Chapter 09 - Chain of Responsibility Pattern

September 18, 2021

Overview

- the `Chain of Responsibility` pattern is used when we want to give a chance to multiple objects to satisfy a single request, or when we don't know in advance which object (form a chain of objects) should process a specific request

imagine a chain (linked list, tree, or any other convient data structure) of objects, and the following flow: 1. we start by sending a request to the first object in the chain 2. the object decides whether it should satisfy the request or not 3. the object forwards the request to the next object 4. the procedure is repeated untill we reach the end of the chain

- at the application level, instead of talking about cables or network nodes, we can focus on objects and the flow of a request
- the client code can send a request to the `first` processing element
- each processing element knows about its `successor` and not all processing element
- this is usually a one-way relationship, which in programming terms means a signly linked list in contrast to a doubly linked list
- a singly linked list does not allow navigation in both ways, while a doubly linked list allows that
- this chain organization is used for a good reason as it achieves decoupling between the sender (`client`) and the recievers (`processing elements`)

0.1 Real-World Examples

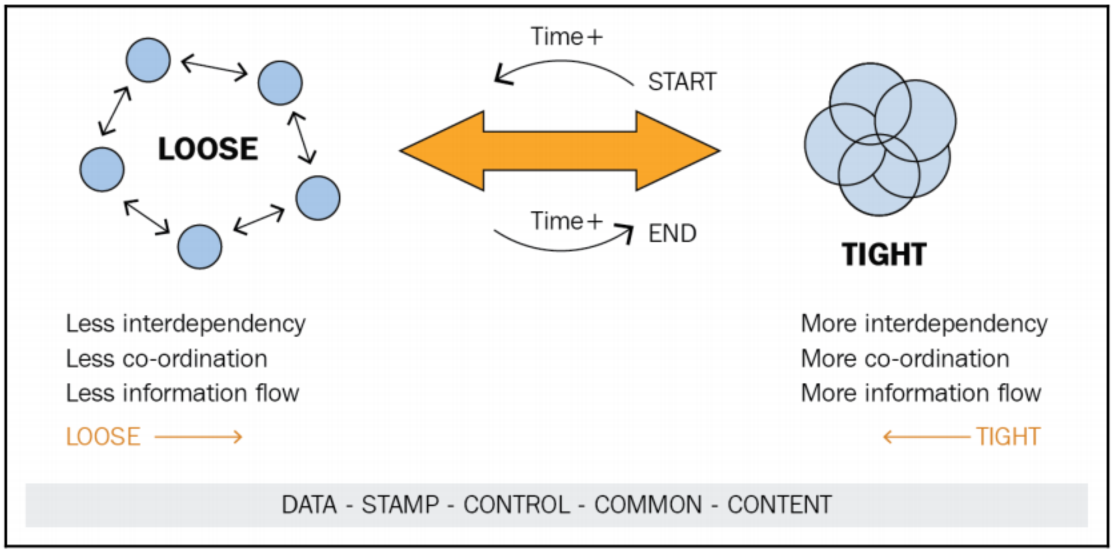
- `ATMs` and any kind of machine that accepts/returns banknotes or coins (for example, a snack-vending maching) use the `Chain of Responsibility` pattern
- there is always a single slot for all banknotes
- when a banknote is dropped, it is routed to the appropriate receptacle
- when it is returned, it is taken from the appropriate receptacle

0.2 Use cases

- by using the `Chain of Responsibility` pattern, we provide a chance to a number of different objects to satisfy a specific request
- this is useful when we dont know which object should satisfy a request in advance
- an example is a purchase system
  - there are many approval authorties
  - one approval authority might be able to order up to a certain value like 100 dollars
  - if the order is for more than 100 dollars, the order is sent to the next approval authority in the chain that can approve orders up to 200 dollars

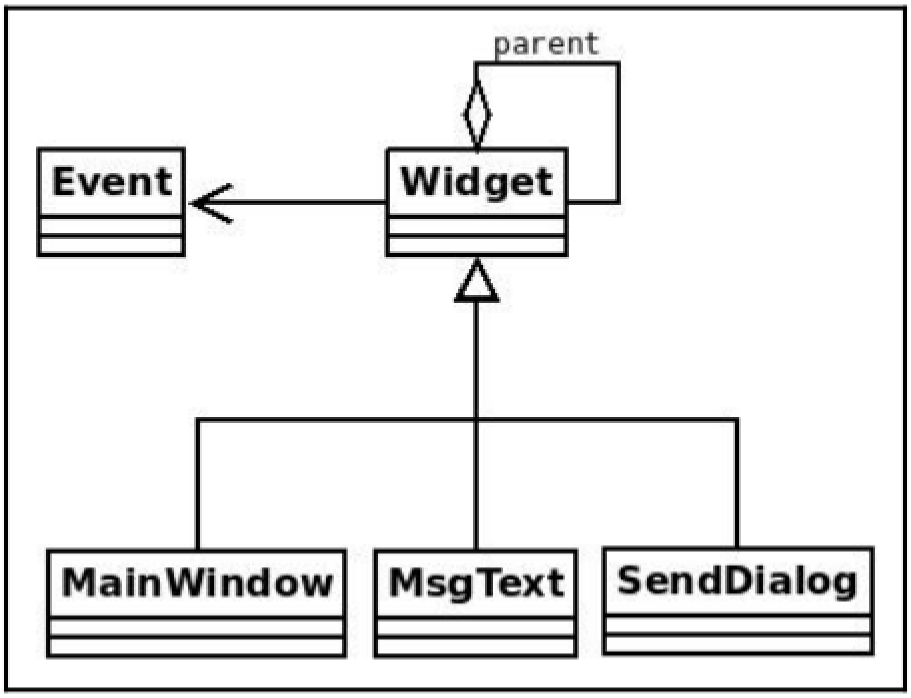
- another case where the `Chain of Responsibility` is useful is when we know that more than one object might need to process a single request
- this is what happens in event-based programming
- a signle event, such as a left-mouse click, can be caught by more than one listener

