CSCC01 – Introduction to Software Engineering

Design Patterns

SOLID Design Principles

- Five design principles that promote software maintainability and reusability
  - **S**ingle Responsibility Principle
  - **O**pen-Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle

SOLID Design Principles

- Single Responsibility Principle
  - *A class should have one reason to change.*
- Open-Closed Principle
  - *Software entities should be open for extension but closed for modification.*
- Liskov Substitution Principle
  - *Subtypes must be substitutable for their base types.*
- Interface Segregation Principle
  - *Clients should not be forced to depend on methods that they do not use.*
- Dependency Inversion Principle
  - *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - Abstractions should not depend on details. Details should depend on abstractions.

What is a Design Pattern?

- A description of communicating objects and classes that are customized to **solve** a general design **problem** in a particular **context**
- Gamma et al. described 23 design patterns divided into three categories:
  - Creational patterns
  - Structural patterns
  - Behavioral patterns

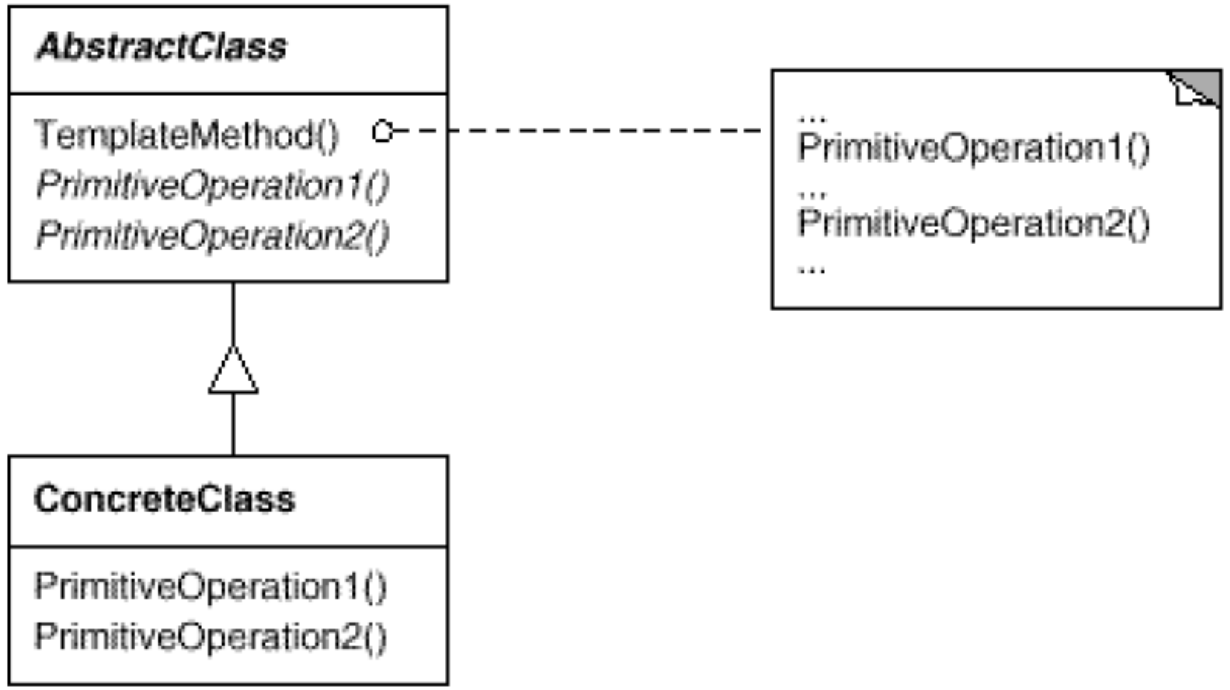
Design Patterns - Categories

- Creational Patterns
  - Concern the process of object creation
  - Six patterns: **Factory Method**, **Abstract Factory**, **Singleton**, **Prototype**, **Builder**, **Object Pool**
- Structural Patterns
  - Deal with the composition of classes or objects
  - Seven patterns: **Adapter**, **Bridge**, **Composite**, **Decorator**, **Façade**, **Flyweight**, **Proxy**
- Behavioural Patterns
  - Characterize the ways in which classes or objects interact and distribute responsibility
  - Ten patterns: **Chain of Responsibility**, **Command**, **Interpreter**, **Iterator**, **Mediator**, **Memento**, **Observer**, **State**, **Strategy**, **Template Method**

