Software Design
Behavioral Design Patterns

Dr. Samuel Cho, PhD

NKU

Behavioral Patterns

Chain of Responsibility

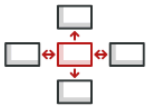
Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

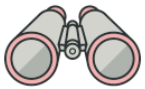


Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

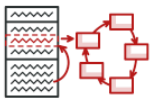
Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.



Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Visitor

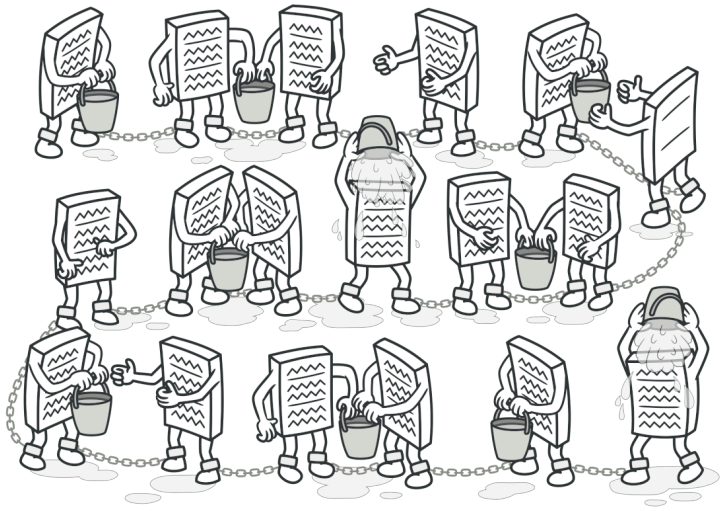
Lets you separate algorithms from the objects on which they operate.

Chain of Responsibility

CoR in Real World

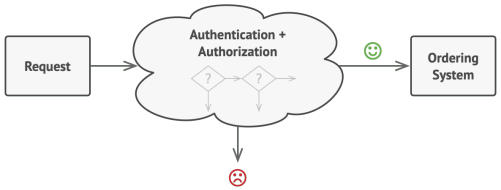
- ATM uses the Chain of Responsibility in money giving mechanism.
- When ATM withdraw money, it chooses the largest paper bills, then the next ones, and on, until it gives all the cache we request.

Chain of Responsibility Design Pattern



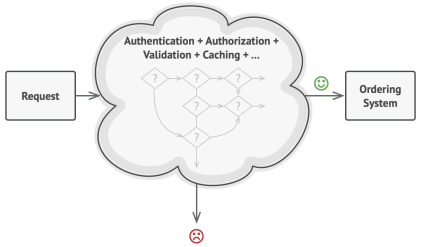
- Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Problem



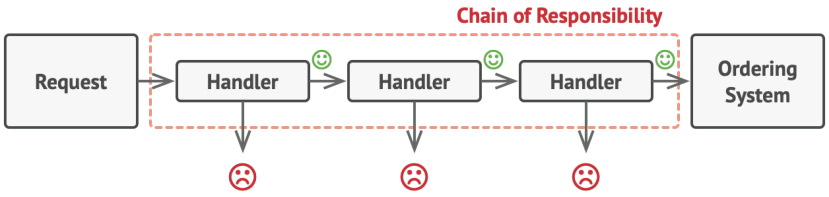
- Imagine that you're working on an online ordering system.
- You want to check and restrict access to the system so only authenticated users can create orders.
- Also, users who have administrative permissions must have full access to all orders.
- We have requests on how to check each customer over and over again.

Problem



- The code became increasingly bloated as you added each new feature.
- Changing one check sometimes affected the others.
- Worst of all, when you tried to reuse the checks to protect other components of the system, you had to duplicate some of the code since those components required some of the checks, but not all of them.

Solution



- the Chain of Responsibility relies on transforming particular behaviors into stand-alone objects called handlers.
- In our case, each check should be extracted to its class with a single method that performs the check.

Code

- Let's say we handle a request.
- We can process it with the base handler when it does not have the (next) handler that it passes the request.
- We have two handlers, when the request starts with "A", HandlerA processes the request, and with "B", Handler B processes it.

Interface and Implementation

- We have the Handler interface with two methods: set_next and handle.

Code #

```
# Interface
class Handler(object):
  def set_next(self, handler): pass
  def handle(self, request): pass
```

- BaseHandler processes the request only when it does not have the next handler.

Code #

```
# Implementations
class BaseHandler(Handler):
  def __init__(self): self.next = None
  def set_next(self, handler):
    self.next = handler
  def handle(self, request):
    if self.next is not None:
      self.next.handle(request)
    else: # no next handler
      print(f"Base handler is processing {request}")
```

Code #

```
class HandlerA(BaseHandler):
  def set_next(self, handler): self.next = handler
  def handle(self, request):
    if self.can_handle(request):
      print("HandleA can handle it")
    else:
      print("Hand over to the next handler")
      self.next.handle(request)
  def can_handle(self, request):
    if request.startswith("A"): return True
    else: return False

class HandlerB(BaseHandler):
  def set_next(self, handler): self.next = handler
  def handle(self, request):
    if self.can_handle(request):
      print(f"HandleB can handle {request}")
    else:
      print("Hand over to the next handler")
      self.next.handle(request)
  def can_handle(self, request):
    if request.startswith("B"): return True
    else: return False
```

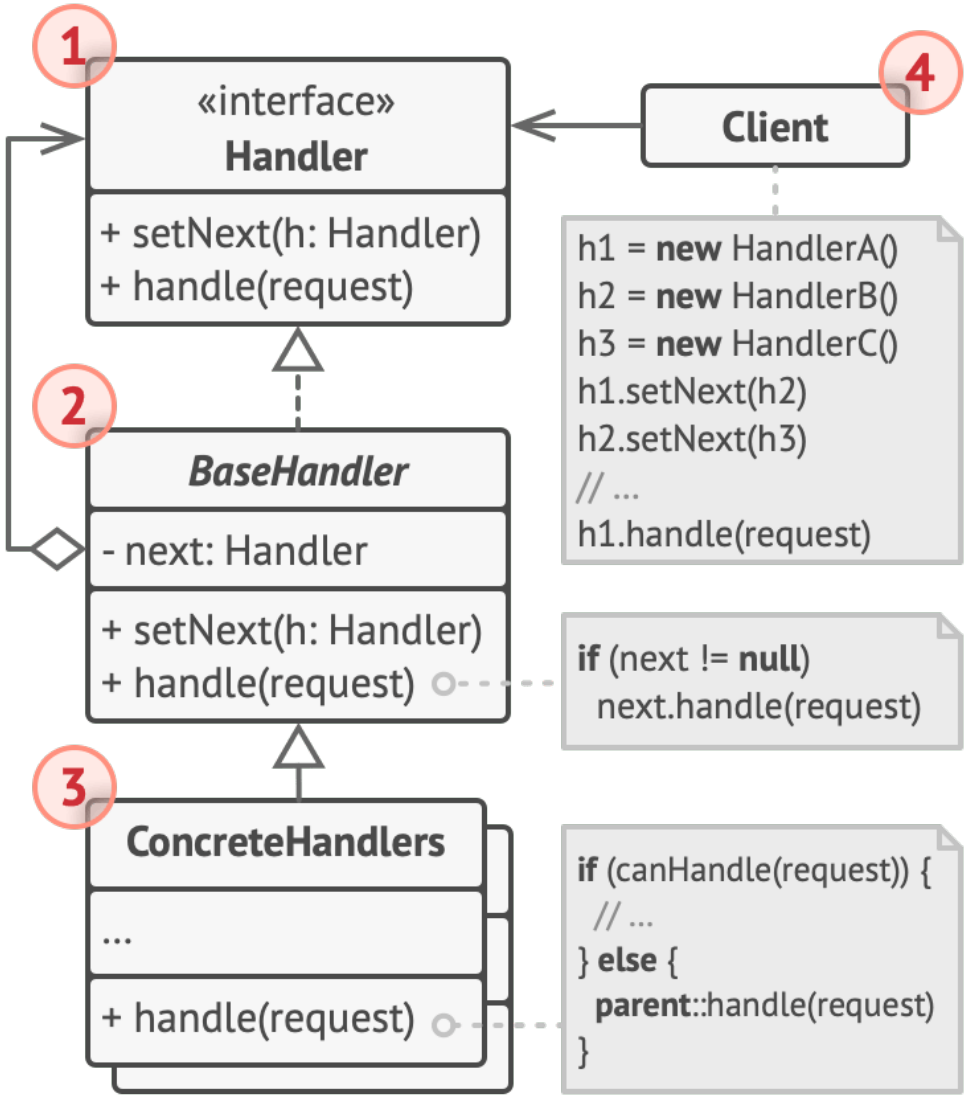
Driver

Code #

```
1  h = BaseHandler()
2  h.handle("Process B")
3
4  h1 = HandlerA()
5  h2 = HandlerB()
6
7  h.set_next(h1)
8  h1.set_next(h2)
9
10 h.handle("A - process C")
11 h.handle("B - process D")
```

```
Base handler is processing Process B
HandleA can handle it
Hand over to the next handler
HandleB can handle B - process D
```

Structure


1
«interface»
Handler
+ setNext(h: Handler)
+ handle(request)
4
Client
h1 = new HandlerA()
h2 = new HandlerB()
h3 = new HandlerC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
2
BaseHandler
- next: Handler
+ setNext(h: Handler)
+ handle(request)
if (next != null)
next.handle(request)
3
ConcreteHandlers
...
+ handle(request)
if (canHandle(request)) {
// ...
} else {
parent::handle(request)
}

Applicability

- Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand.
- Use the pattern when it's essential to execute several handlers in a particular order.
- Use the CoR pattern when the set of handlers and their order are supposed to change at runtime.

Pros

- You can control the order of request handling.
- Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform operations.
- Open/Closed Principle. You can introduce new handlers into the app without breaking the existing client code.

Cons

- Some requests may end up unhandled.

Relationship with Other Patterns

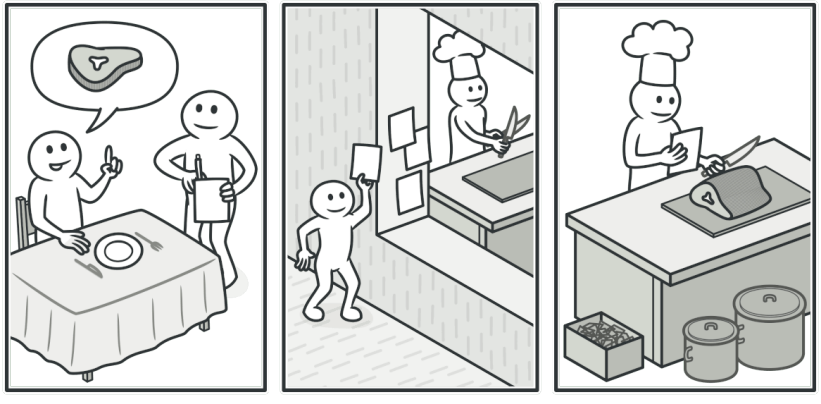
- Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests:
- Chain of Responsibility is often used in conjunction with Composite. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree.

- Handlers in Chain of Responsibility can be implemented as Commands. In this case, you can execute a lot of different operations over the same context object, represented by a request.
- Chain of Responsibility and Decorator have very similar class structures. Both patterns rely on recursive composition to pass the execution through a series of objects. However, there are several crucial differences.

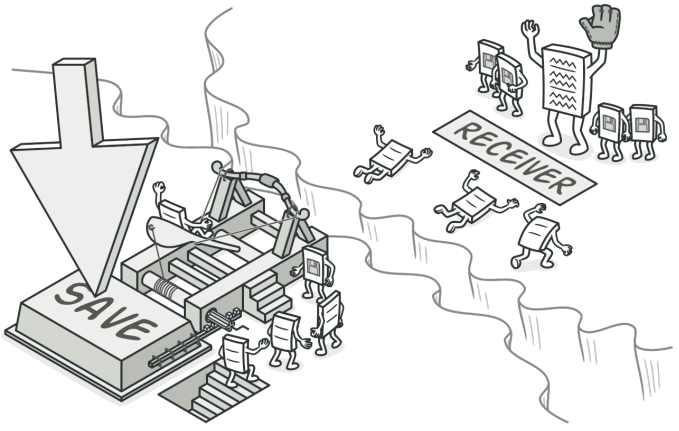
Command

Command in Real World

- When we visit a restaurant, a server takes order (command) and delivers the order to the cook to make the food we order.
- In this scenario, the invoker (server) generates a command (order) that is processed by a cook (receiver).



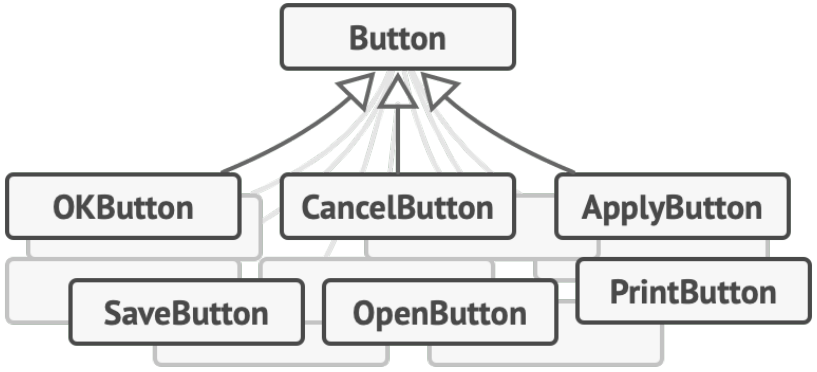
Command Design Pattern



- Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request.

