Behavioral Patterns

Behavioral patterns are about how problems are solved and how responsibilities are splitted between objects. They are more about communication than structure.

1. **Chain of Responsibility**
   Decouple sender of a request from its receiver by giving more than one object a chance to handle that request.
   Use when: more than one object can handle a request and that information is known in runtime.
2. **Command**
   Encapsulate a request as an object.
   Use when: you have a queue of requests to handle or you want to log them. Also when you want to have “undo” action.
3. **Interpreter**
   Interprets a sentence in a given language by using representation of a grammar in that language.
   Use when: you want to interpret given language and you can represent statements as an abstract syntax trees.
4. **Iterator**
   Provide a way to access elements of an aggregated objects sequentially without exposing how they are internally stored.
   Use when: you want to access object’s content without knowing how it is internally represented.
5. **Mediator**
   Define an object that knows how other objects interact. It promotes loose coupling by removing direct references to objects.
   Use when: a set of objects communicate in structured by complex ways.

6. **Memento**
   Capture external state of an object if there will be a need to restore it without violating encapsulation.
   Use when: you need to take a snapshot of an object.
7. **Observer**
   When one object changes state, all its dependents are notified about that fact.
   Use when: a change to one object requires changing others.
8. **State**
   Object is allow to change its behaviour when it's internal state changes. It looks like the object is changing its class.
   Use when: the object's behaviour depends on its state and its behaviour changes in run-time depends on that state.
9. **Strategy**
   It lets to algorithm to be independent from clients that use it.
   Use when: you have many classes that differ in their behaviour. Strategies allow to configure a class with one of many behaviours.
10. **Template Method**
    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.
    Use when: you have to define steps of the algorithm once and let subclasses to implement its behaviour.
11. **Visitor**
    Represent an operation to be performed on the elements of the structure. It lets you to define new operations without changing the classes of the elements.
    Use when: an object structure includes many classes and you want to perform an operations on the elements of that structure that depend on their classes.

Related Patterns

1. **Chain of responsibility**, **Command**, **Mediator**, and **Observer**, address how you can decouple senders and receivers, but with different trade-offs. **Chain of responsibility** passes a sender request along a chain of potential receivers. **Command** normally specifies a sender-receiver connection with a subclass. **Mediator** has senders and receivers reference each other indirectly. **Observer** defines a much decoupled interface that allows for multiple receivers to be configured at run-time.
2. **Chain of responsibility** can use **Command** to represent requests as objects.
3. **Chain of responsibility** is often applied in conjunction with **Composite**. There, a component's parent can act as its successor.
4. **Command** and **Memento** act as magic tokens to be passed around and invoked at a later time. In **Command**, the token represents a request; in **Memento**, it represents the internal state of an object at a particular time. Polymorphism is important to **Command**, but not to **Memento** because its interface is so narrow that a memento can only be passed as a value.
5. **Command** can use **Memento** to maintain the state required for an undo operation.
6. Macro **Command**s can be implemented with **Composite**.
7. A **Command** that must be copied before being placed on a history list acts as a **Prototype**.
8. **Interpreter** can use **State** to define parsing contexts.
9. The abstract syntax tree of **Interpreter** is a **Composite** (therefore **Iterator** and **Visitor** are also applicable).
10. Terminal symbols within **Interpreter**'s abstract syntax tree can be shared with **Flyweight**.
11. **Iterator** can traverse a **Composite**. **Visitor** can apply an operation over a **Composite**.
12. Polymorphic **Iterator**s rely on **Factory Method**s to instantiate the appropriate **Iterator** subclass.
13. **Mediator** and **Observer** are competing patterns. The difference between them is that **Observer** distributes communication by introducing "observer" and "subject" objects, whereas a **Mediator** object encapsulates the communication between other objects. It easier to make reusable **Observer**s and Subjects than to make reusable **Mediator**s.
14. On the other hand, **Mediator** can leverage **Observer** for dynamically registering colleagues and communicating with them.
15. **Mediator** is similar to **Facade** in that it abstracts functionality of existing classes. **Mediator** abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects

(i.e. it defines a multidirectional protocol). In contrast, **Facade** defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes (i.e. it defines a unidirectional protocol where it makes requests of the subsystem classes but not vice versa).

16. **Memento** is often used in conjunction with **Iterator**. An **Iterator** can use a **Memento** to capture the state of an iteration. The **Iterator** stores the **Memento** internally.
17. **State** is like **Strategy** except in its intent.
18. **Flyweight** explains when and how **State** objects can be shared.
19. **State** objects are often **Singleton**s.
20. **Strategy** lets you change the guts of an object. **Decorator** lets you change the skin.
21. **Strategy** is to algorithm as **Builder** is to creation.
22. **Strategy** has 2 different implementations, the first is similar to **State**. The difference is in binding times (**Strategy** is a bind-once pattern, whereas **State** is more dynamic).
23. **Strategy** is like **Template method** except in its granularity.
24. **Template method** uses inheritance to vary part of an algorithm. **Strategy** uses delegation to vary the entire algorithm.
25. The **Visitor** pattern is like a more powerful **Command** pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.

Code review

Embold

Sonarcloud

Cppcheck

Smallcode

Deepsource