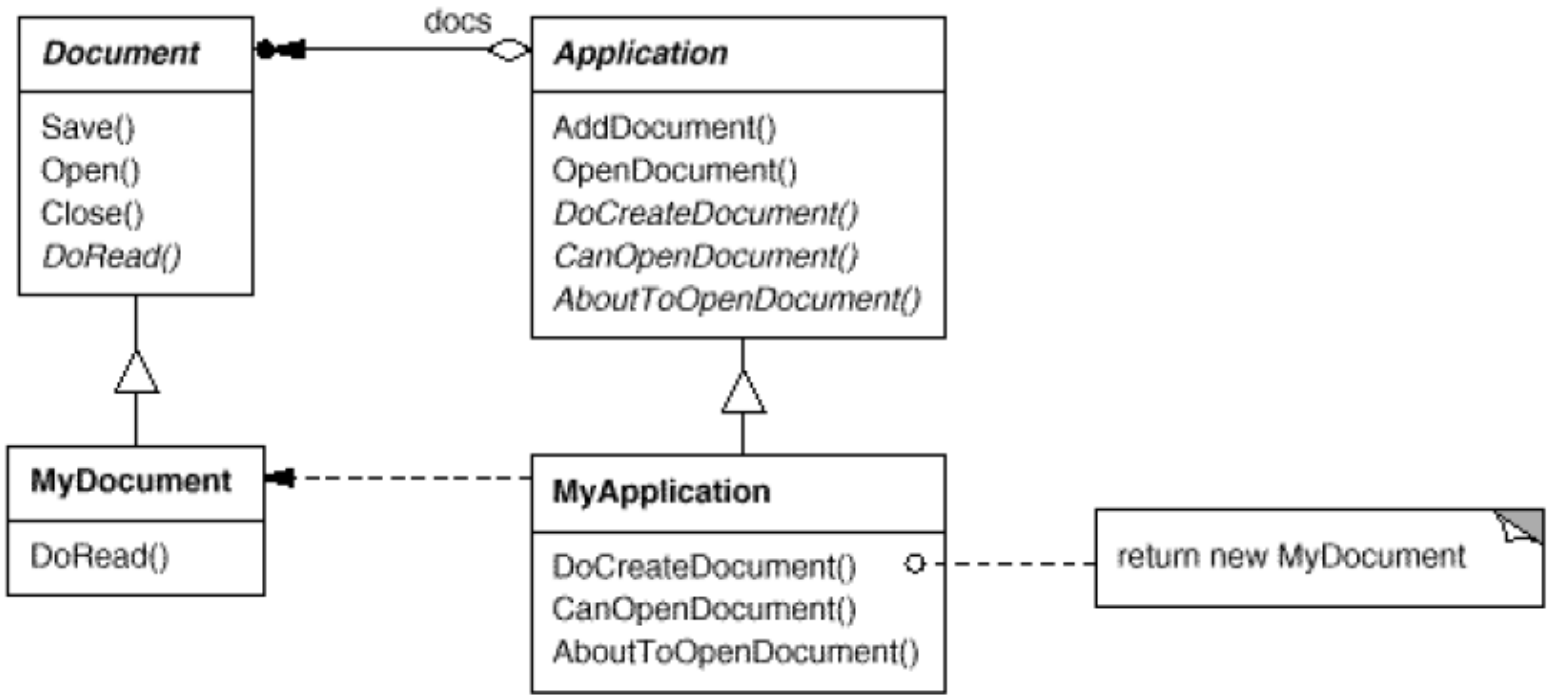
Gamma et al., Design Patterns: Elements of Reusable Object-oriented Software, 1995