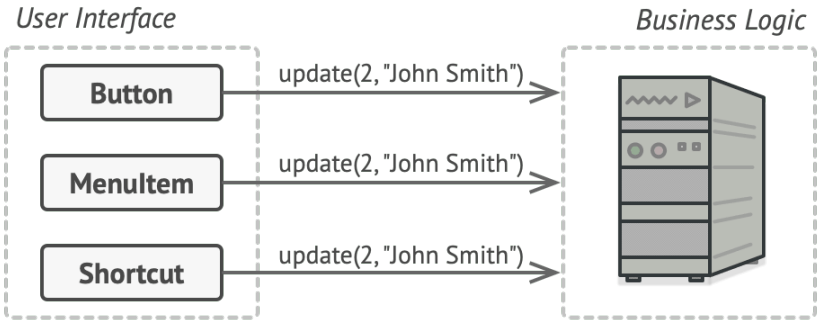
Problem

- Consider we are making a new text-editor app; we make many buttons that look similar but do different things.
- Where would you put the code for the various click handlers of these buttons?
- We decide to make subclasses to handle the requests from the buttonbs.

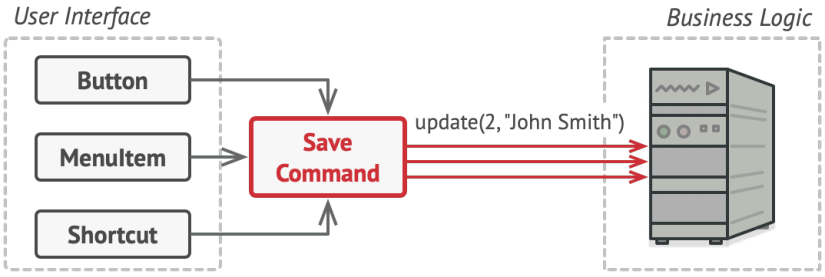


- We soon realize that this approach is flawed.
- We have an enormous number of subclasses that are depend upon the base Class.
- Also, we are in danger of duplicating code among subclasses.

Solution



- Good software design is often based on the principle of separation of concerns, which usually results in breaking an app into layers.
- In the code it might look like this: a GUI object calls a method of a business logic object, passing it some arguments. This process is usually described as one object sending another a reques



- The Command pattern suggests that GUI objects shouldn't send these requests directly.
- Instead, you should extract all of the request details, such as the object being called, the name of the method and the list of arguments into a separate command class with a single method that triggers this request.

Code

- In this example, we have interfaces Command/Invoker/Receiver to make a simple GUI editor.
- We create Button(Invoker) that creates CopyCommand or PasteCommand (both are Commands) to send it over to Editor(Receiver).
- Depending on the Command, the Editor will copy or paste the content that it receives.

Interfaces

- The Command has one method: execute().
- The Receiver does copy or paste action depending on the message it receives.

Code #

```
# interface
class Command(object):
  def execute(self): pass

class Receiver(object):
  def __str__(self):
    return "editor"
  def paste(self):
    print("Doing the paste action!")
  def copy(self):
    print("Doing the copy action!")

class Invoker(object):
  def set_command(self, command):
    self.command = command
```

- The CopyCommand and PasteCommand implements the execute method with receiver and params (detailed message) as parameters.

Code #

```
# Implementations
class CopyCommand(Command):
  def execute(self, receiver, params):
    print(f"To ({receiver}): copy {params}")
    receiver.copy()

class PasteCommand(Command):
  def execute(self, receiver, params):
    print(f"To ({receiver}): copy {params}")
    receiver.paste()

class Button(Invoker): pass
class Editor(Receiver): pass
```

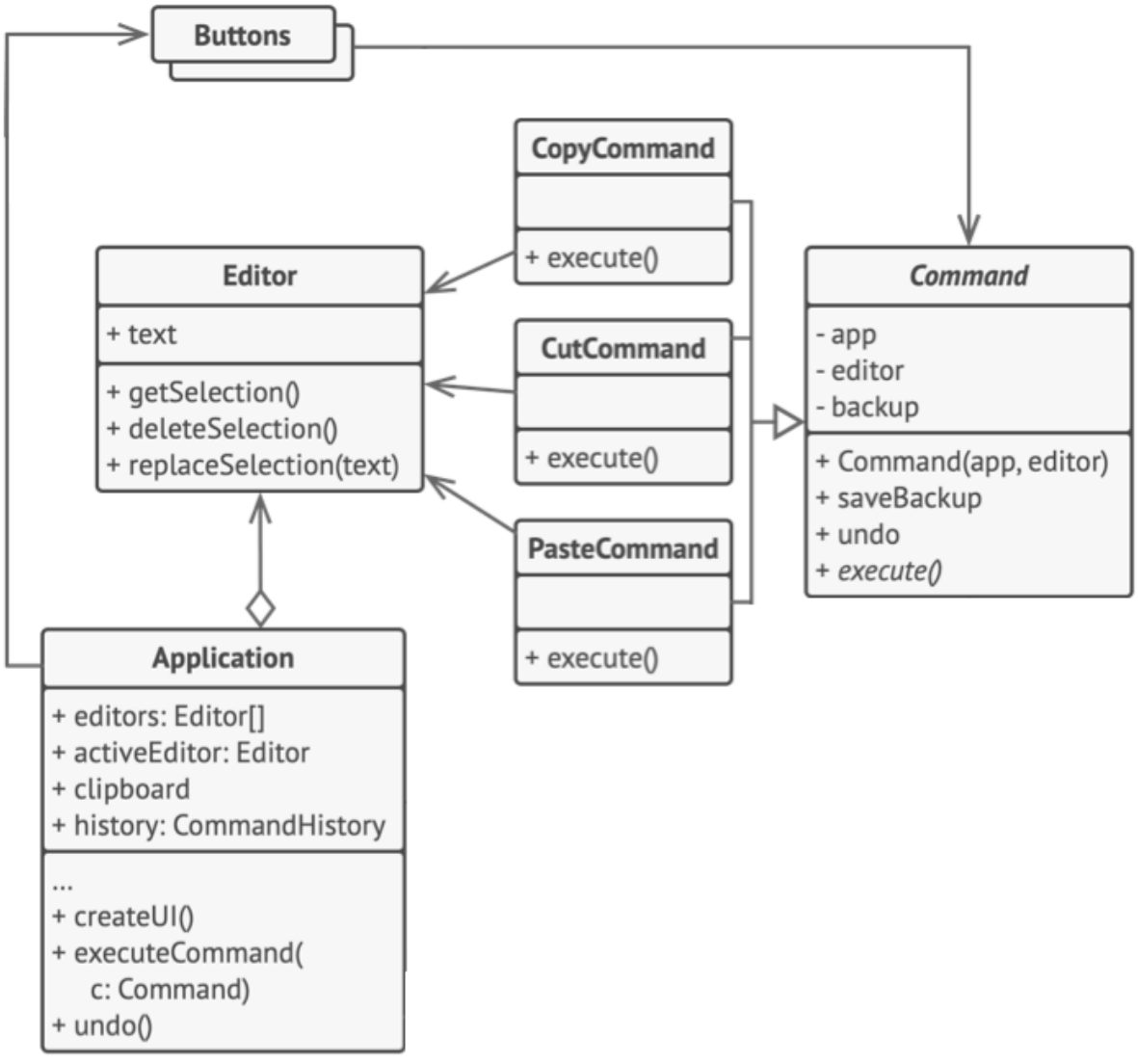
Driver

Code #

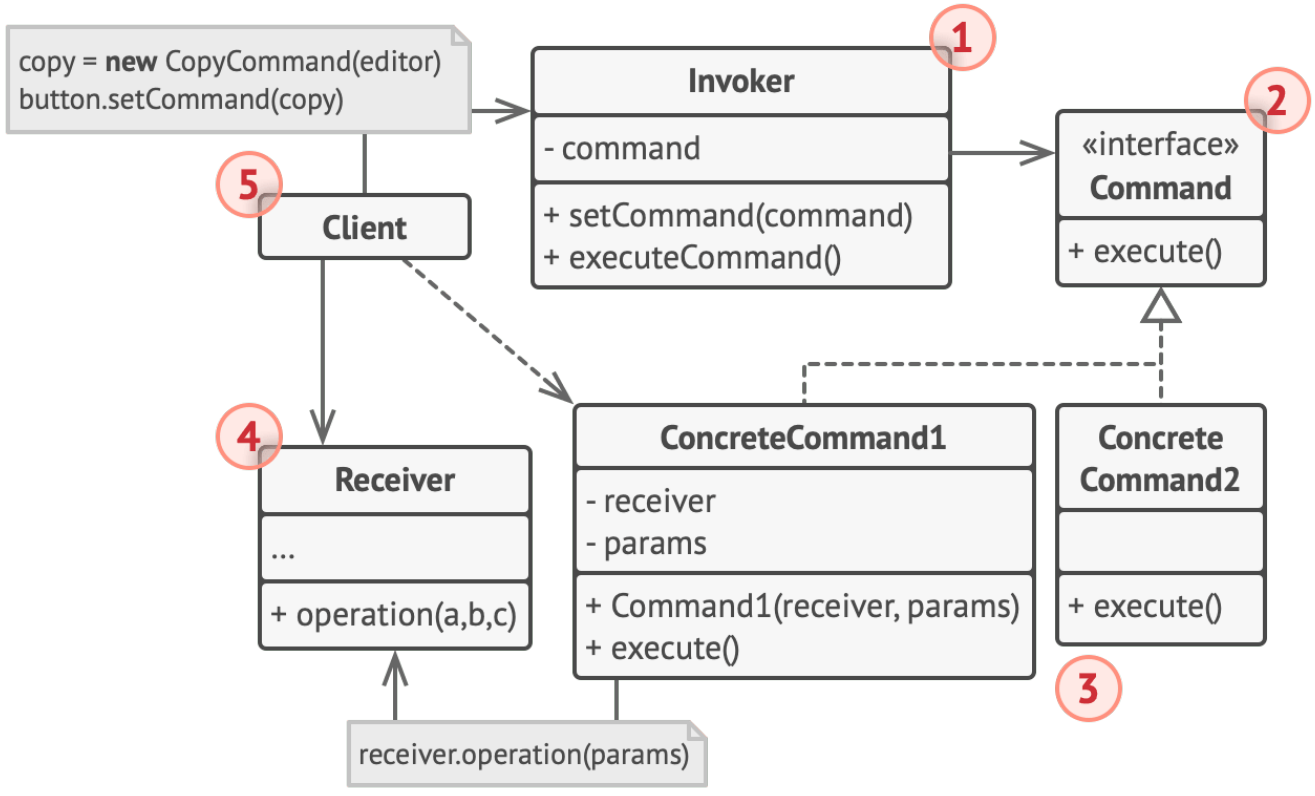
```
1  # Driver
2  ## Invoker (button) initiating a request (
       ↪ command) to a receiver (editor)
3  button1 = Button()
4  button2 = Button()
5  receiver = Editor()
6
7  copy = CopyCommand()
8  # when button is clicked, copy command is
       ↪ created
9  button1.set_command(copy)
10
11 paste = PasteCommand()
12 button2.set_command(paste)
13
14 # copy command sends a command to the
       ↪ receiver
15 copy.execute(receiver, 'line 1')
16 # paste
17 paste.execute(receiver, 'line 3')
```

```
To (editor): copy line 1
Doing the copy action!
To (editor): copy line 3
Doing the paste action!
```

Structure


Buttons
CopyCommand
+ execute()
Editor
+ text
+ getSelection()
+ deleteSelection()
+ replaceSelection(text)
CutCommand
+ execute()
Command
- app
- editor
- backup
+ Command(app, editor)
+ saveBackup
+ undo
+ execute()
PasteCommand
+ execute()
Application
+ editors: Editor[]
+ activeEditor: Editor
+ clipboard
+ history: CommandHistory
...
+ createUI()
+ executeCommand(
c: Command)
+ undo()

- This is a generalized UML for the command design pattern.



Applicability

- Use the Command pattern when you want to parametrize objects with operations.
- Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely.
- Use the Command pattern when you want to implement reversible operations, as we can send undo/redo commands.

Pros

- Use the Command pattern when you want to parametrize objects with operations.
- Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely.
- Use the Command pattern when you want to implement reversible operations.

Cons

- Single Responsibility Principle. You can decouple classes that invoke operations from classes that perform these operations.
- Open/Closed Principle. You can introduce new commands into the app without breaking existing client code.
- You can implement undo/redo.
- You can implement deferred execution of operations.
- You can assemble a set of simple commands into a complex one.

Relationship with Other Patterns

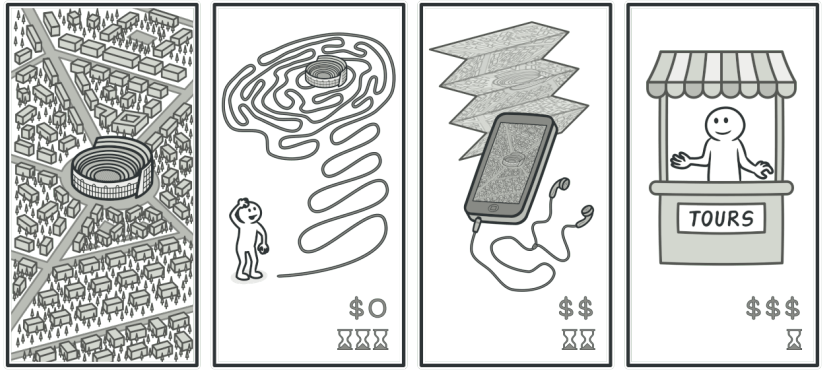
- Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests:
- Handlers in Chain of Responsibility can be implemented as Commands. In this case, you can execute a lot of different operations over the same context object, represented by a request.
- You can use Command and Memento together when implementing "undo". In this case, commands are responsible for performing various operations over a target object, while mementos save the state of that object just before a command gets executed.

