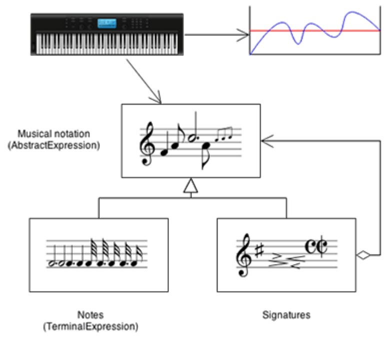
/ Design Patterns

Behavioral patterns

In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

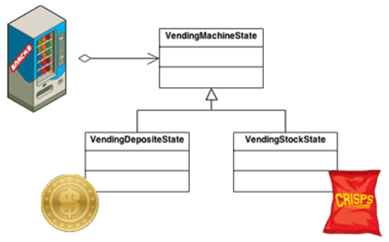

Musical notation (AbstractExpression)
Notes (TerminalExpression)
Signatures

- **Chain of responsibility**
  A way of passing a request between a chain of objects
- **Command**
  Encapsulate a command request as an object
- **Interpreter**
  A way to include language elements in a program
- **Iterator**
  Sequentially access the elements of a collection
- **Mediator**
  Defines simplified communication between classes
- **Memento**
  Capture and restore an object's internal state
- **Null Object**
  Designed to act as a default value of an object
- **Observer**
  A way of notifying change to a number of classes
- **State**
  Alter an object's behavior when its state changes

- **Strategy**
  Encapsulates an algorithm inside a class

- **Template method**
  Defer the exact steps of an algorithm to a subclass

- **Visitor**
  Defines a new operation to a class without change



Rules of thumb

1. Behavioral patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it.

2. **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 very decoupled interface that allows for multiple receivers to be configured at run-time.

3. **Chain of responsibility** can use **Command** to represent requests as objects.

4. **Chain of responsibility** is often applied in conjunction with **Composite**. There, a component's parent can act as its successor.

5. **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.

6. **Command** can use **Memento** to maintain the state required for an undo operation.

7. Macro**Command**s can be implemented with **Composite**.

8. A **Command** that must be copied before being placed on a history list acts as a **Prototype**.

9. **Interpreter** can use **State** to define parsing contexts.

10. The abstract syntax tree of **Interpreter** is a **Composite** (therefore **Iterator** and **Visitor** are also applicable).

11. Terminal symbols within **Interpreter**'s abstract syntax tree can be shared with **Flyweight**.

12. **Iterator** can traverse a **Composite**. **Visitor** can apply an operation over a **Composite**.

13. Polymorphic **Iterator**s rely on **Factory Method**s to instantiate the appropriate **Iterator** subclass.

14. **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. We've found it easier to make reusable **Observer**s and Subjects than to make reusable **Mediator**s.

15. On the other hand, **Mediator** can leverage **Observer** for dynamically registering colleagues and communicating with them.

16. **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).

17. **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.

18. **State** is like **Strategy** except in its intent.

19. **Flyweight** explains when and how **State** objects can be shared.

20. **State** objects are often **Singleton**s.

21. **Strategy** lets you change the guts of an object. **Decorator** lets you change the skin.

22. **Strategy** is to algorithm. as **Builder** is to creation.

23. **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).

24. **Strategy** objects often make good **Flyweight**s.

25. **Strategy** is like **Template method** except in its granularity.

26. **Template method** uses inheritance to vary part of an algorithm. **Strategy** uses delegation to vary the entire algorithm.

27. 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.

Read next

This article is taken from our book **Design Patterns Explained Simply**.

All of the design patterns are compiled there. The book is written in clear, simple language that makes it easy to read and understand (just like this article).

We distribute it in PDF & EPUB formats so you can get it onto your iPad, Kindle, or other portable device immediately after your purchase.

Learn more

Code examples

READ NEXT

Chain of Responsibility

RETURN

Design Patterns
AntiPatterns
Refactoring
UML

My account
Forum
Contact us
About us

Privacy policy