Template Method

- Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

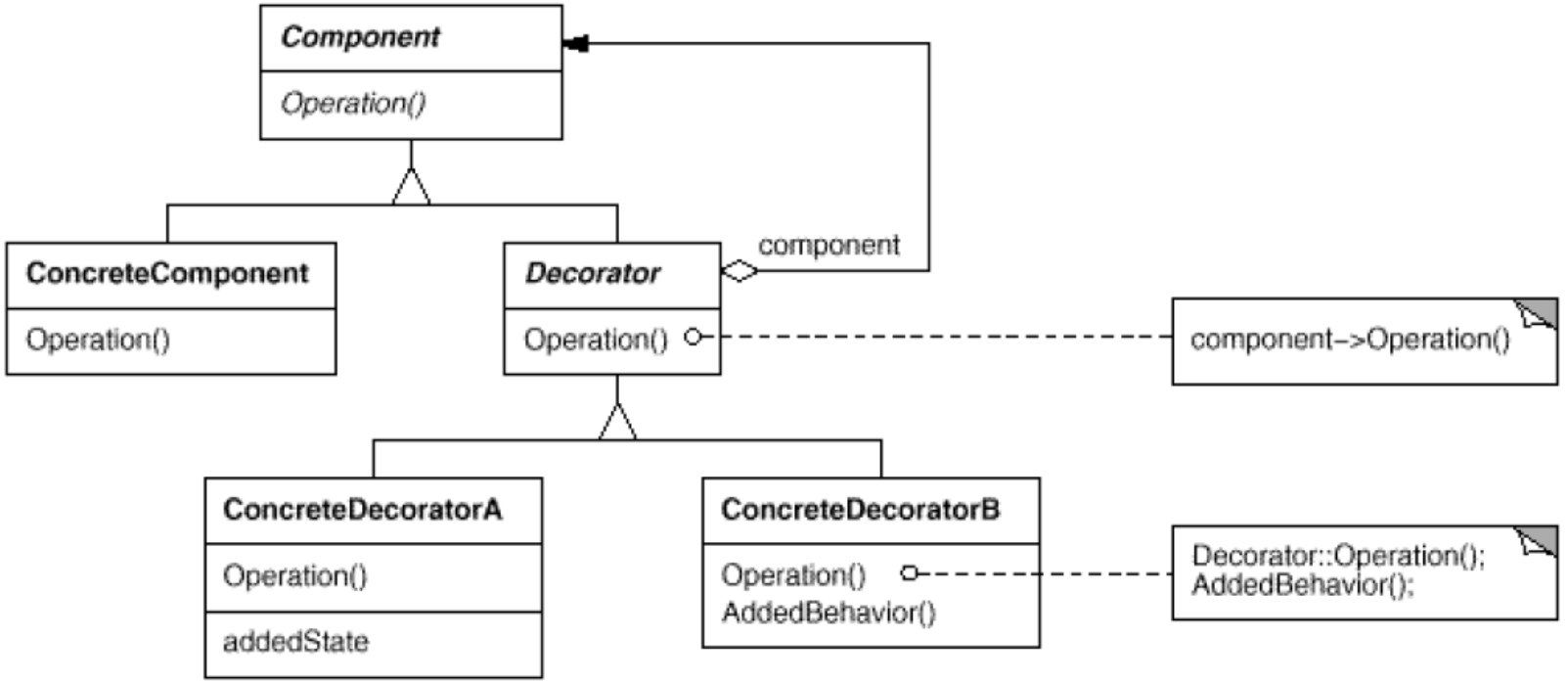


Template Method (Example)