- Command and Strategy may look similar because you can use both to parameterize an object with some action. However, they have very different intents.
- Prototype can help when you need to save copies of Commands into history.
- You can treat Visitor as a powerful version of the Command pattern. Its objects can execute operations over various objects of different classes.

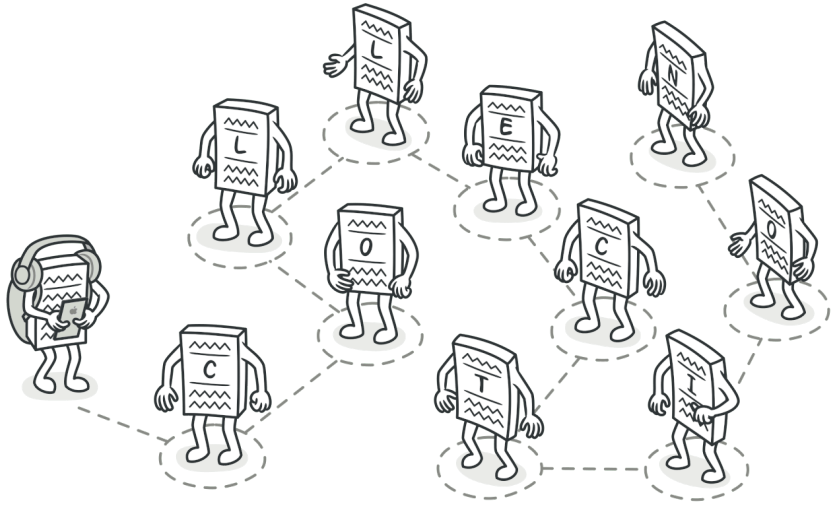
Iterator

Iterator in Real World

- We can visit Rome and visit all the main streets on foot, but possibly waste time.
- We can use Google map (street view) to enjoy the streets virtually.
- We can hire tour guide to visit any locations effectively.
- All of these options act as iterators over the vst collection of sights and attractions in Rome.

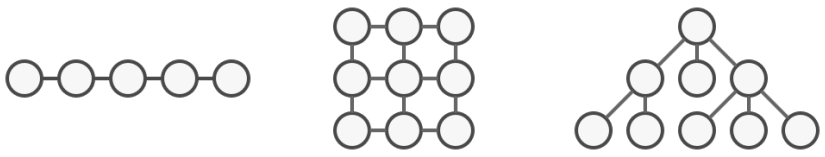


Iterator Design Pattern

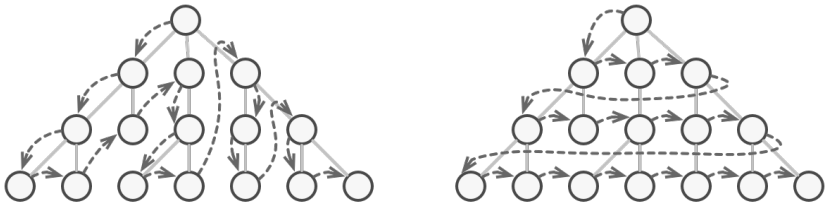


- Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Problem

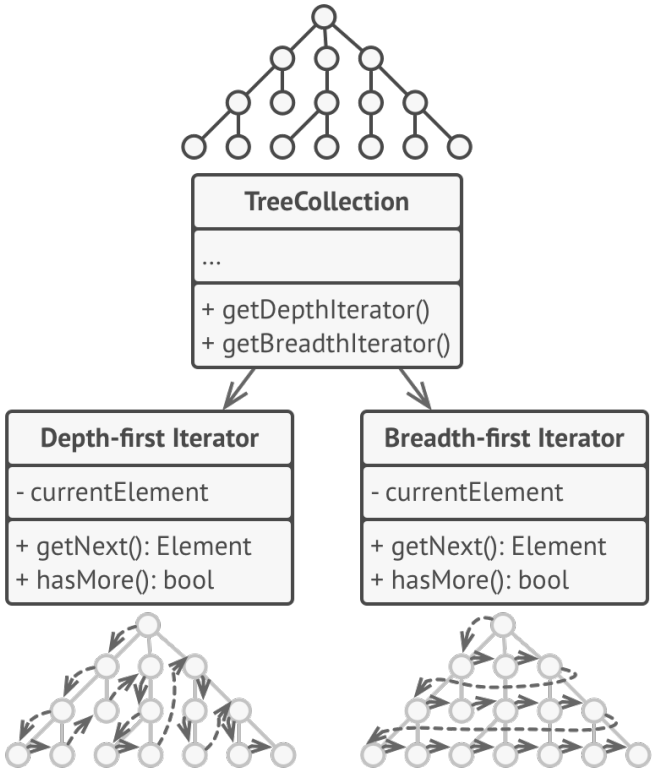


- Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.



- We have different algorithms to traverse the data structures such as BFS or DFS.
- However, the client code that's supposed to work with various collections may not even care how they store their elements.
- The inner workings should be encapsulated and isolated from the users who would use them.

Solution



- The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an iterator.

Code

- In CSC 360, we learned Java iterator that uses two methods next and hasNext to iterate over any data structures.
- In this example, we make an Iterable (Bookshelf) and its Iterator to implement a business logic that iterates over the iterable with interface methods.

Code #

```
1 ArrayList<String> cars = new ArrayList<String>();
2 cars.add("BMW"); ...
3
4 # Iterator from the Iterable
5 Iterator<String> it = cars.iterator();
6 # Driver
7 while (it.hasNext()) {
8   String res = it.Next; ...
9 }
```

Interfaces

- The iterator has only two methods next() and has_next().
- The Iterable has one method iterator().

Code #

```
# Interface
class Iterator(object):
  def next(self): pass
  def has_next(self): pass

class Iterable(object):
  def iterator(self): pass
```

- The Iterable is Bookshelf that has Book as its component.

Code #

```
# Book object
class Book(object):
  def __init__(self, name):
    self.name = name
  def __str__(self):
    return self.name

# Iterable Collection
class Bookshelf(Iterable):
  def __init__(self):
    self._books = []
  def length(self): ...
  def add(self, book): ...
  def get(self, index): ...
  def iterator(self):
    return BookshelfIterator(self)
```

- We have a BookIterator that iterates over the Bookshelf.
- Notice that the index should start with -1 as the next() method is used to access the first (index 0) element (line 6).

Code #

```
1  # Book iterator
2  class BookshelfIterator(Iterator):
3    def __init__(self, bookshelf):
4      self.bookshelf = bookshelf
5      # we always ge the next one, so the first index is -1
6      self.index = -1
7    def next(self):
8      if self.has_next():
9        self.index += 1
10       return self.bookshelf.get(self.index)
11   def has_next(self):
12     if self.index < self.bookshelf.length() - 1: return True
13     return False
```

Driver

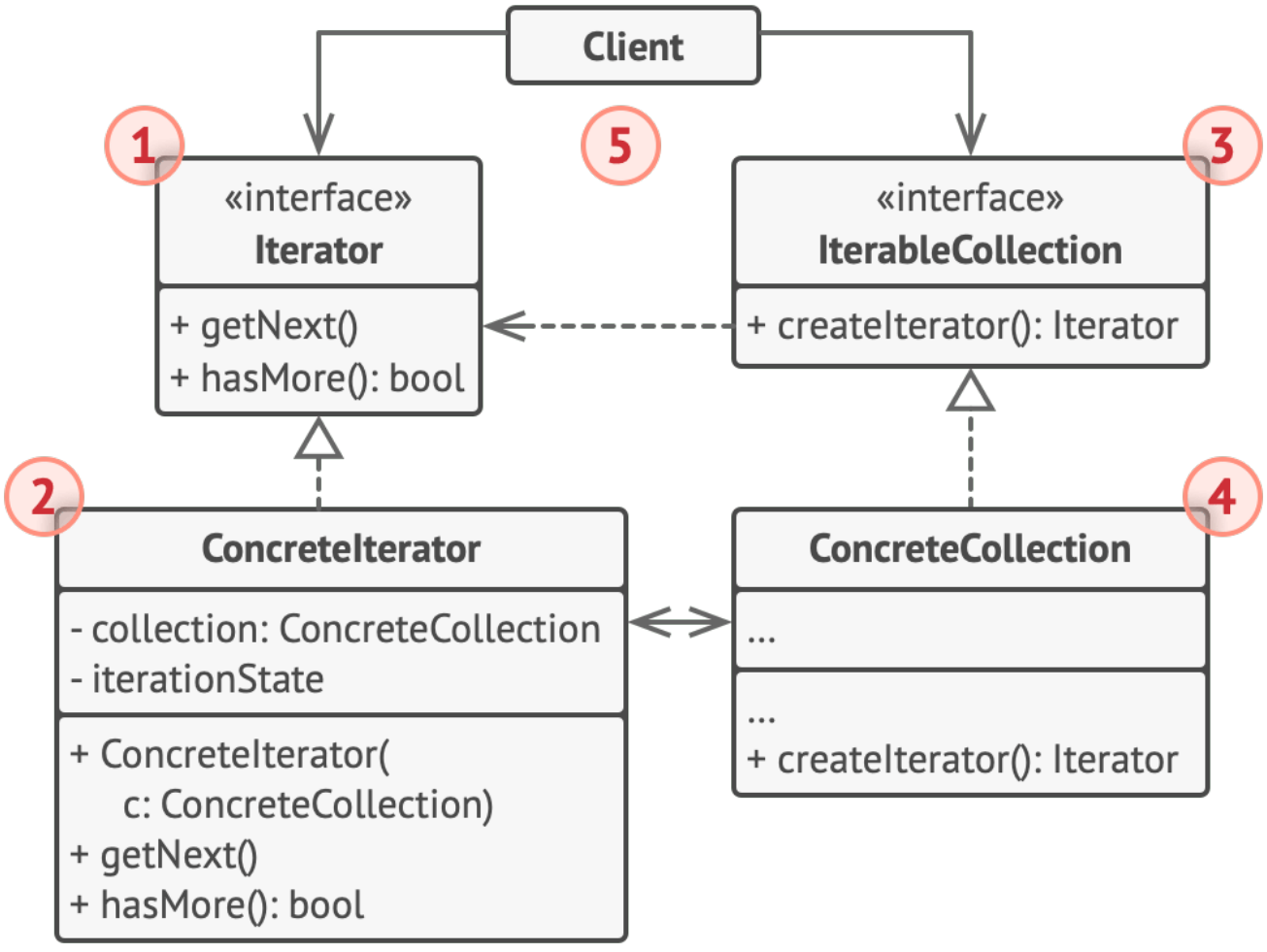
- We use the has_next() and next() method to access all the elements (lines 9 - 11).
- The iterable has the iterator information that can be accessed using the iterator() method (line 6).

Code #

```
1  # Driver
2  bookshelf = Bookshelf()
3  bookshelf.add(Book('book1'))
4  bookshelf.add(Book('book2'))
5  iterator = bookshelf.iterator()
6
7  # Iterate only two methods has_more and
       ↪ get_next
8  while iterator.has_next():
9    res = iterator.next()
10   print(res)
```

```
book1
book2
book3
```

Structure


Client
1
5
3
«interface»
Iterator
+ getNext()
+ hasMore(): bool
«interface»
IterableCollection
+ createIterator(): Iterator
2
4
ConcreteIterator
- collection: ConcreteCollection
- iterationState
+ ConcreteIterator(
c: ConcreteCollection)
+ getNext()
+ hasMore(): bool
ConcreteCollection
...
...
+ createIterator(): Iterator

Applicability

- Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons).
- Use the pattern to reduce duplication of the traversal code across your app.
- Use the Iterator when you want your code to be able to traverse different data structures or when types of these structures are unknown beforehand.

Pros

- Single Responsibility Principle. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.
- Open/Closed Principle. You can implement new types of collections and iterators and pass them to existing code without breaking anything.
- You can iterate over the same collection in parallel because each iterator object contains its own iteration state.
- For the same reason, you can delay an iteration and continue it when needed.

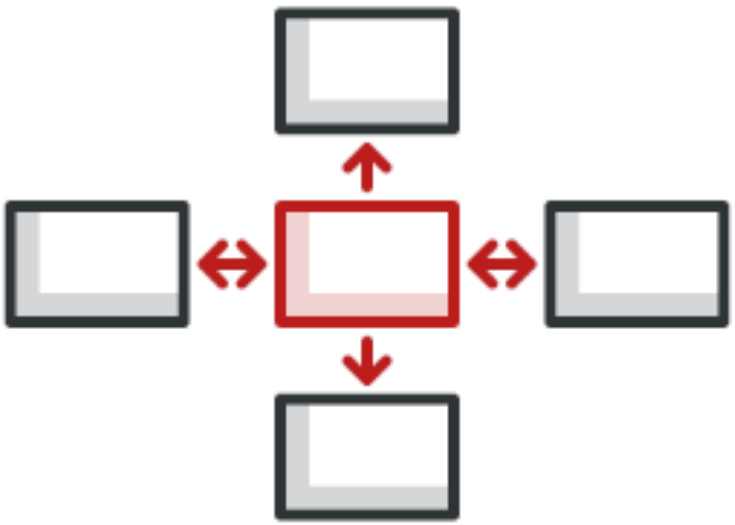
Cons

- Applying the pattern can be an overkill if your app only works with simple collections.
- Using an iterator may be less efficient than going through elements of some specialized collections directly.

