COMP 2511
Object Oriented Design & Programming

Week 10

Command Pattern
(Encapsulating Method Invocation)

A Case Scenario

Imagine, a text editor application which:

(1) Implements a generic Button class, that can represent different types of buttons e.g., tool bar button, dialog button, but clicking each button does a different thing

- How to implement click handlers of different button types?
- One possible design solution
  - Create a sub class for each button type, that has all the operations that must be executed, after the button has been clicked
- Problem: You could end up with tons of sub classes

(2) Supports copying text, which could be invoked from several places (e.g., pressing a tool bar button, or keyboard Ctrl-C, or click on the context menu item

- Problem: You will have to duplicate the code of copying operation in two other places.

Command Pattern

Motivation

- Need a way to decouple the requester of a particular action from the object that performs the action

Intent

- The command pattern is a behavioural design pattern that encapsulates requests and operations into their very own objects called commands
- If the operation had any parameters, they become fields of the new command class.
- Most commands serve as links between clients, which trigger requests and receiving objects, which handle them by performing some operations.

Command Pattern

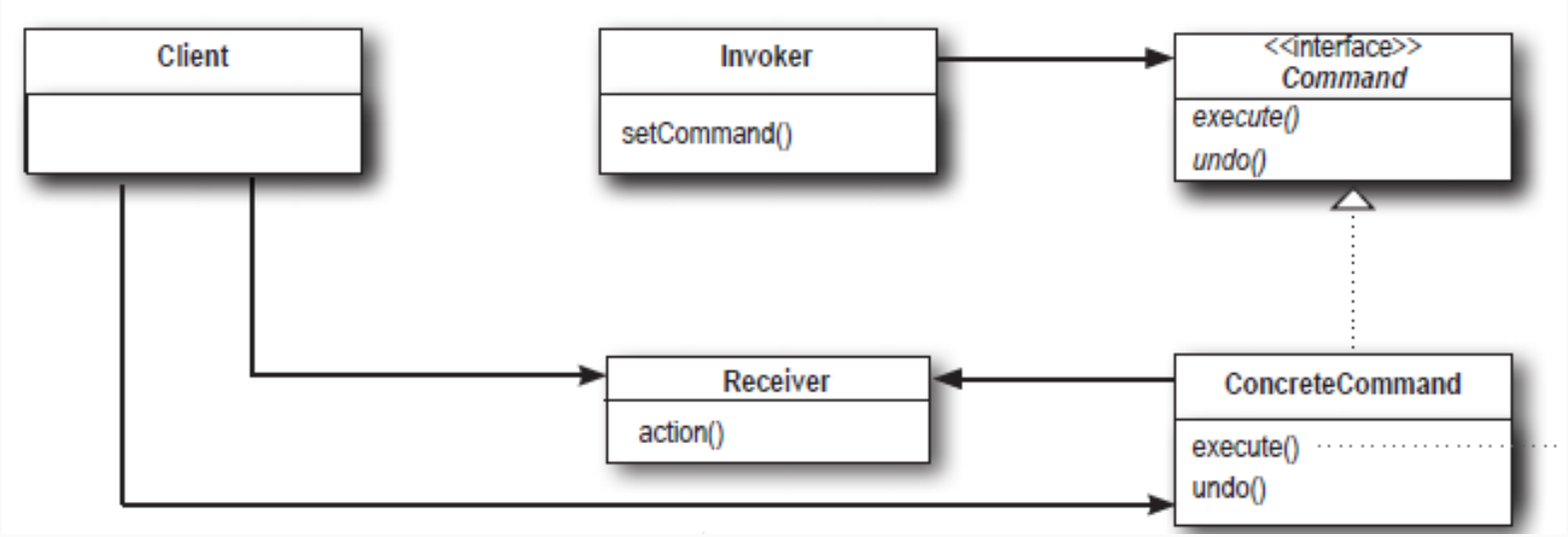
The pattern requires implementing four key components: the **Command** object, the **Receiver**, the **Invoker** and the **Client**

The Client is responsible for creating a ConcreteCommand and sets a Receiver for the command

An invoker is an object that **makes a request calling execute() on the given command, but doesn't know how the command has been implemented.** It only knows the command's interface.

A command is an object whose role is to **store all the information required for executing an action**, (the method to call, the method arguments, and the **Receiver** that implements the method).

Provides a simple **execute()** method which asks Receiver of the command to carry out operation



A receiver is an object that **performs a set of cohesive actions**. It's the component that performs the actual action when the command's execute() method is called.