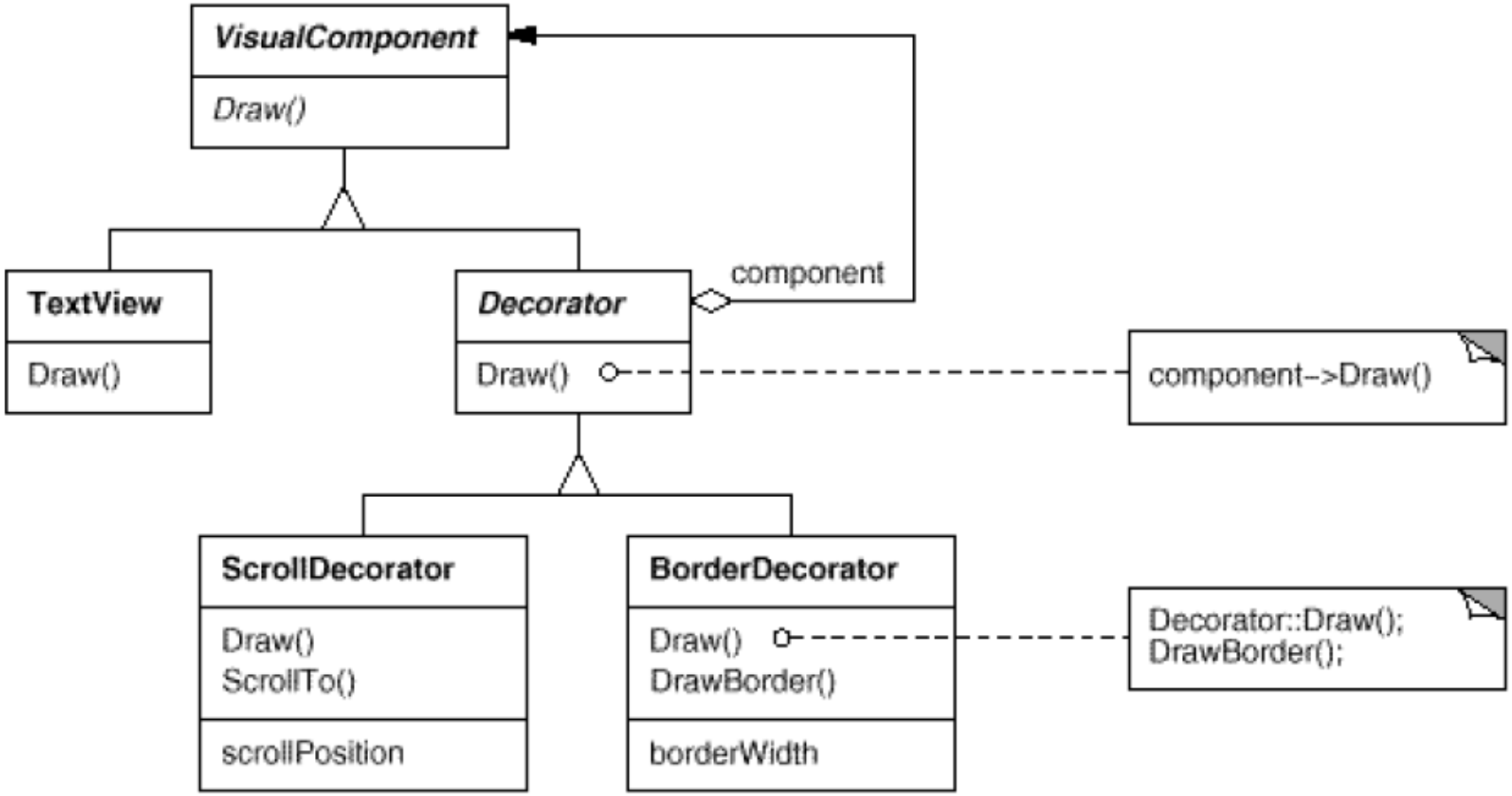


Decorator

- Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

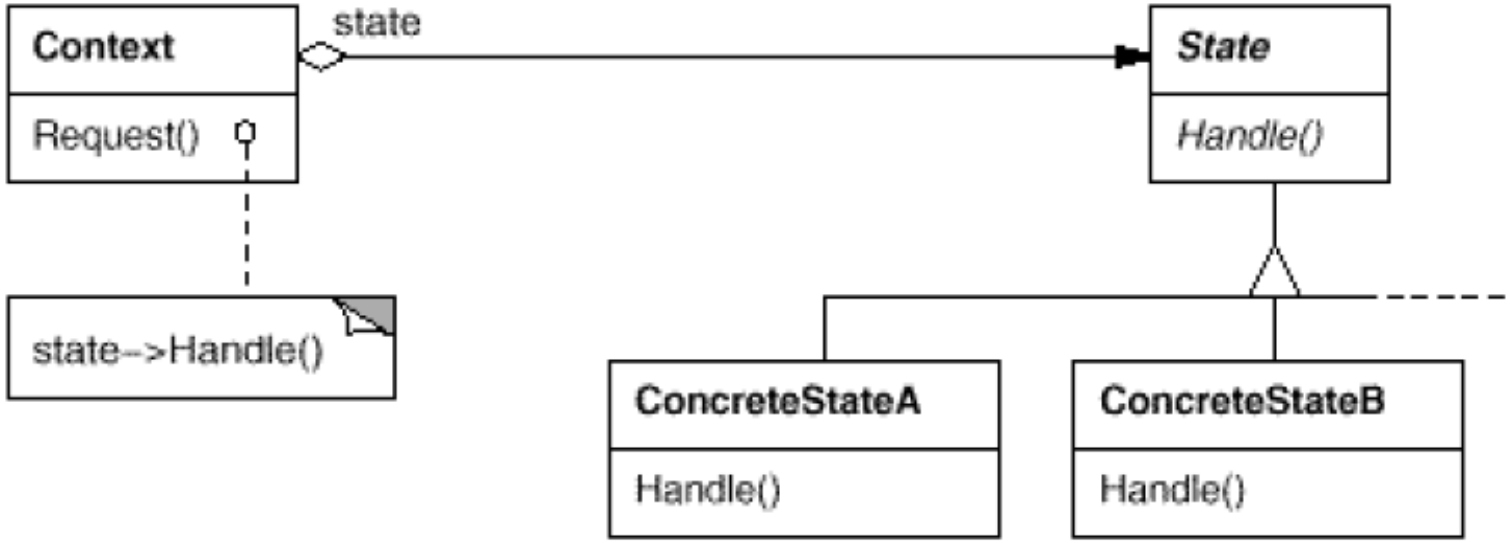


Decorator (Example)