Relationship with Other Patterns

- You can use Iterators to traverse Composite trees.
- You can use Factory Method along with Iterator to let collection subclasses return different types of iterators that are compatible with the collections.

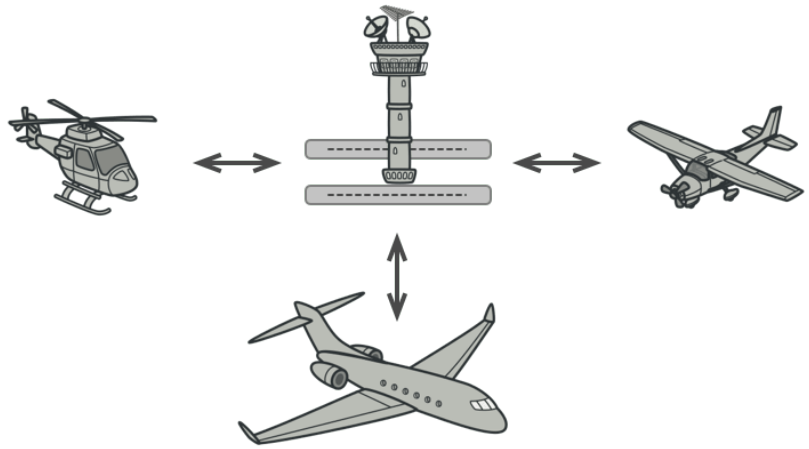
- You can use Memento along with Iterator to capture the current iteration state and roll it back if necessary.
- You can use Visitor along with Iterator to traverse a complex data structure and execute some operation over its elements, even if they all have different classes.



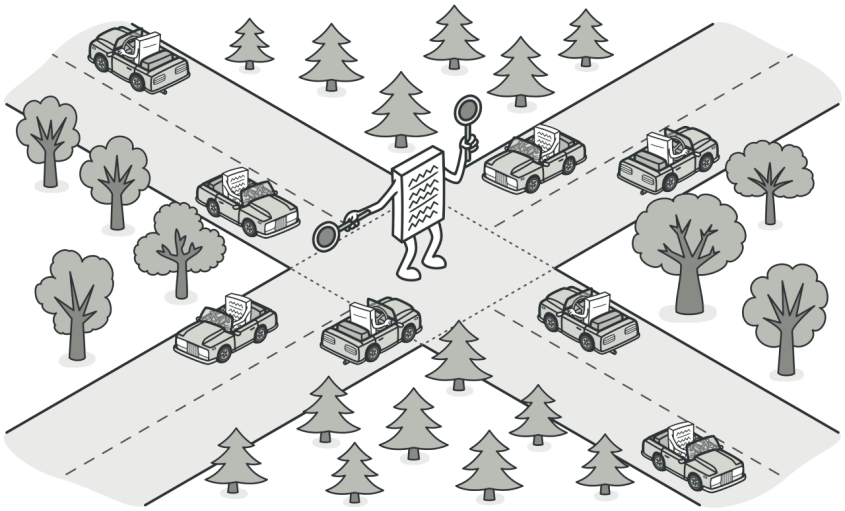
Mediator

Mediator in Real World

- Pilots of aircraft that approach or depart the airport control area don't communicate directly with each other.
- Instead, they speak to an air traffic controller, who sits in a tall tower somewhere near the airstrip.

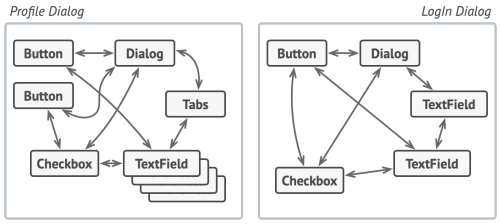


Mediator Design Pattern



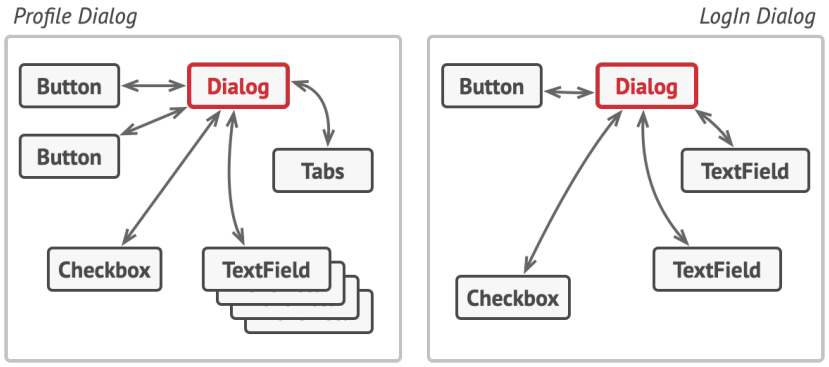
- Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

Problem



- Say you have a dialog for creating and editing customer profiles.
- It consists of various form controls such as text fields, checkboxes, buttons, etc; Some of the form elements may interact with others.
- By having this logic implemented directly inside the code of the form elements you make these elements' classes much harder to reuse in other forms of the app.

Solution



- The Mediator pattern suggests that you should cease all direct communication between the components which you want to make independent of each other.
- Instead, these components must collaborate indirectly, by calling a special mediator object that redirects the calls to appropriate components.

- As a result, the components depend only on a single mediator class instead of being coupled to dozens of their colleagues.
- This way, the Mediator pattern lets you encapsulate a complex web of relations between various objects inside a single mediator object.
- The fewer dependencies a class has, the easier it becomes to modify, extend or reuse that class.

Code

- We have the Mediator (AirtrafficController) that mediates among the Airplanes.
- Each airplane communicates only to the Mediator to manage its operation.

Interfaces

- The Mediator interface notify to the sender using notify() method.
- The Airplane interface has the operation() method to send information to the Mediator.
- It has the receive() method to receive message from the Mediator.

Code #

```
1 # Interfaces
2 class Mediator(object):
3   def notify(sender): pass
4
5 class Airplane(object):
6   def __init__(self, id): self.id = id
7   def __str__(self): return self.id
8   def operation(self, mediator): pass
9   def receive(self, message): pass
```

Mediator Implementation

- Airtraffic Controller aggregates airplanes (line 4).
- Airplanes use the notify(sender) to notify any information to the Mediator.
- The react_on(id) is the reponse

Code #

```
1  # Implementations
2  class AirtrafficController(Mediator):
3    def __init__(self):
4      self._airplanes = {}
5    def __str__(self): return "AC1"
6    def register(self, airplane):
7      self._airplanes[airplane.id] = airplane
8    def notify(self, sender): # receive
9      self.react_on(sender.id)
10   def react_on(self, id): # send
11     message = ...
12     self._airplanes[id].receive(message)
```

Airplanes

Code #

```
class AirplaneA(Airplane):
  def operation(self, mediator):
    print(f"Airplane A notifies to the {mediator}")
    mediator.notify(self)
  def receive(self, message): print(message)

class AirplaneB(Airplane):
  def operation(self, mediator):
    print(f"Airplane B notifies to the {mediator}")
    mediator.notify(self)
  def receive(self, message): print(message)

class AirplaneC(Airplane):
  def operation(self, mediator):
    print(f"Airplane C notifies to the {mediator}")
    mediator.notify(self)
  def receive(self, message): print(message)
```