- the `Chain of Responsibility` pattern is not very useful if all the requests can be taken care of by a single processing element, unless we really dont know which element that is
- the value of this pattern is the decoupling that it offers
- instead of having many-to-many relationship with a client and all processing elements
  - and the same is true regarding the relationship between a processing element and all other processing elements
- a client only needs to know how to communicate witht he start (head) of the chain



0.3 Implementation

- we will be using `Vespe's` implementation which uses dynamic dispatching in a Pythonic style to handle requests



- the `Event` class describes an event

[1]:
```python
class Event:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name
```

- the `Widget` class is the core class of the application
- the `parent` aggregation shown in the `UML` diagram indicates that each widget can have a reference to a `parent` object, which by convention, we assume is a `Widget` instance
- note, howerver, that according to the rules of inheritance, an instance of any subclass of `Widget` (like `MsgText`) is also an instance of `Widget`
- the default value of `parent` is `None`

- the `handle()` method uses dynamic dispatching through `hasattr()` and `getattr()` to decide who is handler of a specific request (event)
- if the widget that is asked to handle and event does not support it, there are two fallback mechanisms

[3]:
```python
class Widget:
    def __init__(self, parent=None):
        self.parent = parent

    def handle(self, event):
        handler = f'handle_{event}'
        if hasattr(self, handler):
            method = getattr(self, handler)
            method(event)
        elif self.parent is not None:
            self.parent.handle(event)
        elif hasattr(self, 'handle_default'):
            self.handle_default(event)
```

- on a note, you might be thinking why the `Widget` and `Event` classes are only associated (no aggregation or composition relationship) in the diagram above
- the association is used to show that the `Widget` class knows about the `Event` class but does not have any strict references to it, since an event needs to be passed only as a parameter to `handle()`

- `MainWindow`, `MsgText`, and `SendDialog` are all widgets with different behaviors
- not all these three widgets are expected to be able to handle the same events, and even if they can handle the same event, they might behave differently

- `MainWindow` can handle only the close and default events

[4]:
```
class MainWindow(Widget):
    def handle_close(self, event):
        print(f'MainWindow: {event}')

    def handle_default(self, event):
        print(f'MainWindow Default: {event}')
```

- `SendDialog` can also handle only the paint event

[5]:
```
class SendDialog(Widget):
    def handle_paint(self, event):
        print(f'SendDialog: {event}')
```

- Finally `MsgText` can handle only the down event

[6]:
```
class MsgText(Widget):
    def handle_down(self, event):
        print(f'MsgText: {event}')
```

- the `main()` function shows how we can create a few widgets and events, and how the widget react to those events
- all events are sent to all the widgets
- note the parent relationship of each widget
- the `sd` object (and instance of `SendDialog`) has as its parent the `mw` object (an instance of `MainWindow`)
- howerver, not all objects need to have a parent that is an instance of `MainWindow`
- for example, the `msg` object (an instance of `MsgText`) has the `sd` object as a parent

[8]:
```python
def main():
    mw = MainWindow()
    sd = SendDialog(mw)
    msg = MsgText(sd)

    for e in ('down', 'paint', 'unhandled', 'close'):
        evt = Event(e)
        print(f'Sending event -{evt}- to MainWindow')
        mw.handle(evt)
        print(f'Sending event -{evt}- to SendDialog')
        sd.handle(evt)
        print(f'Sending event -{evt}- to MsgText')
        msg.handle(evt)

main()
```

```
Sending event -down- to MainWindow
MainWindow Default: down
Sending event -down- to SendDialog
MainWindow Default: down
Sending event -down- to MsgText
MsgText: down
Sending event -paint- to MainWindow
MainWindow Default: paint
Sending event -paint- to SendDialog
SendDialog: paint
Sending event -paint- to MsgText
SendDialog: paint
Sending event -unhandled- to MainWindow
MainWindow Default: unhandled
Sending event -unhandled- to SendDialog
MainWindow Default: unhandled
Sending event -unhandled- to MsgText
MainWindow Default: unhandled
Sending event -close- to MainWindow
MainWindow: close
Sending event -close- to SendDialog
MainWindow: close
Sending event -close- to MsgText
MainWindow: close
```