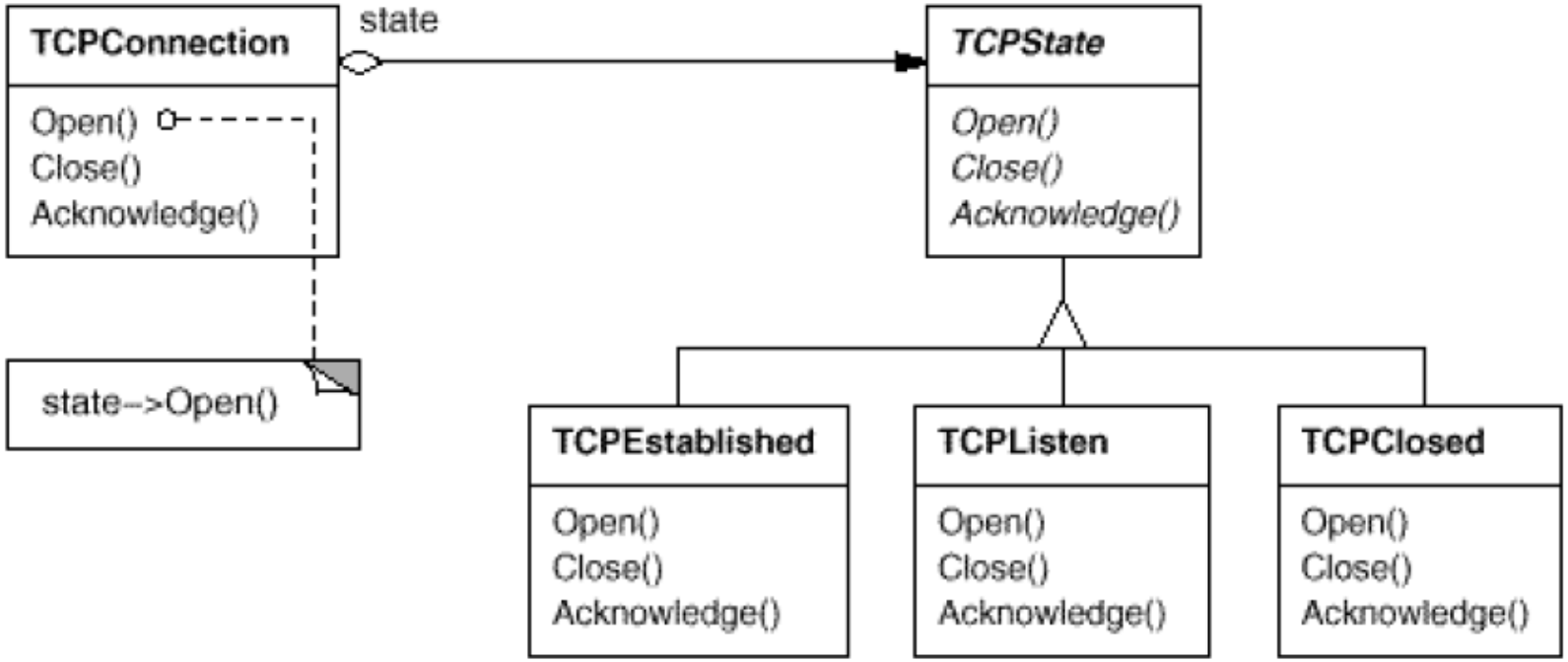


State

- Intent: Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