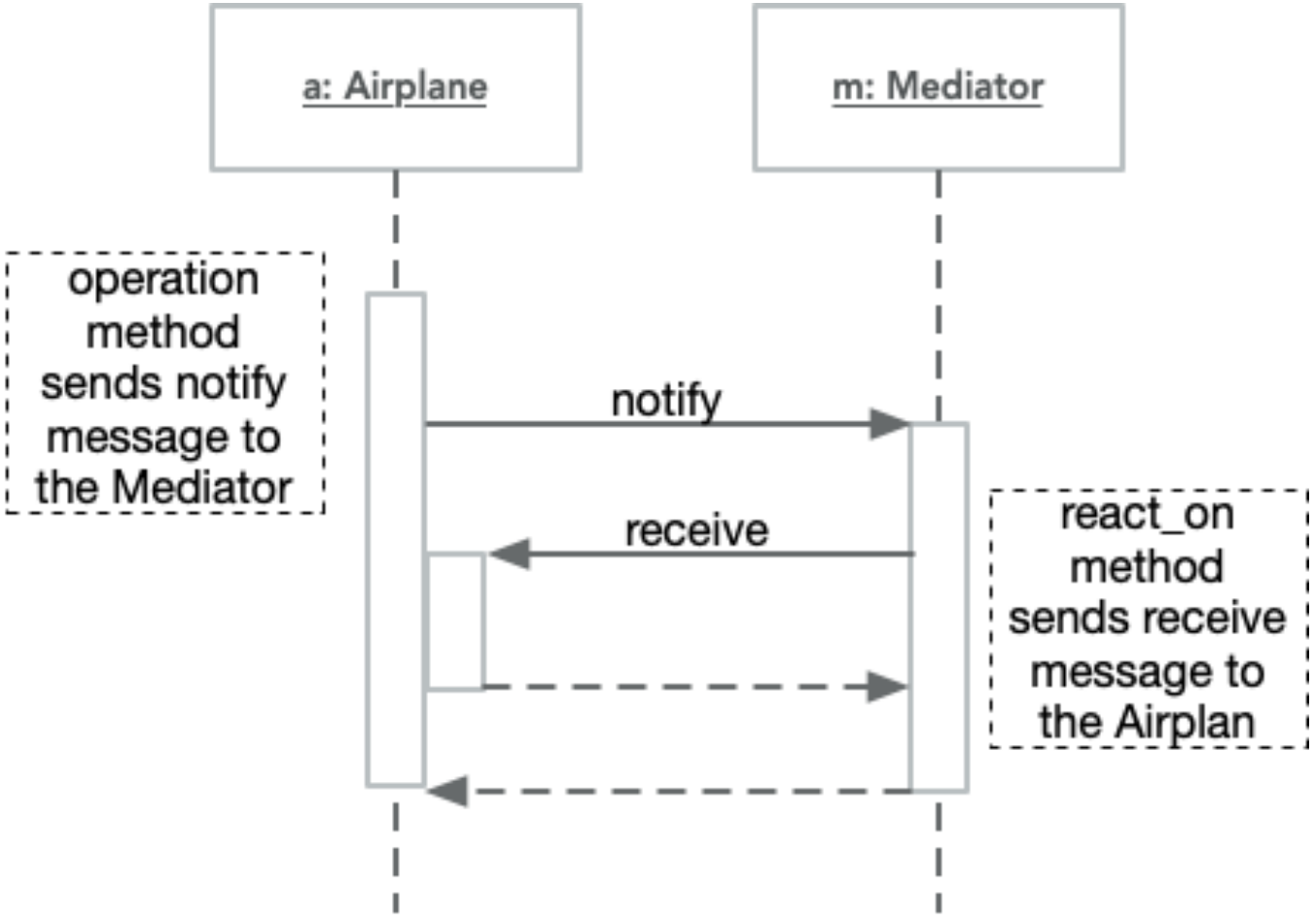
Driver

Code #

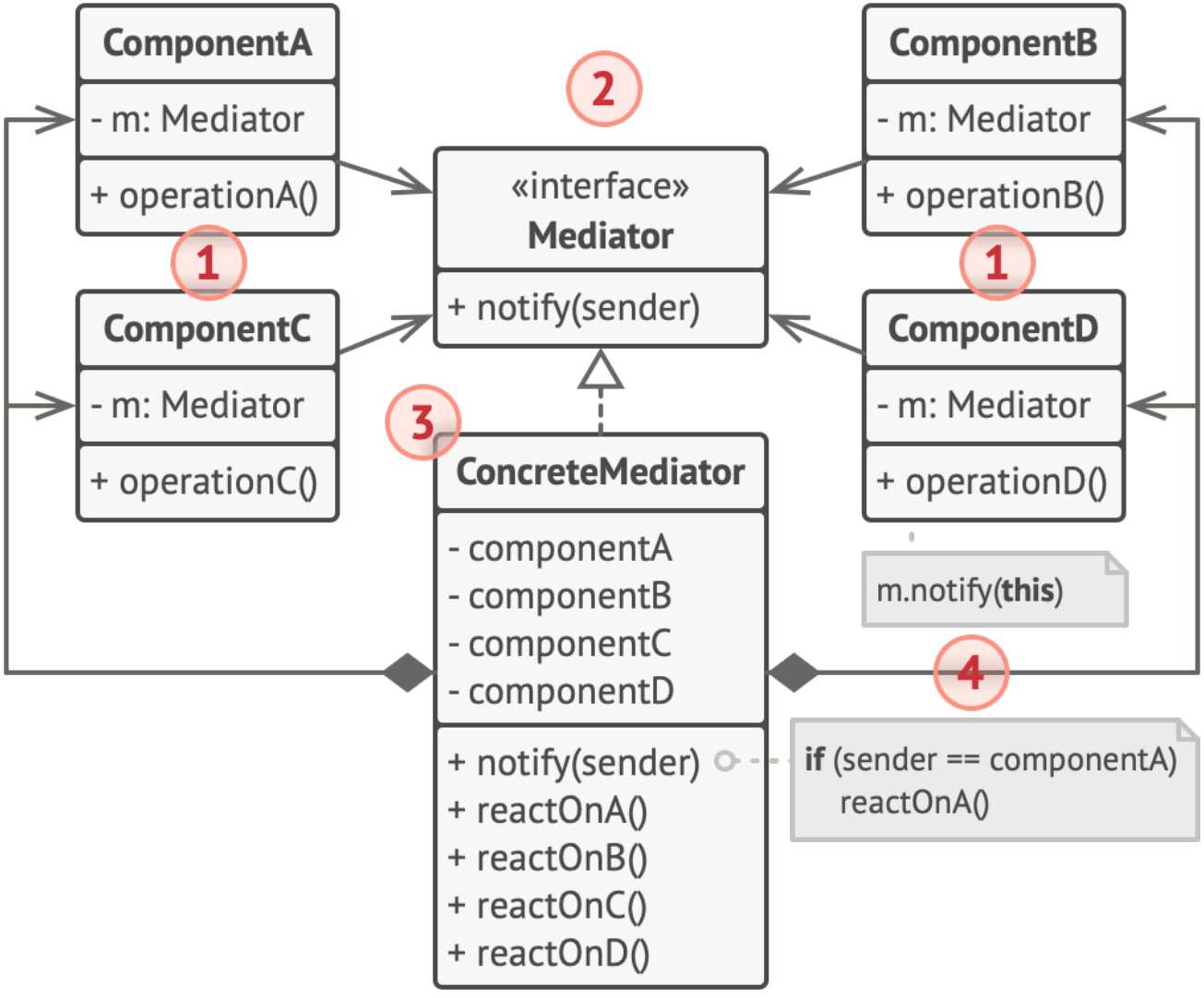
```
# Driver
m = AirtrafficController()
a = AirplaneA('a')
m.register(a)
b = AirplaneB('b')
m.register(b)
c = AirplaneC('c')
m.register(c)

# plane a notifies to the mediator
a.operation(m)
# plane b notifies to the mediator
b.operation(m)
# plane c notifies to the mediator
c.operation(m)
```

```
Airplane A notifies to the AC1
Mediator(AC1): responses to a
Airplane B notifies to the AC1
Mediator(AC1): responses to b
Airplane C notifies to the AC1
Mediator(AC1): responses to c
```

Sequence Diagram



Structure


ComponentA
- m: Mediator
+ operationA()
ComponentB
- m: Mediator
+ operationB()
2
«interface»
Mediator
+ notify(sender)
1
1
ComponentC
- m: Mediator
+ operationC()
ComponentD
- m: Mediator
+ operationD()
3
ConcreteMediator
- componentA
- componentB
- componentC
- componentD
+ notify(sender)
+ reactOnA()
+ reactOnB()
+ reactOnC()
+ reactOnD()
m.notify(this)
4
if (sender == componentA)
reactOnA()

Applicability

- Use the Mediator pattern when it's hard to change some of the classes because they are tightly coupled to a bunch of other classes.
- Use the pattern when you can't reuse a component in a different program because it's too dependent on other components.
- Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.

Pros

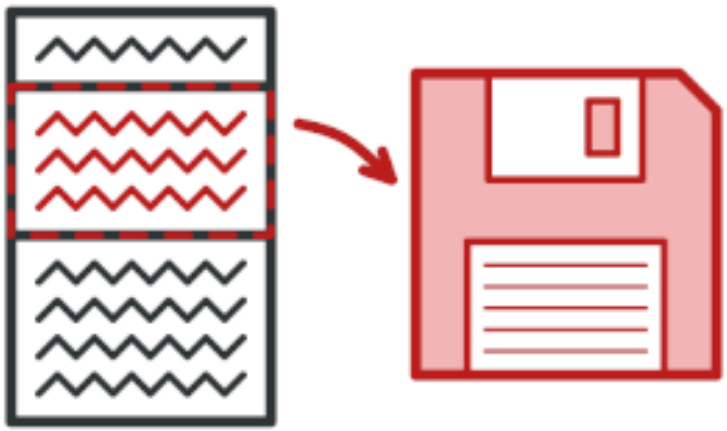
- Single Responsibility Principle. You can extract the communications between various components into a single place, making it easier to comprehend and maintain.
- Open/Closed Principle. You can introduce new mediators without having to change the actual components.
- You can reduce coupling between various components of a program.
- You can reuse individual components more easily.

Cons

- Over time a mediator can evolve into a God Object.

Relationship with Other Patterns

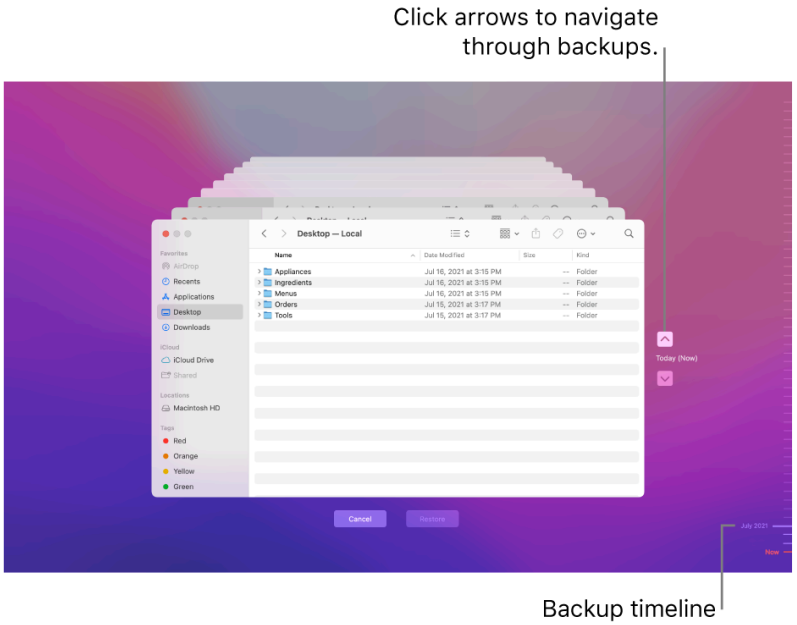
- Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests:
- Facade and Mediator have similar jobs: they try to organize collaboration between lots of tightly coupled classes.
- The difference between Mediator and Observer is often elusive. In most cases, you can implement either of these patterns; but sometimes you can apply both simultaneously. Let's see how we can do that.



Memento

Memento in Real World

- Apple's Time Machine backs up files on your Mac by making snapshots.
- When Time Machine is turned on, it automatically backs up your Mac and performs hourly, daily, and weekly backups of your files.



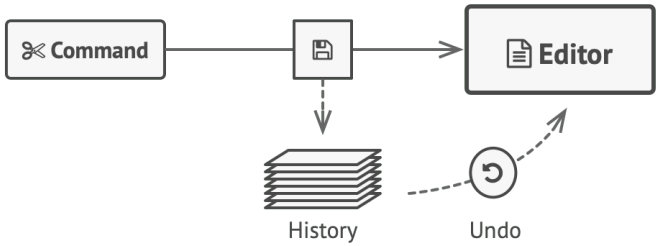
Memento Design Pattern

- Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Problem

- Imagine that you're creating a text editor app.
- People request the undo feature, but this would also require changing the classes responsible for copying the state of the affected objects.
- However, this would only work if the object had quite relaxed access restrictions to its contents.
- Unfortunately, most real objects won't let others peek inside them that easily, hiding all important data in private fields (encapsulation).

Solution



- All problems that we've just discussed are caused by encapsulation.
- The Memento pattern delegates creating the state snapshots to the actual owner of that state, the originator object.
- Hence, instead of other objects trying to copy the editor's state from the "outside," the editor class can make the snapshot since it has full access to its state.

Code

- We have an Originator that generates Snapshots whenever necessary.
- The Caretaker uses its Snapshot (Memento) to make a backup action ready for the undo; This Memento restores its previous states with the restore() method.

Interfaces

- Originator creates a Memento (Snapshot).
- Caretaker stores the Memento (Snapshot) and uses it to restore states with undo() method.

Code #

```
# interface
class Memento(object):
  def restore(self): pass
class Originator(object):
  def save(self): pass
class Caretaker(object):
  def undo(self): pass
```

Code #

```
# Implementations
class SnapShot(Memento):
  def __init__(self, state):
    self.state = state
  def restore(self, editor):
    editor.set_state(self.state)

class Editor(Originator):
  def set_state(self, state):
    self.state = state
  def save(self):
    return SnapShot(self.state)
  def print_state(self):
    print(f"Current editor state is {self.state}")

class Command(Caretaker):
  def __init__(self, editor):
    self.snapshot = None
    self.editor = editor
  def make_backup(self, snapshot):
    self.snapshot = snapshot
  def undo(self):
    self.snapshot.restore(self.editor)
```

Driver

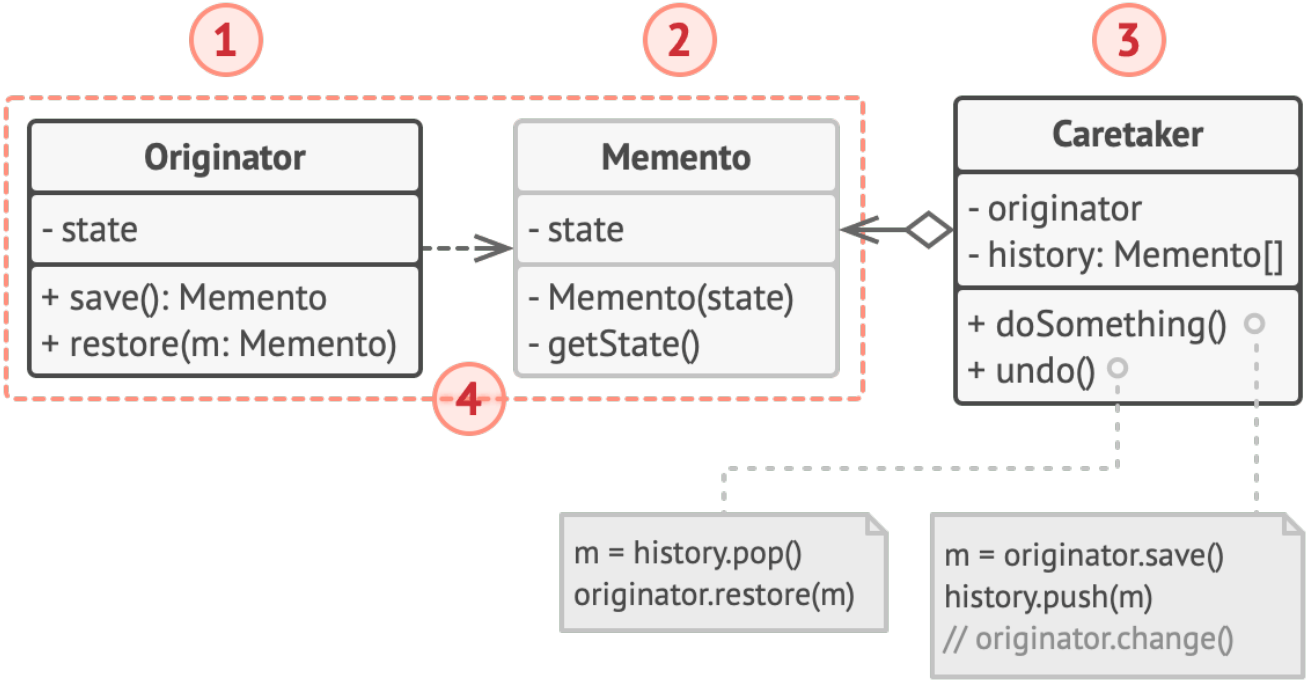
- Editor makes a snapshot (line 5).
- The Caretaker (Command) makes a backup (line 7).
- The Memento in the Caretaker recovers the state (line 12).

Code #

```
1  # Driver
2  editor = Editor(); editor.set_state(1) #
       ↪ state 1
3  editor.print_state()
4  # take a snapshot to store state 1
5  snapshot = editor.save()
6  caretaker = Command(e);
7  caretaker.make_backup(snapshot)
8  # state is now changed to 2
9  editor.set_state(2)
10 editor.print_state()
11
12 caretaker.undo()
13 editor.print_state()
```

```
Current editor state is 1
Current editor state is 2
Current editor state is 1
```

Structure


1
2
3
Originator
- state
+ save(): Memento
+ restore(m: Memento)
Memento
- state
- Memento(state)
- getState()
Caretaker
- originator
- history: Memento[]
+ doSomething()
+ undo()
4
m = history.pop()
originator.restore(m)
m = originator.save()
history.push(m)
// originator.change()

Applicability

- Use the Memento pattern when you want to produce snapshots of the object's state to be able to restore a previous state of the object.
- Use the pattern when direct access to the object's fields/getters/setters violates its encapsulation.

Pros

- You can produce snapshots of the object's state without violating its encapsulation.
- You can simplify the originator's code by letting the caretaker maintain the history of the originator's state.