The ConcreteCommand creates a binding between an action and a Receiver. When the Invoker makes a request calling **execute()**, the ConcreteComponent carries it out by calling one or more actions on the Receiver

When to apply the Command Pattern

- The command pattern is useful when:
  - The invoker should be decoupled from the object handling the invocation
  - A history of requests is needed
  - You need callback functionality or undo operations
- For example, applying the Command pattern to our example with the text editor, will eliminate the need for tons of Button subclasses
  - A button object will delegate the request to a linked command object upon receiving a click from a user. The command will either execute an operation on its own or delegate it to one of the business logic objects.
- Lecture Demo:
  - Example 1: Using a remote to control a light switch
  - Example 2: Text Editor

Command Pattern in the Java Library

- Here are some examples of Commands in core Java libraries:
  - All implementations of **java.lang.Runnable**
  - All implementations of **javax.swing.Action**

Command Pattern: Pros and Cons

Pros

- Decouples classes that invoke operations from classes that perform them
- Allows reversal (undo) of operations
- Allows deferred execution of operations
- Follows the *Open/Closed Principle*

Cons

- Increases overall code complexity by creating multiple Command classes that can make your design look cluttered

Visitor Pattern

Review: Method Overloading vs Method Overriding

Lecture Demo:

- Method overriding
- Method overloading
- Double dispatch

Double Dispatch

- Overloaded methods: The compiler uses the early (or static) binding for overloaded methods:
  - **Early** because it happens at compile time, before program is launched.
  - **Static** because it can't be altered at runtime.
- Double Dispatch is a technique that allows using dynamic binding alongside with overloaded methods
- The Visitor pattern is built on this principle

A Case Scenario

- A geographic information app uses a graph structure to represent the map of the location, where:
  - each node in the graph represents cities, towns and places of interest (e.g., sight-seeing, theatres etc.)
  - each node is represented by its own class
- A new requirement – export the graph as an XML
- One possible design solution -
  - add an export method for each node type and use the Composite pattern to go over the graph, executing this method for each node.
  - A simple and elegant solution that leverages polymorphism to avoid coupling to concrete node classes.

Design Considerations

- But, what if the risk of changing existing code was high?
- Moreover, does it make sense to include an XML export function in a node class, whose primary purpose is to work with geo-data?
- What if the export format was to change from XML to something else (e.g., JSON)
- So, need a way to add new behaviour without changing the existing classes

Visitor Pattern

Motivation

- Need a pattern where one class/interface (Visitor) defines a computation/operation and another (Visitable) is responsible for providing data access
- Need a way to add new behaviour to a family of classes without modifying the existing classes