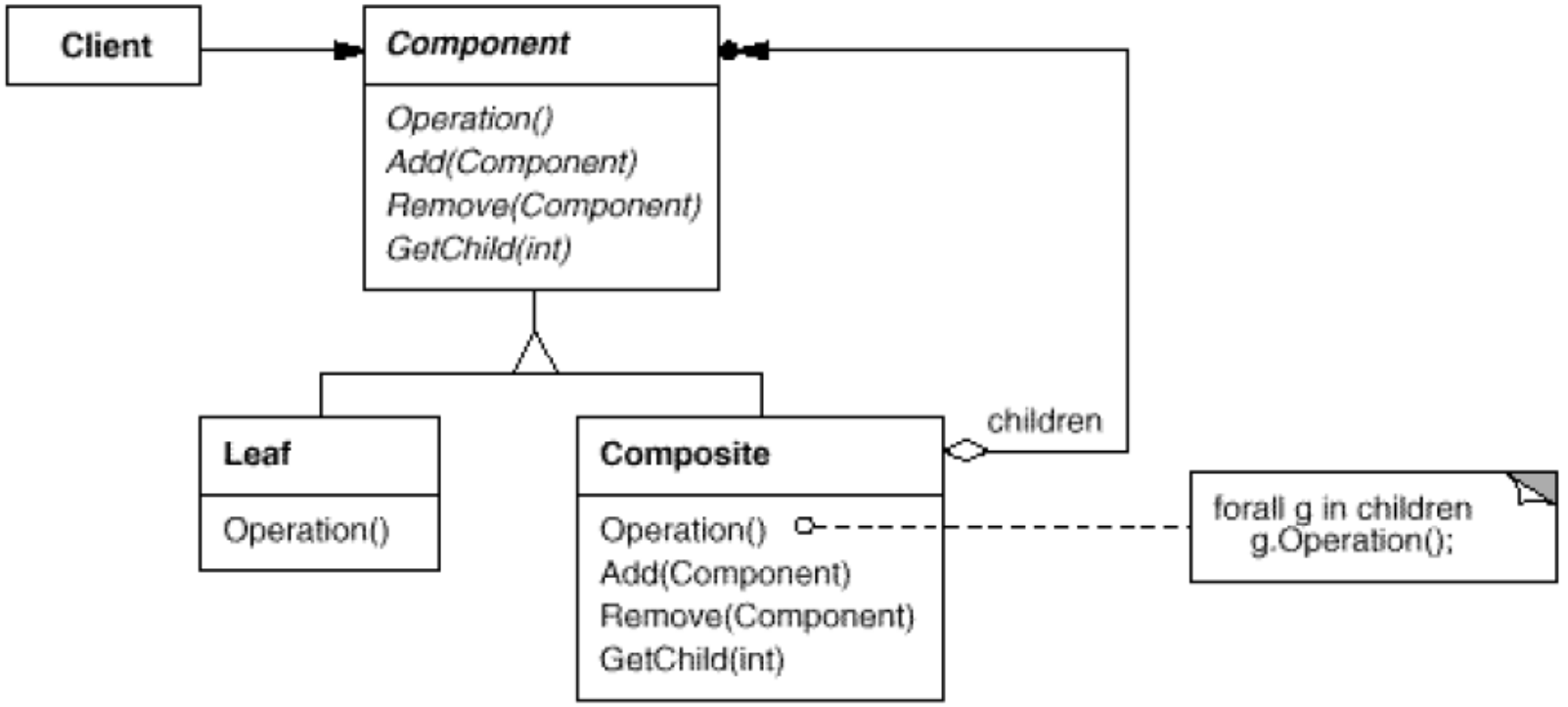


State (Example)