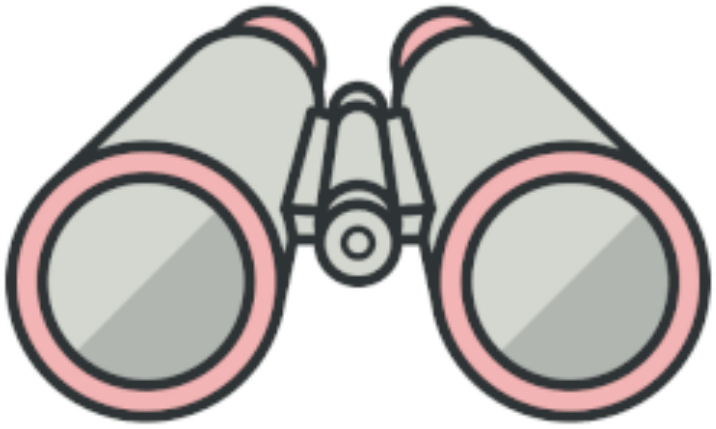
Cons

- The app might consume lots of RAM if clients create mementos too often.
- Caretakers should track the originator's lifecycle to be able to destroy obsolete mementos.
- Most dynamic programming languages, such as PHP, Python, and JavaScript, can't guarantee that the state within the memento stays untouched.

Relationship with Other Patterns

- You can use Command and Memento together when implementing "undo." In this case, commands are responsible for performing various operations over a target object. At the same time, mementos save the state of that object just before a command gets executed.
- You can use Memento along with Iterator to capture the current iteration state and roll it back if necessary.

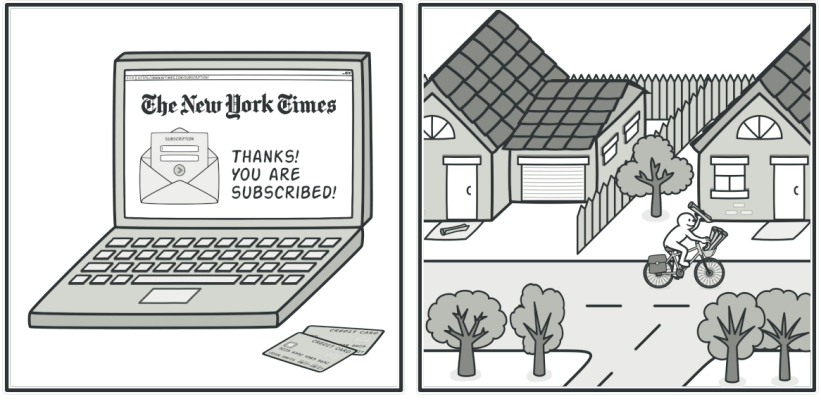
- Sometimes Prototype can be a simpler alternative to Memento. This works if the object, the state of which you want to store in the history, is fairly straightforward and doesn't have links to external resources, or the links are easy to re-establish.



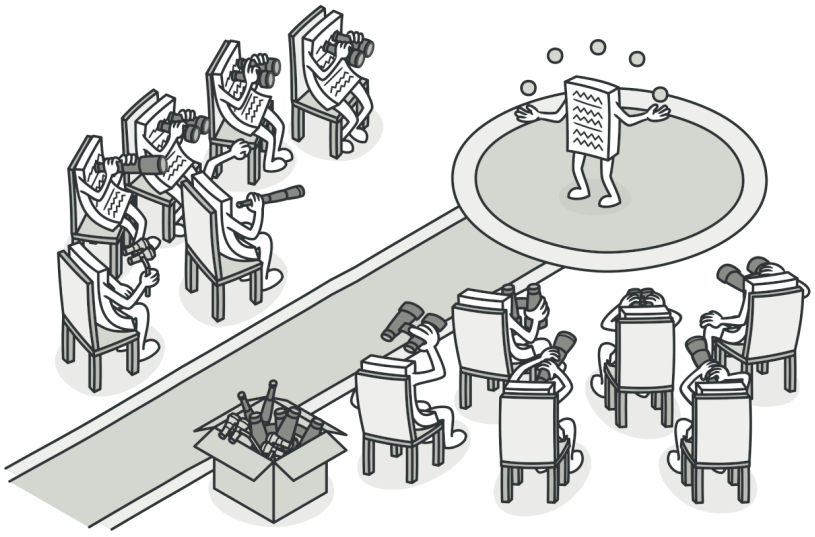
Observer

Observer in Real World

- If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available.
- Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance.

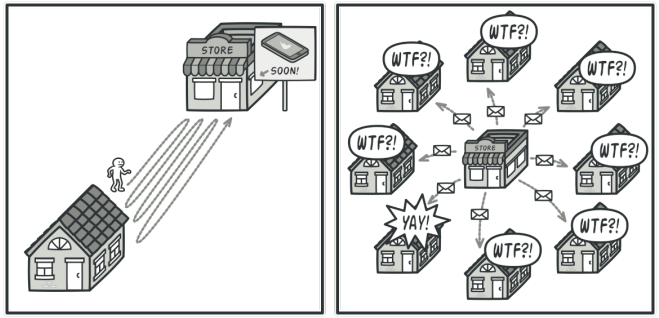


Observer Design Pattern

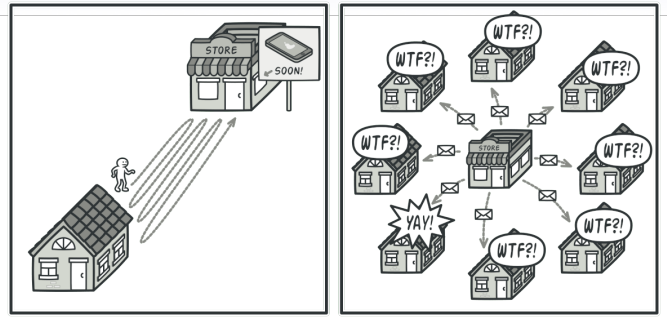


- Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Problem

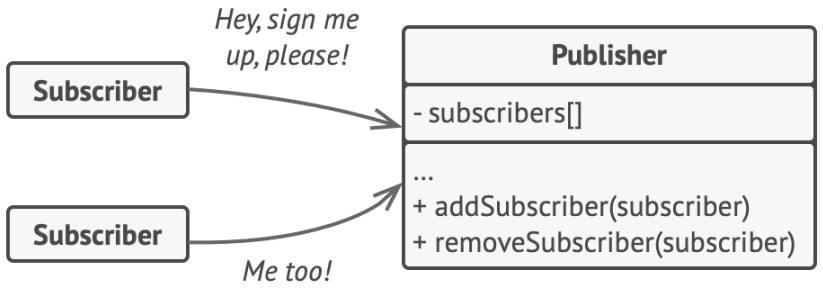


- Imagine that you have two types of objects: a Customer and a Store.
- The customer is very interested in a particular product, so the customer could visit the store every day and check product availability.
- But while the product is still en route, most of these trips would be pointless.



- On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available.
- This would save some customers from endless trips to the store.
- At the same time, it'd upset other customers who aren't interested in new products.

Solution



- The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.

Code Interfaces

- The Subsriber gets updated information.
- The Publisher notifies Substribers; it also subscribes or unsubscibes the Subscriber.

Code #

```
# interface
class Subscriber(object):
  def update(filename): pass
class Publisher(object):
  def subscribe(self, event_type, listner):
    pass
  def unsubscribe(self, event_type, listner):
    pass
  def notify(self, event_type, data):
    pass
```

Code #

```
# Implementation
class EventManager(Publisher):
  def __init__(self):
    self._listeners = []
    self.state = None
  def subscribe(self, listner):
    self._listeners.append(listner)
  def unsubscribe(self, listner):
    self._listeners.remove(listner)
  def notify(self):
    for listner in self._listeners:
      listner.update(self)

class ListnerA(Subscriber):
  def update(self, event_manager):
    print(f"Listner A is notified a new state {event_manager
        ↪ .state}")

class ListnerB(Subscriber):
  def update(self, event_manager):
    print(f"Listner B is notified a new state {event_manager
        ↪ .state}")
```

Driver

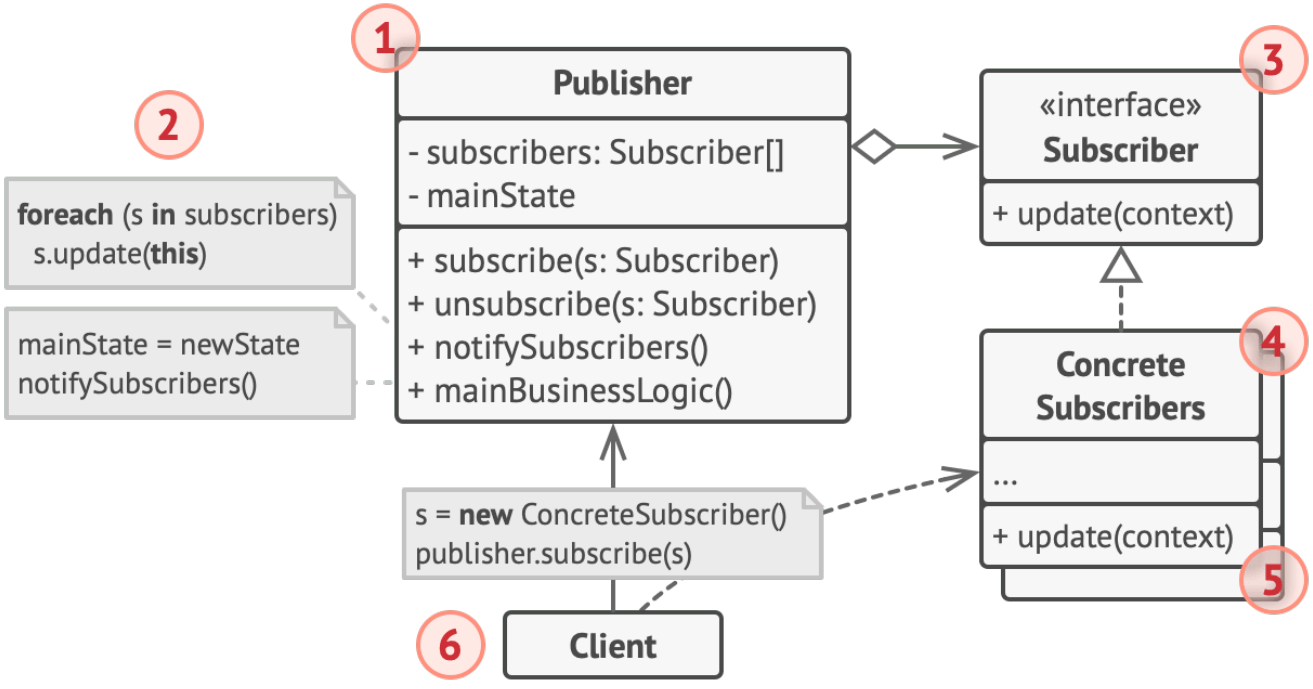
- EventManager(Subscriber) makes Subscribers (Listners A, B) its subcribers (line 5) and notifies its state (line 6).
- When ListenerA unsubscribes (line 10), only ListenerB is notified (line 12).

Code #

```
1  l_a = ListnerA()
2  l_b = ListnerB()
3
4  m = EventManager()
5  m.subscribe(l_a); m.subscribe(l_b)
6  m.state = 1
7  m.notify()
8
9  print('\n<<Removing A in the subscriber
       ↪ list>>')
10 m.unsubscribe(l_a)
11 m.state = 2
12 m.notify()
```

```
Listner A is notified a new state 1
Listner B is notified a new state 1

<<Removing A in the subscriber list>>
Listner B is notified a new state 2
```

Structure


1
Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
2
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
3
«interface»
Subscriber
+ update(context)
4
Concrete
Subscribers
...
+ update(context)
5
s = new ConcreteSubscriber()
publisher.subscribe(s)
6
Client

Event Driven Programming

- In CSC 360 event-driven programming, we already used this Observer design pattern.
- In this code, button new (btNews) as a Subscriber, registers itself to the Publisher (JavaFX) so that the click action is happening, it is invoked with the ActionEvent object.

Code #

```
1 btNew.setOnAction((ActionEvent e) -> {
2   System.out.println("Process New");
3 });
```

Applicability

- Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.
- Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.

Pros

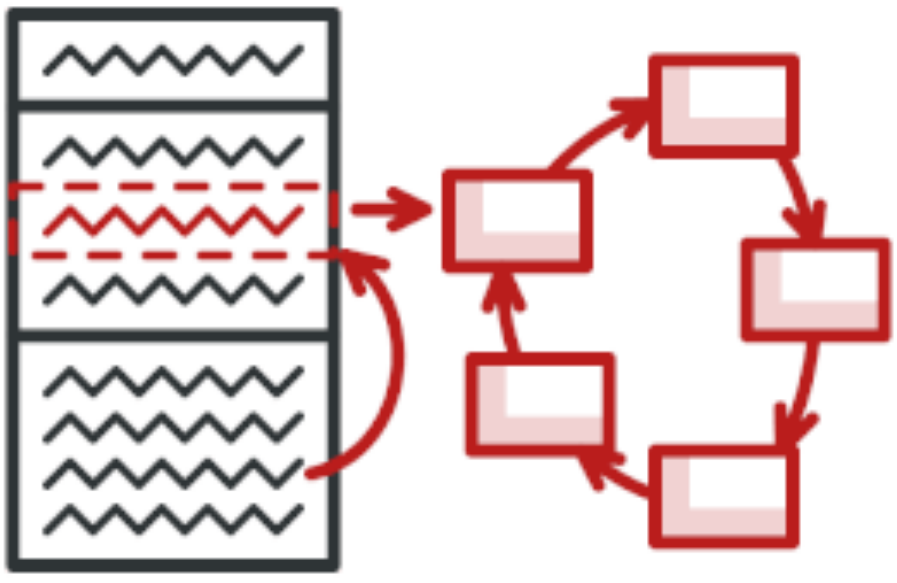
- Open/Closed Principle. You can introduce new subscriber classes without having to change the publisher's code (and vice versa if there's a publisher interface).
- You can establish relations between objects at runtime.

