

04834580 Software Engineering (Honor Track) 2025-26

Behavioral Design Patterns

Sergey Mechtaev

mechtaev@pku.edu.cn

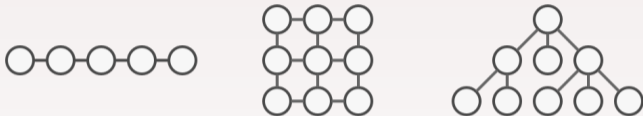
School of Computer Science, Peking University



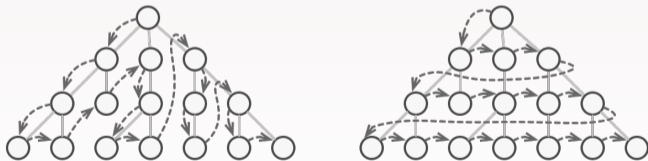
Definition ([1])

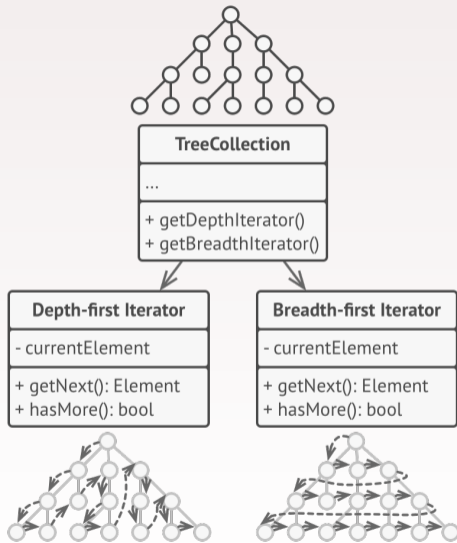
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

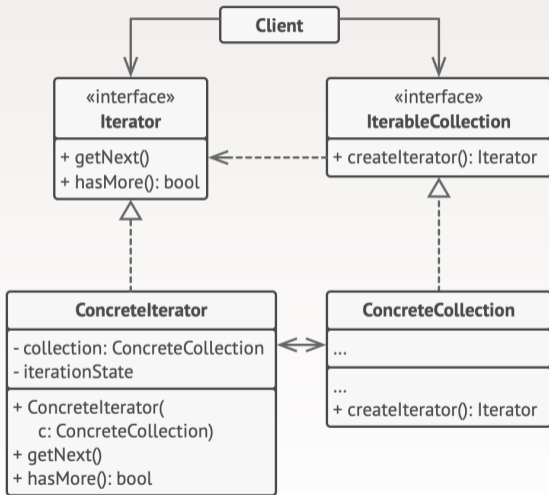
There are various types of collections:

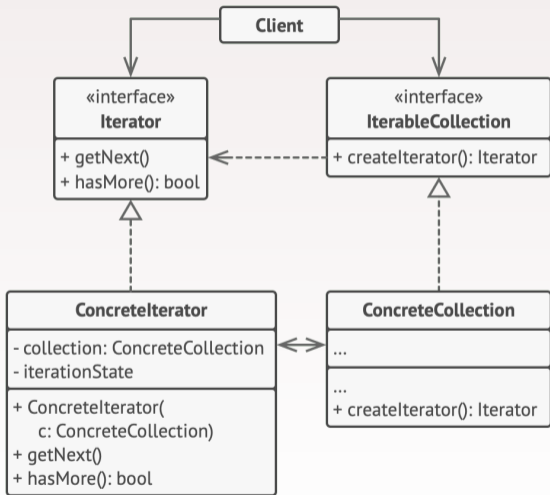


The same collection can be traversed in several different ways.









Exceptions are used instead of hasNext():

```
class Iterator:
    def __next__(self):
        if self.has_more_elements():
            return self.next_element()
        raise StopIteration

it = Iterator()
elem = next(it) # or elem = next(it, default)
```

An iterator is obtained from an iterable using `iter()`:

```
class Iterable:
    def __iter__(self):
        return Iterator()
```

```
x = Iterable()
it = iter(x)
```

Notable iterables:

- ▶ lists, dicts, sets;
- ▶ strings are iterables over characters;
- ▶ `range(end)`, `range(begin, end)`;
- ▶ `open('a.txt')` is iterable over lines if the file.