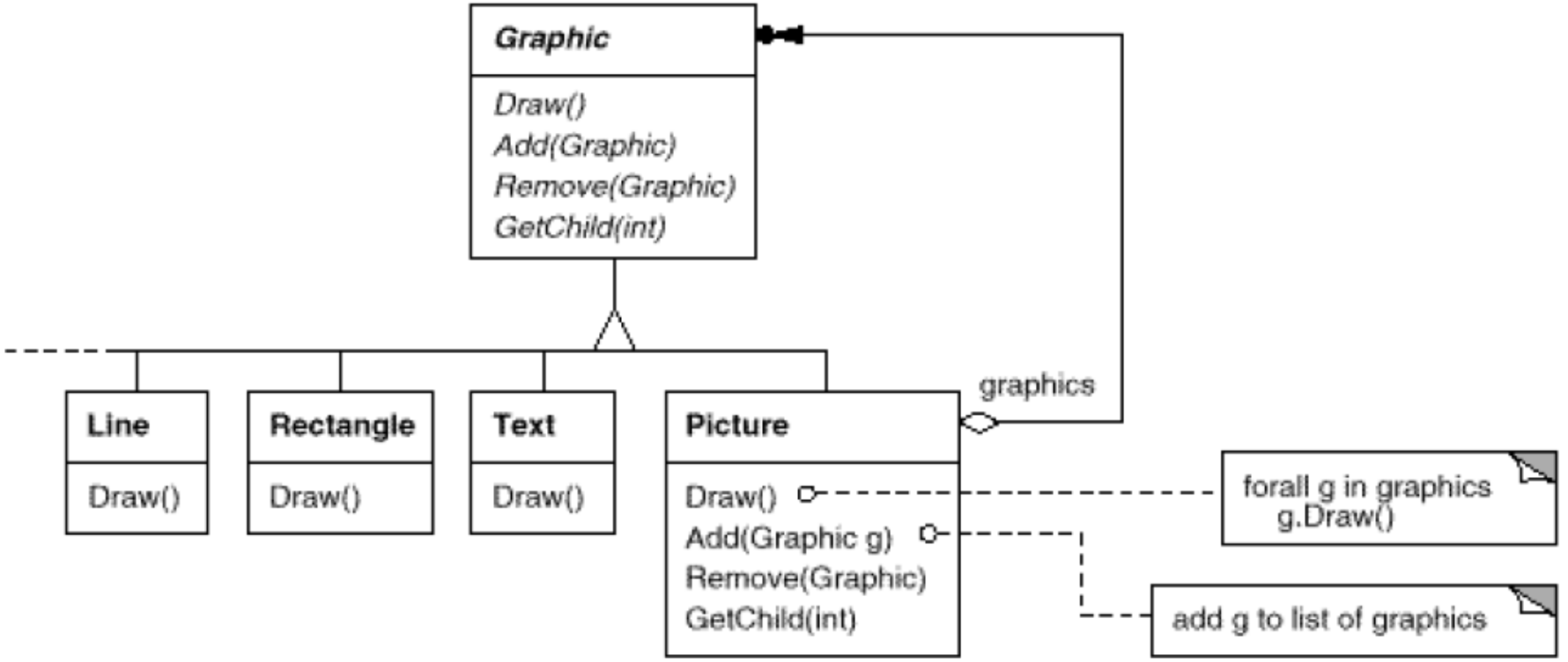


Composite

- Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

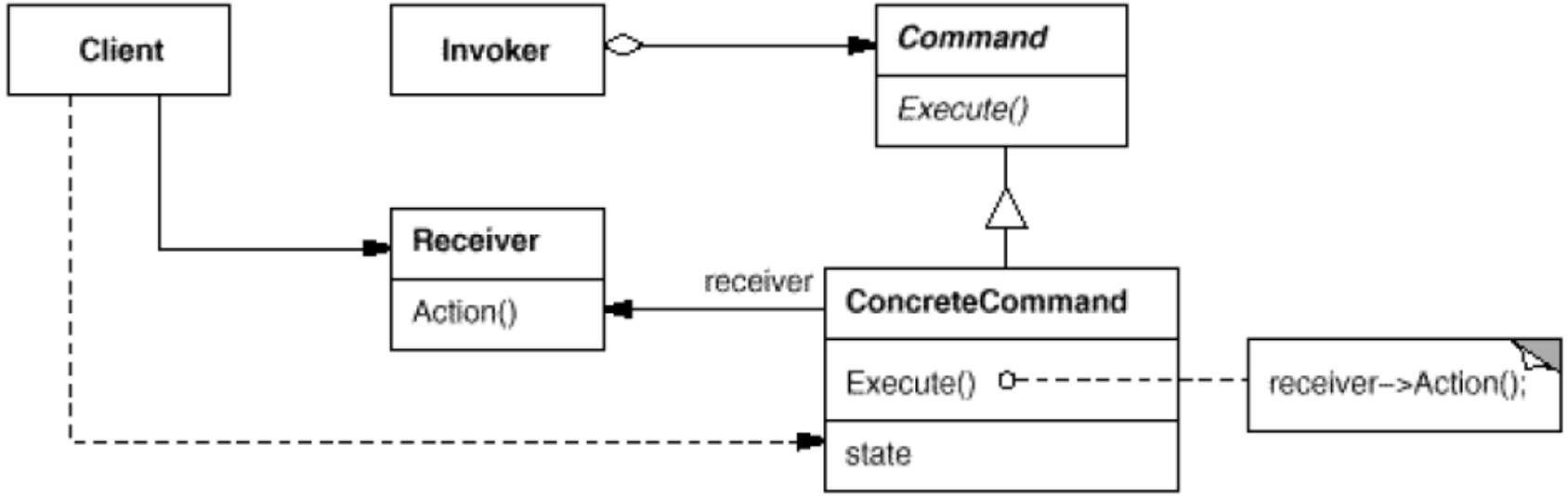


Composite (Example)