Cons

- Subscribers are notified in random order.

Relationship with Other Patterns

- Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests:
- The difference between Mediator and Observer is often elusive. In most cases, you can implement either of these patterns; but sometimes you can apply both simultaneously. Let's see how we can do that.

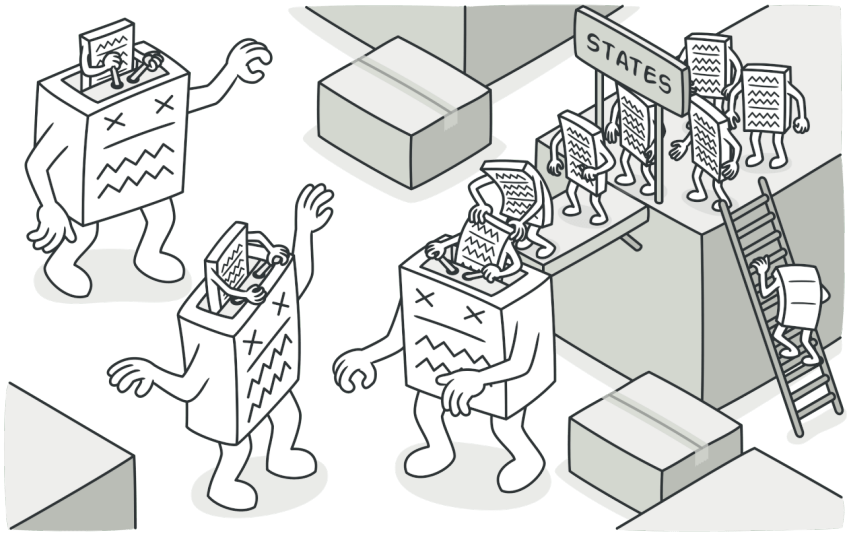


State

State in Real World

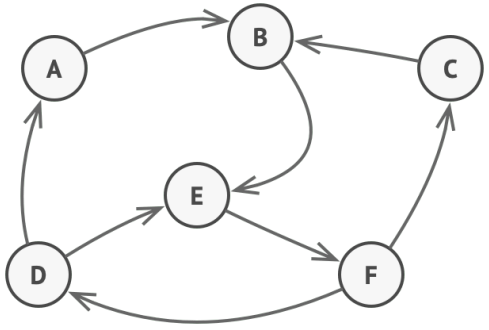
- It is interesting to observe that we use the State pattern in a variety of situations.
- The traffic signal has three states (go, stop, ready to stop).
- The music player has two states (play, stop).
- Each state transits to other states from the input (timer or button click).

State Design Pattern



- State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

Problem



- At any given moment, there's a finite number of states in which a program can be in.
- Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously.

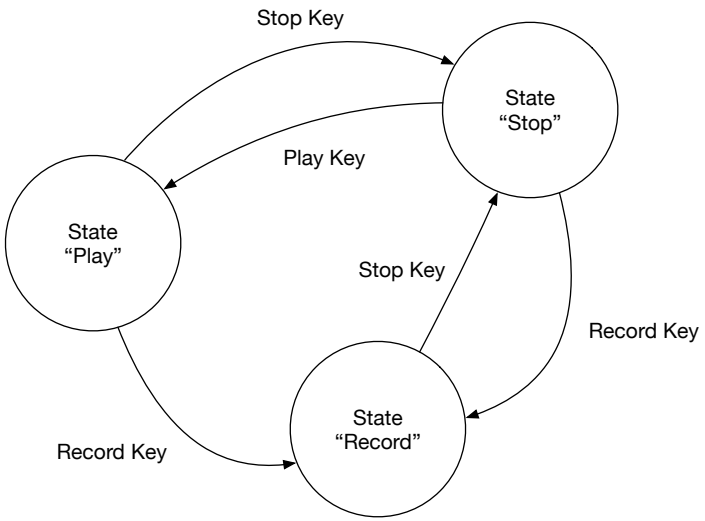
- However, depending on a current state, the program may or may not switch to certain other states.
- These switching rules, called transitions, are also finite and predetermined.
- How can we implement this system (finite state machine)?

Solution

- The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.

Code

- This state machine diagram shows how the tape recorder, which has three states (play, stop, and record) works.
- With each key action (Stop, Play, and Stop Key), the state transits from one state to another.



Interfaces

- Ths state has actions (lines 5 – 7)

Code #

```
1 # interfaces
2 class State(object):
3     def __init__(self, player):
4         self.player = player
5     def click_stop(self): pass
6     def click_play(self): pass
7     def click_record(self): pass
```

- Depending on which key is clicked, the state is changed (lines 4 and 12).
- For example, in the Record State, when the stop is clicked, the state is changed into Stop State (line 4).

Code #

```
1  # Implementations
2  class RecordState(State):
3    def __str__(self): return "Record state"
4    def click_stop(self): self.player.change_state(StopState(
       ↪ self.player))
5    def click_play(self): pass
6    def click_record(self): pass
7
8  class PlayState(State):
9    def __str__(self): return "Play state"
10   def click_stop(self): self.player.change_state(StopState(
       ↪ self.player))
11   def click_play(self): pass
12   def click_record(self): self.player.change_state(
       ↪ RecordState(self.player))
```

Player

- The Player class manages the current state.

Code #

```
# Client
class Player(object):
  def __init__(self):
    self.state = StopState(self)
  def change_state(self, state):
    print(f"Changing state to {state}")
    self.state = state
  def click_stop(self): self.state.click_stop()
  def click_play(self): self.state.click_play()
  def click_record(self): self.state.click_record()
```

Driver

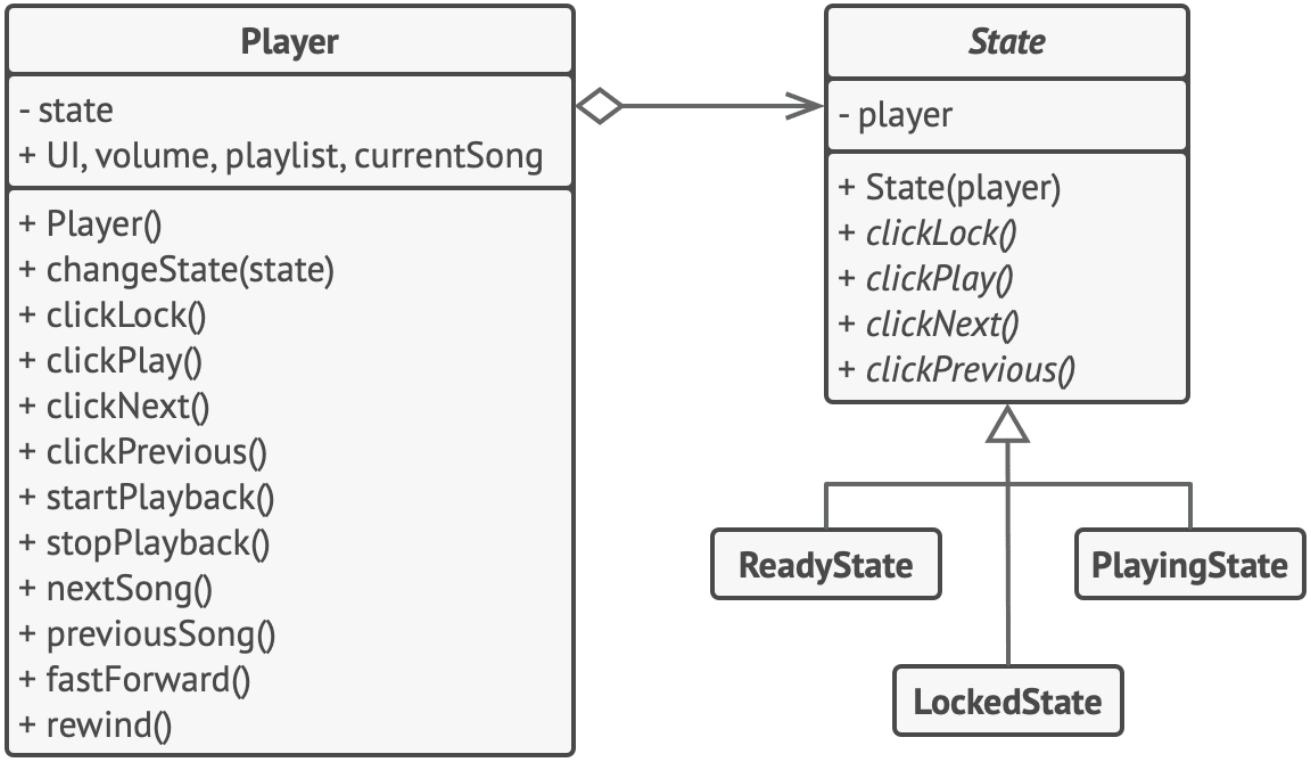
- Player changes its state from keyboard inputs (lines 3 – 7).
- Then, each state behaves differently depending on its state.

Code #

```
1 # Driver
2 p = Player()
3 p.click_play()
4 p.click_stop()
5 p.click_record()
6 # This will be ignored as there is
      ↪ no state transition
7 p.click_play()
8 p.click_stop()
```

```
Changing state to Play state
Changing state to Stop state
Changing state to Record state
Changing state to Stop state
```

Structure


Player
- state
+ UI, volume, playlist, currentSong
+ Player()
+ changeState(state)
+ clickLock()
+ clickPlay()
+ clickNext()
+ clickPrevious()
+ startPlayback()
+ stopPlayback()
+ nextSong()
+ previousSong()
+ fastForward()
+ rewind()
State
- player
+ State(player)
+ clickLock()
+ clickPlay()
+ clickNext()
+ clickPrevious()
ReadyState
PlayingState
LockedState

Applicability

- Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently.
- Use the pattern when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class's fields.
- Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.

Pros

- Single Responsibility Principle. Organize the code related to particular states into separate classes.
- Open/Closed Principle. Introduce new states without changing existing state classes or the context.
- Simplify the code of the context by eliminating bulky state machine conditionals.

Cons

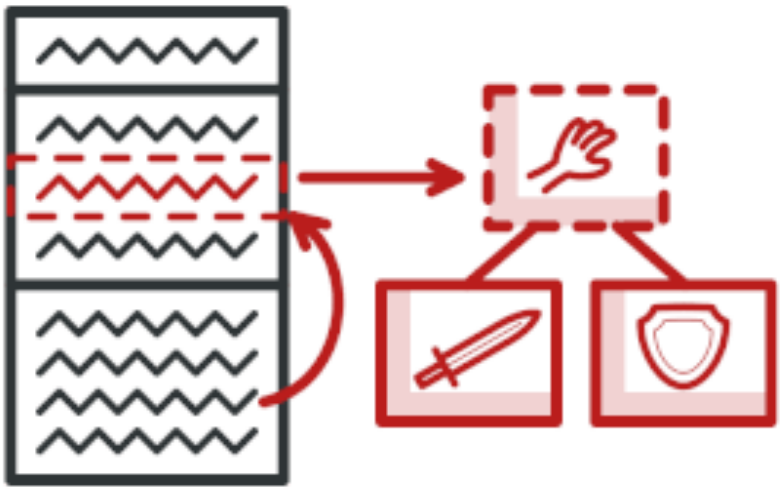
- Applying the pattern can be overkill if a state machine has only a few states or rarely changes.

Relationship with Other Patterns

- Bridge, State, Strategy (and to some degree Adapter) have very similar structures.
- Indeed, all of these patterns are based on composition/aggregation, which is delegating work to other objects.
- However, they all solve different problems.

Strategy Pattern

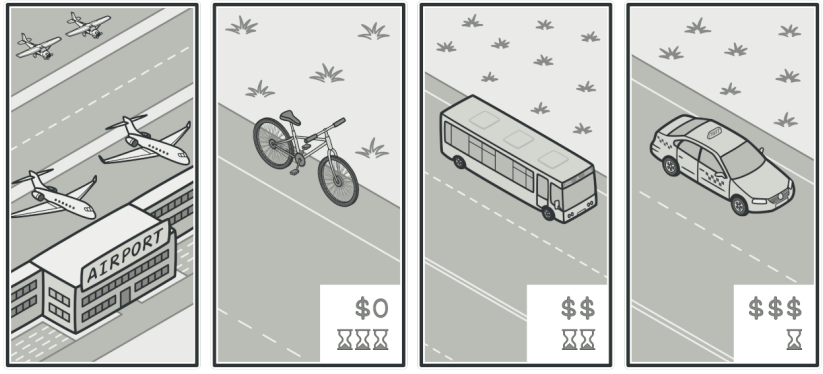
- State can be considered as an extension of Strategy.
- Both patterns are based on composition: they change the behavior of the context by delegating some work to helper objects.
- Strategy makes these objects completely independent and unaware of each other.
- However, State doesn't restrict dependencies between concrete states, letting them alter the state of the context at will.