Iterators are iterables:

```
class Iterator:
    def __next__(self):
        ...

    def __iter__(self):
        return self
```

For loops are implemented using iterators:

```
for x in xs:  
    body
```

```
it = xs.__iter__()  
while True:  
    try:  
        x = it.__next__()  
    except StopIteration:  
        break  
  
    body
```

Generator functions behave like iterators:

```
def fibonacci():  
    """Generator for the Fibonacci sequence."""  
    a, b = 0, 1  
    while True:  
        yield a  
        a, b = b, a + b  
  
# Using the generator  
fib_gen = fibonacci()  
  
# Print the first 10 Fibonacci numbers  
for _ in range(10):  
    print(next(fib_gen))
```

Representing infinite sequences:

```
from itertools import islice, count

# This generates an infinite sequence of integers starting from 0.
infinite_sequence = count(start=0)

# Use map() to apply a transformation.
mapped_sequence = map(lambda x: x * x, infinite_sequence)

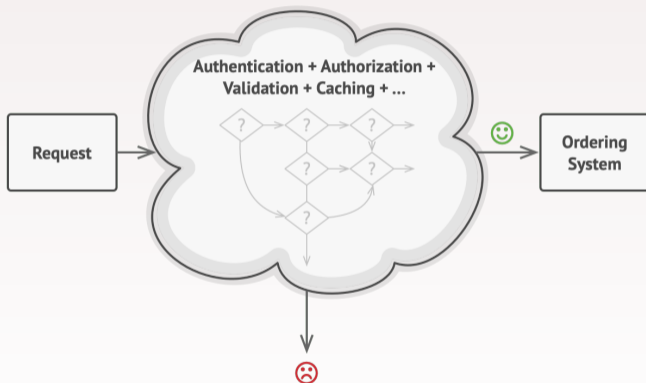
# Use islice() to slice the sequence and take only the nth element.
n = 5
nth_element = next(islice(mapped_sequence, n, n + 1))

print(f"The {n}th element of the mapped sequence is: {nth_element}")
```

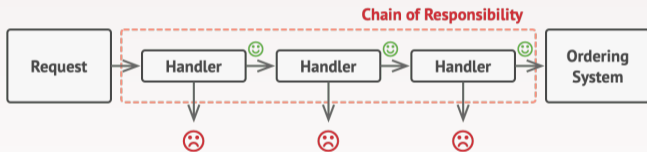
Definition ([1])

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

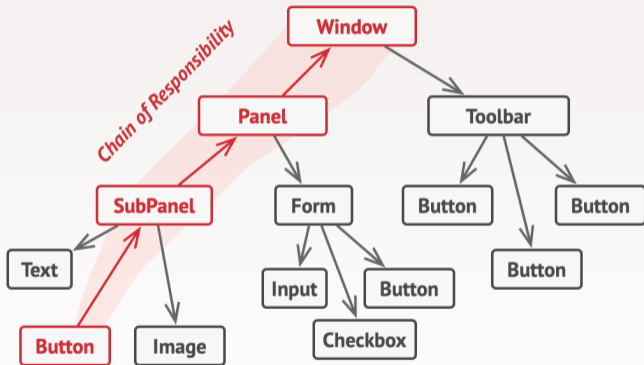
Complex logic of checks might make the system hard to maintain and extend:



CoR transforms particular behaviors into stand-alone objects called handlers, links these handlers into a chain:



Dealing with events in GUI with chain of responsibility in a tree:



The default chaining behavior can be implemented inside a base handler:

```
class AbstractHandler:
    _next_handler: Handler = None

    def set_next(self, handler: Handler) -> Handler:
        self._next_handler = handler
        return handler

    @abstractmethod
    def handle(self, request: Any) -> str:
        if self._next_handler:
            return self._next_handler.handle(request)
        return None
```

Concrete handlers:

```
class MonkeyHandler(AbstractHandler):
    def handle(self, request: Any) -> str:
        if request == "Banana":
            return f"Monkey: I'll eat the {request}"
        else:
            return super().handle(request)

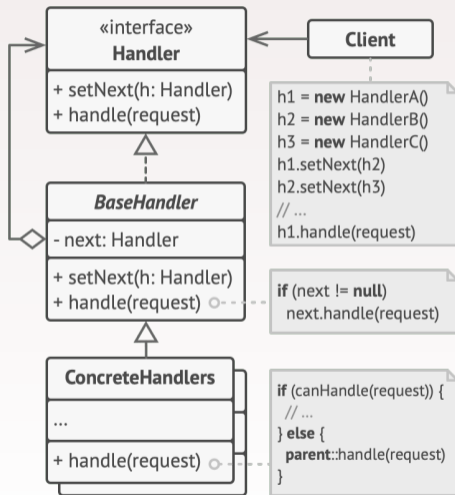
class SquirrelHandler(AbstractHandler):
    def handle(self, request: Any) -> str:
        if request == "Nut":
            return f"Squirrel: I'll eat the {request}"
        else:
            return super().handle(request)
```