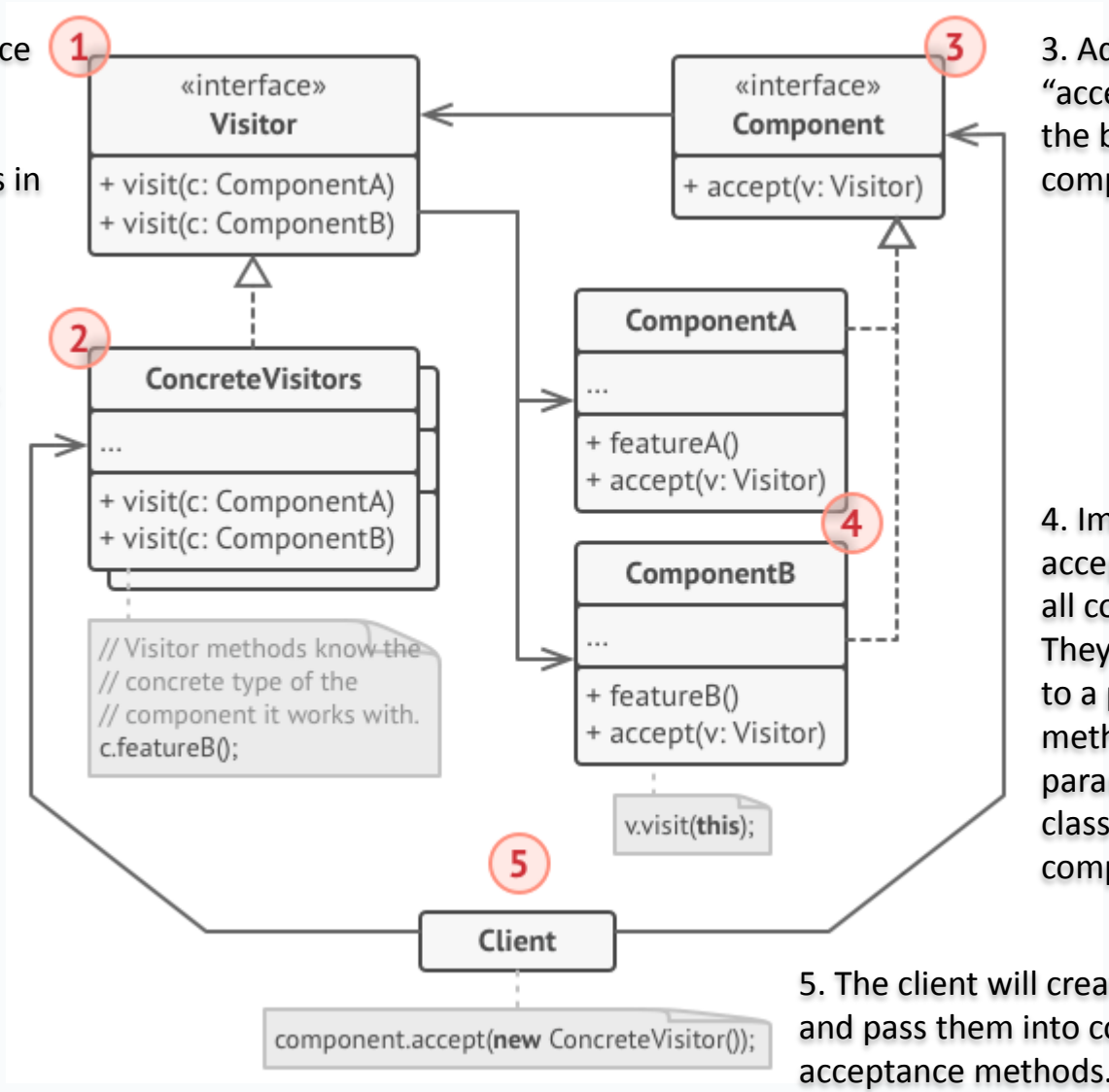
Intent

- A behavioural design pattern that lets you define a new operation without changing the classes of the objects on which it operates
- Places new behaviour into a separate class, instead of integrating into existing classes.
- Objects related to the behaviour, will not be calling it by themselves. They will be passed as arguments to methods of the visitor object instead.

Visitor Pattern

1. Create the *Visitor* interface and declare a "visiting" method for each concrete component class that exists in the program

2. For each new behaviour, create a new *Concrete Visitor* class and implement all of the visiting methods



3. Add an abstract "acceptance" method to the base class of the component hierarchy.

4. Implement the acceptance methods in all concrete components. They must redirect calls to a particular visitor's method that has a parameter of the same class as the current component.

5. The client will create visitor objects, and pass them into components via acceptance methods.

The component hierarchy should only be aware of the *Visitor* interface. On the other hand, visitors will be coupled to all concrete components.

Visitor Pattern: Pros

Pros

- Simplifies adding new operations over complex hierarchy of objects
- Helps your code adhere to the *SRP* When you need to be able to run several unrelated behaviours over a complex object structure, but you do not want to "clog" the structure's classes with the code of these behaviours.
- helps your code adhere to the *SRP* as the component class (Visitable) is just responsible for holding data, while computation or any additional behaviour is moved externally into a separate Visitor class
- A visitor can accumulate state over the course of working with an object structure

Visitor Pattern: Cons

Cons

- The pattern is not justified if a hierarchy of components changes often, violates OCP
- Every time a new component type is added, the Visitor interface needs to be changed to accommodate this new data type
- Violates encapsulation of components

Visitor is not a very common pattern because of its complexity and narrow applicability.

Hence, use the Visitor pattern with caution…

Grouping Design Patterns: Creational Patterns (1)

Involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate

- Factory Method: Defines an interface for creating an object, but lets sub-classes decide which concrete class to instantiate
- Abstract Factory: Provides an interface to create familes of related objects without specifying their concrete classes
- Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations
- Singleton: Ensures a class has only one instance

Grouping Design Patterns: Structural Patterns(2)

Enables composition of objects into larger structures, providing a way to build simple and efficient class hierarchies

- Composite: Compose objects into tree structures to represent part-whole hierarchies, enabling clients to treat individual objects and compositions of objects in a uniform way
- Decorator: Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality

Grouping Design Patterns: Behavioural Patterns(3)

Are concerned with how objects interact and identifies common communication patterns between these objects

- Command: Encapsulates a command request as an object
- Iterator: Provides a way to traverse the elements of a collection without exposing its implementation
- Observer: Allows objects to be notified when state changes
- State: Encapsulates state-based behaviours and uses delegation to alter an object's behaviour when its state changes
- Strategy: Encapsulates a family of interchangeable algorithms or behaviours and uses delegation to decide which one to use
- Template Method: Defer the exact steps of an algorithm to the sub-classes
- Visitor: Defines a new operation to a class without changing it

Design Toolbox

OO Basics

- *Abstraction*
- *Encapsulation*
- *Inheritance*
- *Polymorphism*

OO Design Principles

- *Principle of least knowledge – talk only to your friends*
- *Encapsulate what varies*
- *Favour composition over inheritance*
- *Program to an interface, not an implementation*
- *Classes should be open for extension and closed for modification*
- *Don't call us, we'll call you*
- *A class should have only one reason to change*
- *Strive for loosely coupled designs between objects that interact*
- *Depend on abstractions, do not depend on abstract classes*

Design Patterns

Structural Patterns

- *Composite*
- *Decorator*

Behavioural Patterns

- *Strategy*
- *State*
- *Template Method*
- *Iterator*
- *Observer*
- *Visitor*
- *Command*

Creational Patterns

- *Factory Method*
- *Abstract Factory*
- *Builder*
- *Singleton*