or outdated advertisements.
   * archiveUnusedAdvertisements() ensures that old advertisements are removed systematically.

#### **5. Client (e.g., Task2 Class)**

* **Purpose:** Sets up the system by creating command objects, associating them with the appropriate receivers, and assigning them to the invoker (CommandExecutor).
* **Example Workflow:**
  + Create a CaptureAdvertisementCommand with MarketingStaff as the receiver.
  + Add the command to CommandExecutor.
  + Execute all commands to trigger the workflow.

### **How the Command Pattern-Based Design Differs from the Attached Class Diagram**

| **Aspect** | **Attached Class Diagram** | **Command Pattern-Based Diagram** |
| --- | --- | --- |
| **Action Handling** | Actions like capturing, reviewing, and approving are directly handled within classes like MarketingStaff. | Actions are encapsulated in Command classes, decoupling the logic from the invoker. |
| **Workflow Execution** | Sequential workflow logic is hardcoded across classes. | Managed dynamically by the CommandExecutor, supporting flexible workflows. |
| **Extensibility** | Adding a new action requires modifying existing classes, leading to tight coupling. | New actions can be added by introducing new Command classes without changing existing code. |
| **Decoupling** | The invoker (MarketingStaff, PaymentProcessor, etc.) knows the details of each operation it performs. | The invoker interacts only with Command objects, which delegate actions to receivers. |
| **Reusability** | Logic is tightly coupled with the class implementing it. | Logic is encapsulated in reusable Command objects. |
| **Dynamic Behavior** | No support for dynamic behavior; workflows are fixed. | Commands can be queued, reordered, or executed conditionally, allowing dynamic workflows. |
| **Focus of Classes** | Classes like MarketingStaff and ReviewProcess mix workflow and operational logic. | Workflow logic is separated into CommandExecutor, while operational logic remains with receivers. |

### **Advantages of the Command Pattern-Based Diagram**

1. **Modularity:**
   * Actions are encapsulated into individual command classes, reducing the complexity of other classes like MarketingStaff and ArchiveManager.
2. **Decoupling:**
   * The invoker (CommandExecutor) doesn’t need to know the details of the receivers (e.g., MarketingStaff, ProcessingCenter).
3. **Flexibility:**
   * You can easily change the sequence of actions by reordering commands in the CommandExecutor.
4. **Scalability:**
   * Adding new actions or workflows is simple—just create a new Command class without modifying existing code.
5. **Reusability:**
   * Commands can be reused in different workflows or contexts, making the system more adaptable.

### **Conclusion**

The attached class diagram focuses on direct interactions between classes and workflows, while the Command Pattern-based diagram separates concerns by encapsulating actions into commands. This makes the system easier to maintain, extend, and adapt to new requirements. The Command Pattern-based design is especially beneficial for complex workflows with multiple actions and dynamic requirements.