Chaining handlers in client code:

```
if __name__ == "__main__":
    monkey = MonkeyHandler()
    squirrel = SquirrelHandler()
    dog = DogHandler()

    monkey.set_next(squirrel).set_next(dog)

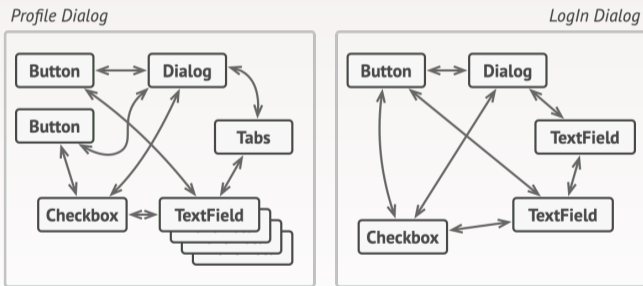
    for food in ["Nut", "Banana", "Cup of coffee"]:
        print(f"\nClient: Who wants a {food}?")
        result = handler.handle(food)
        if result:
            print(f" {result}", end="")
        else:
            print(f" {food} was left untouched.", end="")
```



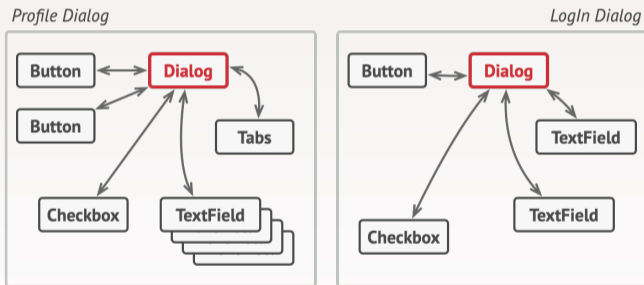
Definition ([1])

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

By having the logic of interactions implemented directly inside the code of the form elements you make these elements' classes much harder to reuse in other forms of the app:



Cease all direct communication between the components; these components must collaborate indirectly:



The mediator interface declares a method used by components to notify the mediator about various events:

```
class ConcreteMediator(Mediator):
    def __init__(self, component1: Component1, component2: Component2) -> None:
        self._component1 = component1
        self._component1.mediator = self
        self._component2 = component2
        self._component2.mediator = self

    def notify(self, sender: object, event: str) -> None:
        if event == "A":
            print("Mediator reacts on A and triggers following operations:")
            self._component2.do_c()
        elif event == "D":
            print("Mediator reacts on D and triggers following operations:")
            self._component1.do_b()
            self._component2.do_c()
```

The Base Component provides the basic functionality of storing a mediator's instance inside component objects:

```
class BaseComponent:
    def __init__(self, mediator: Mediator = None) -> None:
        self._mediator = mediator

    @property
    def mediator(self) -> Mediator:
        return self._mediator

    @mediator.setter
    def mediator(self, mediator: Mediator) -> None:
        self._mediator = mediator
```

Concrete Components implement various functionality, independent on other components or concrete mediators:

```
class Component1(BaseComponent):
    def do_a(self) -> None:
        print("Component 1 does A.")
        self.mediator.notify(self, "A")

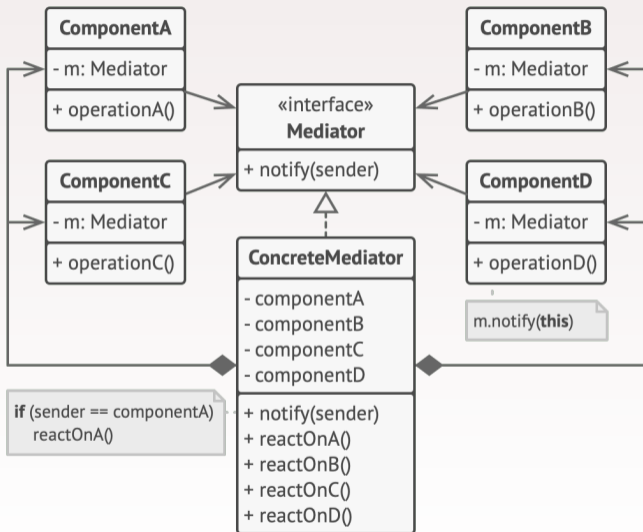
    def do_b(self) -> None:
        print("Component 1 does B.")
        self.mediator.notify(self, "B")

class Component2(BaseComponent):
    def do_c(self) -> None:
        print("Component 2 does C.")
        self.mediator.notify(self, "C")

    def do_d(self) -> None:
        print("Component 2 does D.")
        self.mediator.notify(self, "D")
```

Using mediator in client code:

```
if __name__ == "__main__":  
    # The client code.  
    c1 = Component1()  
    c2 = Component2()  
    mediator = ConcreteMediator(c1, c2)  
  
    print("Client triggers operation A.")  
    c1.do_a()  
  
    print("\n", end="")  
  
    print("Client triggers operation D.")  
    c2.do_d()
```