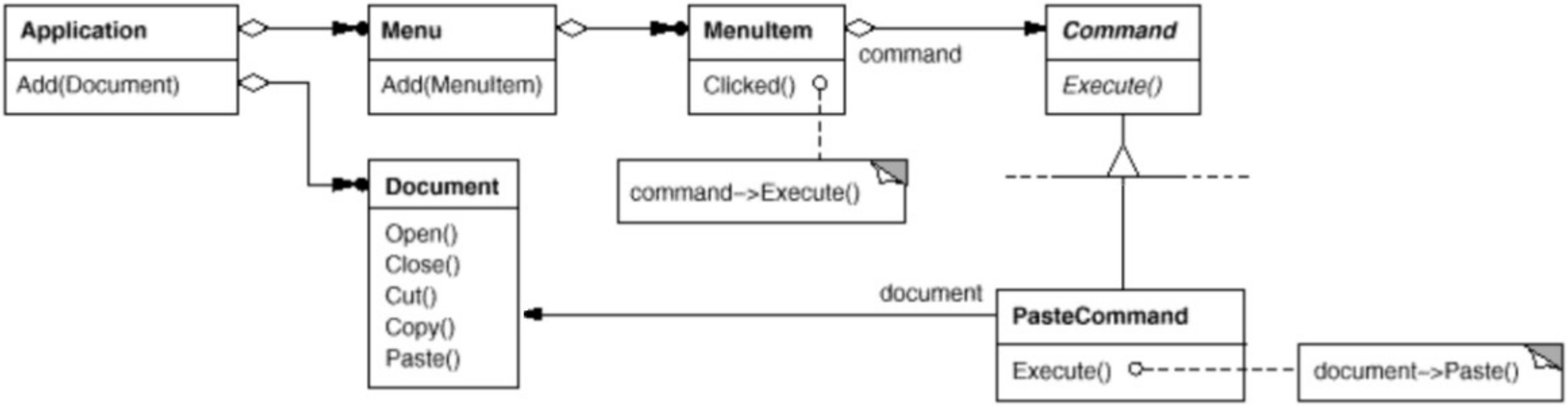


Command

- Intent: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

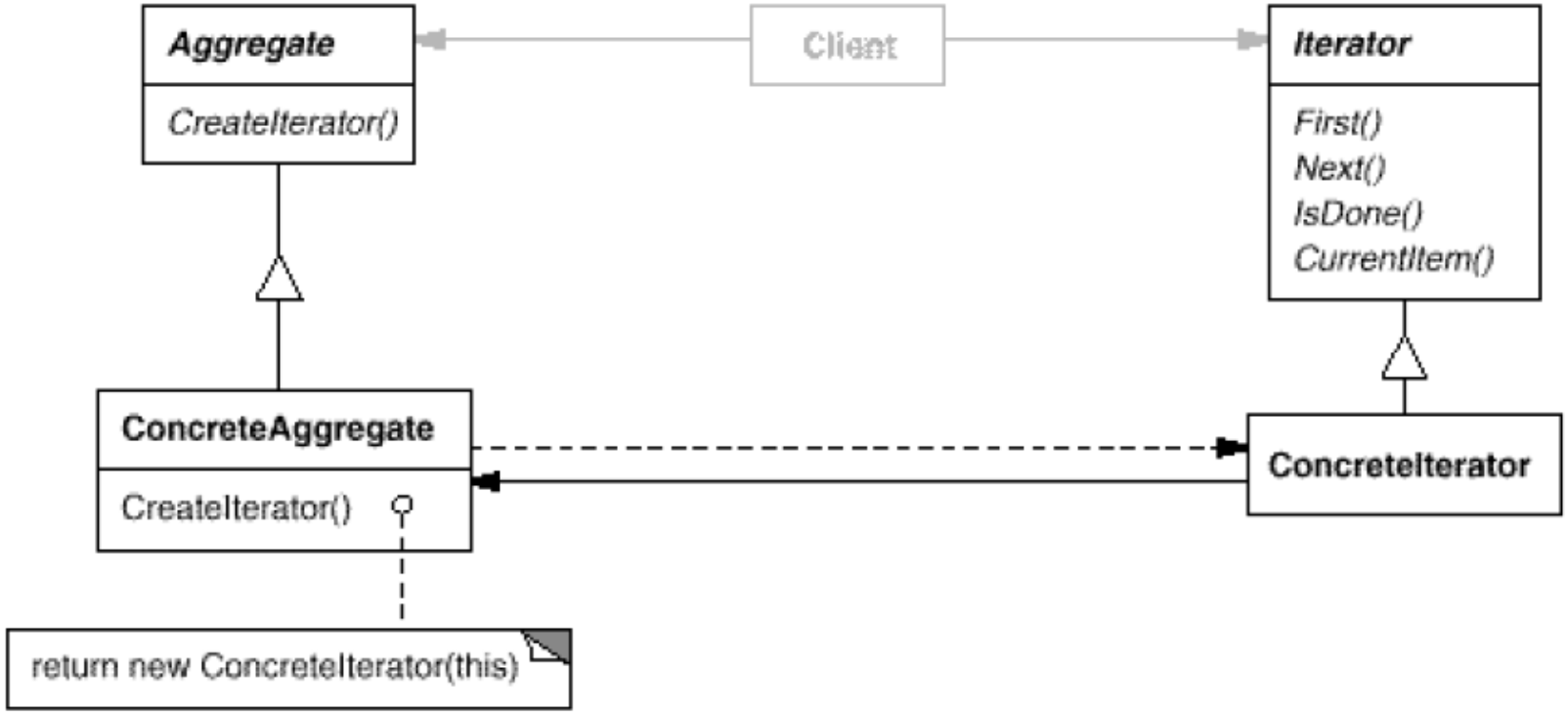


Command (Example)



Iterator

- Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Iterator (Example)