Definition ([1])

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

```
public class EventManager {
    Map<String, List<EventListener>> listeners = new HashMap<>();

    public EventManager(String... operations) {
        for (String operation : operations) {
            this.listeners.put(operation, new ArrayList<>());
        }
    }

    public void subscribe(String eventType, EventListener listener) {
        List<EventListener> users = listeners.get(eventType);
        users.add(listener);
    }

    public void unsubscribe(String eventType, EventListener listener) {
        List<EventListener> users = listeners.get(eventType);
        users.remove(listener);
    }

    public void notify(String eventType, File file) {
        List<EventListener> users = listeners.get(eventType);
        for (EventListener listener : users) {
            listener.update(eventType, file);
        }
    }
}
```

```
public class Editor {
    public EventManager events;
    private File file;

    public Editor() {
        this.events = new EventManager("open", "save");
    }

    public void openFile(String filePath) {
        this.file = new File(filePath);
        events.notify("open", file);
    }

    public void saveFile() throws Exception {
        if (this.file != null) {
            events.notify("save", file);
        } else {
            throw new Exception("Please open a file first.");
        }
    }
}
```

```
public interface EventListener {
    void update(String eventType, File file);
}

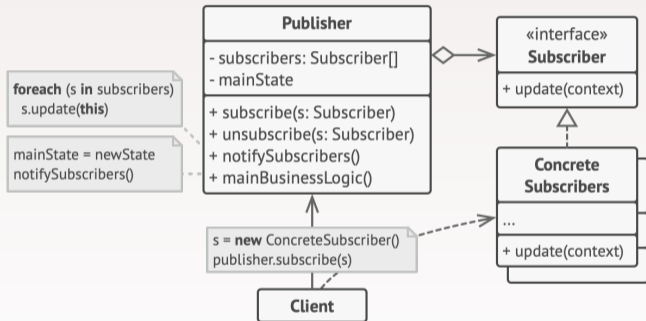
public class EmailNotificationListener implements EventListener {
    private String email;

    public EmailNotificationListener(String email) {
        this.email = email;
    }

    @Override
    public void update(String eventType, File file) {
        System.out.println("Email to " + email + ": Someone has performed " +
            eventType + " operation with the following file: " + file.getName());
    }
}
```

```
public class Demo {
    public static void main(String[] args) {
        Editor editor = new Editor();
        editor.events.subscribe("open", new LogOpenListener("/path/to/log/file.txt"));
        editor.events.subscribe("save",
            new EmailNotificationListener("admin@example.com"));

        try {
            editor.openFile("test.txt");
            editor.saveFile();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Definition ([1])

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

The Context defines the interface of interest to clients:

```
class Context():
    def __init__(self, strategy: Strategy) -> None:
        self._strategy = strategy

    @property
    def strategy(self) -> Strategy:
        return self._strategy

    @strategy.setter
    def strategy(self, strategy: Strategy) -> None:
        self._strategy = strategy

    def do_some_business_logic(self) -> None:
        print("Context: Sorting data using the strategy (not sure how it'll do it)")
        result = self._strategy.do_algorithm(["a", "b", "c", "d", "e"])
        # ...
```

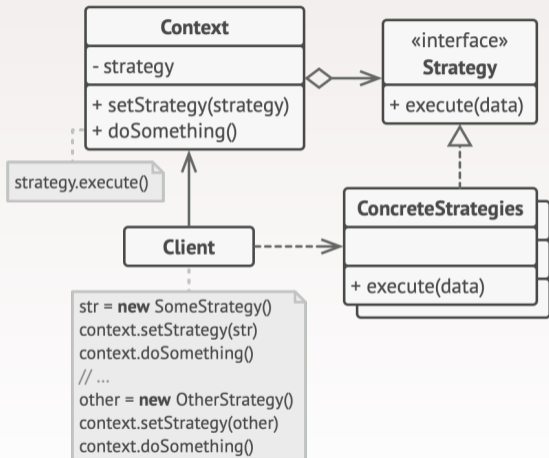
Concrete strategies:

```
class ConcreteStrategyA(Strategy):  
    def do_algorithm(self, data: List) -> List:  
        return sorted(data)  
  
class ConcreteStrategyB(Strategy):  
    def do_algorithm(self, data: List) -> List:  
        return reversed(sorted(data))
```

Using strategies in client code:

```
if __name__ == "__main__":
    context = Context(ConcreteStrategyA())
    print("Client: Strategy is set to normal sorting.")
    context.do_some_business_logic()
    print()

    print("Client: Strategy is set to reverse sorting.")
    context.strategy = ConcreteStrategyB()
    context.do_some_business_logic()
```



- [1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design patterns: elements of reusable object-oriented software.
Pearson Deutschland GmbH, 1995.