Strategy

Strategy in Real World

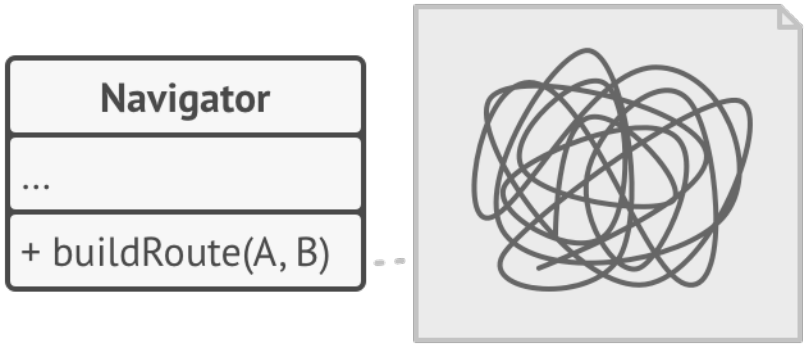
- Imagine that you have to get to the airport.
- You can catch a bus, order a taxi, or get on your bicycle.
- These are your transportation strategies; You can pick one of the strategies depending on factors such as budget or time constraints.



Strategy Design Pattern

- Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Problem



- Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the street score, affected the whole class, increasing the chance of creating an error in already-working code.

Solution

- The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called strategies.
- The original class, called context, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

Code

- In this example, we try to find the best strategy to get to the airport.
- When we don't have money but enough time, we choose a bicycle.
- When we don't have time, we choose a taxi.
- Otherwise, we choose the bus.

Interfaces

- The Strategy interface has one method: execute().

Code #

```
1 # Interfaces
2
3 class Strategy(object):
4   def execute(self): pass
```

- Each strategy BicycleStrategy, BusStrategy, and TaxiStragety implements the Strategy interface.

Code #

```
# Implementation
class BicycleStrategy(Strategy):
  def execute(self):
    print(f"Use bicycle - Time (2 hours): Cost ($0)")

class BusStrategy(Strategy):
  def execute(self):
    print(f"Use Bus - Time (1 hours): Cost ($5)")

class TaxiStrategy(Strategy):
  def execute(self):
    print(f"Use Taxi - Time (0.1 hours): Cost ($20)")
```

Context

- Context makes decision based on its computation results from two inputs: money and time.

Code #

```
1
2 # Client
3 class Context(object):
4   def set_strategy(self, strategy):
5     self.strategy = strategy
6   def make_decision(self, max_money, max_time):
7     if max_money < 3: # No money
8       self.set_strategy(BicycleStrategy())
9     elif max_time < 0.5:
10      self.set_strategy(TaxiStrategy())
11    else:
12      self.set_strategy(BusStrategy())
13    self.strategy.execute()
```

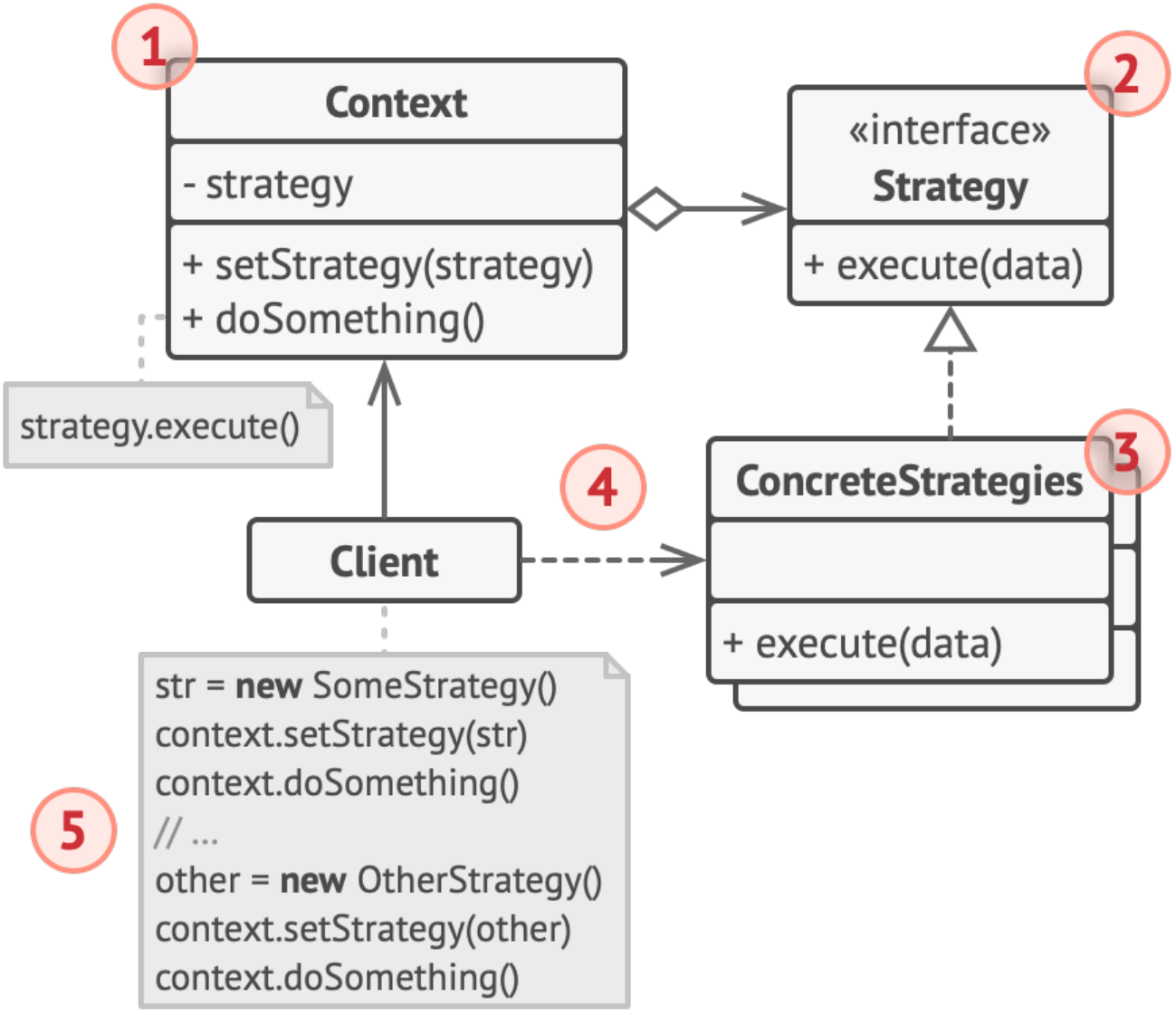
Driver

Code #

```
# Driver
c = Context()
# have some money, and have some time
c.make_decision(10, 2)
# Don't have money
c.make_decision(0, 5)
# Don't have time
c.make_decision(100, 0.2)
```

```
Use Bus - Time (1 hours): Cost ($5)
Use bicycle - Time (2 hours): Cost ($0)
Use Taxi - Time (0.1 hours): Cost ($20)
```

Structure


1
Context
- strategy
+ setStrategy(strategy)
+ doSomething()
2
«interface»
Strategy
+ execute(data)
strategy.execute()
4
3
ConcreteStrategies
Client
+ execute(data)
5
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()

Applicability

- Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
- Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.

- Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.
- Use the pattern when your class has a massive conditional strategy that switches between different variants of the same algorithm.

Pros

- You can swap algorithms used inside an object at runtime.
- You can isolate the implementation details of an algorithm from the code that uses it.
- You can replace inheritance with composition.
- Open/Closed Principle. You can introduce new strategies without having to change the context.

Cons

- If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
- Clients must be aware of the differences between strategies to be able to select a proper one.

- A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions; Then you could use these functions exactly as you'd have used the strategy objects, but without bloating your code with extra classes and interfaces.

Relationship with Other Patterns

- Bridge, Strategy, Strategy (and to some degree Adapter) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects.
- Command and Strategy may look similar because you can use both to parameterize an object with some action. However, they have very different intents.

- Decorator lets you change the skin of an object, while Strategy lets you change the guts.
- Template Method is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses. Strategy is based on composition: you can alter parts of the object's behavior by supplying it with different strategies that correspond to that behavior. Template Method works at the class level, so it's static. Strategy works on the object level, letting you switch behaviors at runtime.

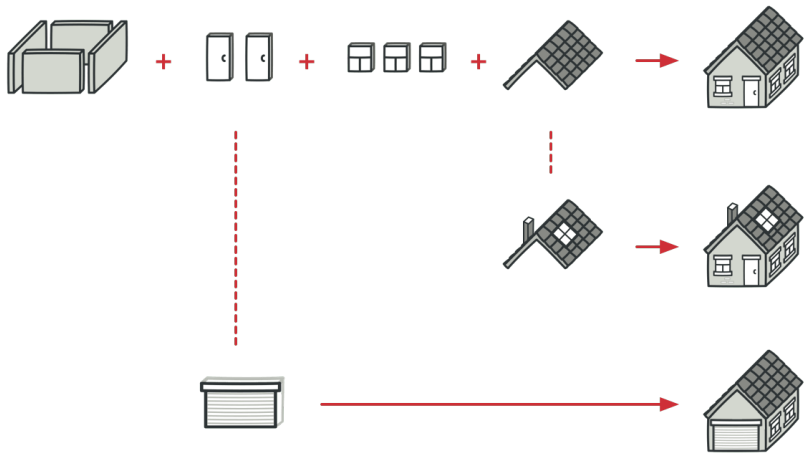
State

- State can be considered as an extension of Strategy.
- Both patterns are based on composition: they change the behavior of the context by delegating some work to helper objects.
- Strategy makes these objects completely independent and unaware of each other.
- However, strategy doesn't restrict dependencies between concrete strategys, letting them alter the strategy of the context at will.

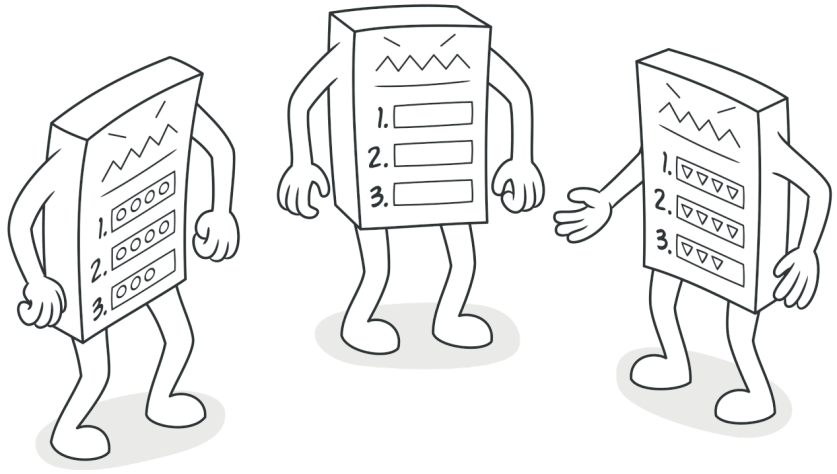
Template Method

Template in Real World

- The template method approach can be used in mass housing construction.
- The architectural plan for building a standard house may contain several extension points that would let a potential owner adjust some details of the resulting house.

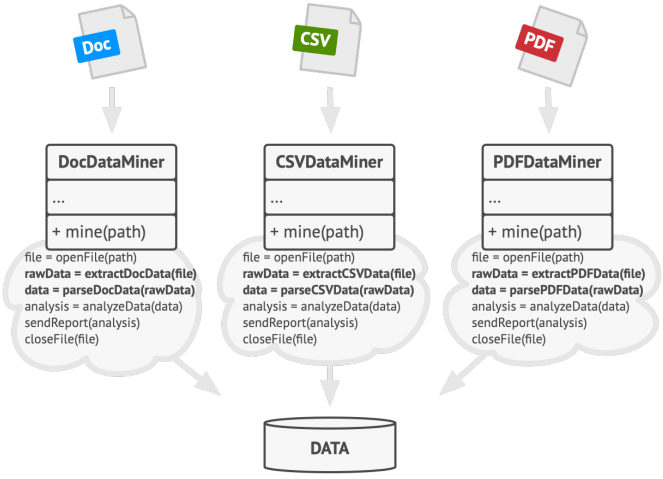


template Design Pattern



- Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Problem



- Imagine that you're creating a data mining application that analyzes corporate documents.
- Users feed the app documents in various formats (PDF, DOC, CSV), and it tries to extract meaningful data from these docs in a uniform format.

- At some point, you noticed that all three classes have a lot of similar code.
- While the code for dealing with various data formats was entirely different in all classes, the code for data processing and analysis is almost identical.
- Wouldn't it be great to get rid of the code duplication, leaving the algorithm structure intact?

Solution

- The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single template method.
- The steps may either be abstract, or have some default implementation.
- To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).

Code

- In this example, we build houses.
- The HouseBuild class builds a house with everything, while the SmallHouseBuild builds a house without many facilities that the HouseBuild has.
- Notice that many methods can be shared from the superclass, but some details may change.

Interfaces

Code #

```
# Interfaces
class AbstractBuild(object):
  def build_roof(self):
    print("Building roof")
  def build_walls(self):
    print("Building walls")
  def build_chimney(self):
    print("Building chimney")
  def build_basement(self):
    print("Building basement")
```

- SmallHouseBuild overrides the build_walls method as it has different approaches for building a wall.

Code #

```
# Implementations
class HouseBuild(AbstractBuild):
  def design_house(self):
    print("Design before building")
  def build(self):
    self.design_house()
    self.build_basement()
    self.build_walls()
    self.build_roof()
    self.build_chimney()

class SmallHouseBuild(AbstractBuild):
  def build_walls(self):
    print(">>> Build small walls")
  def build(self):
    self.build_walls()
    self.build_roof()
```

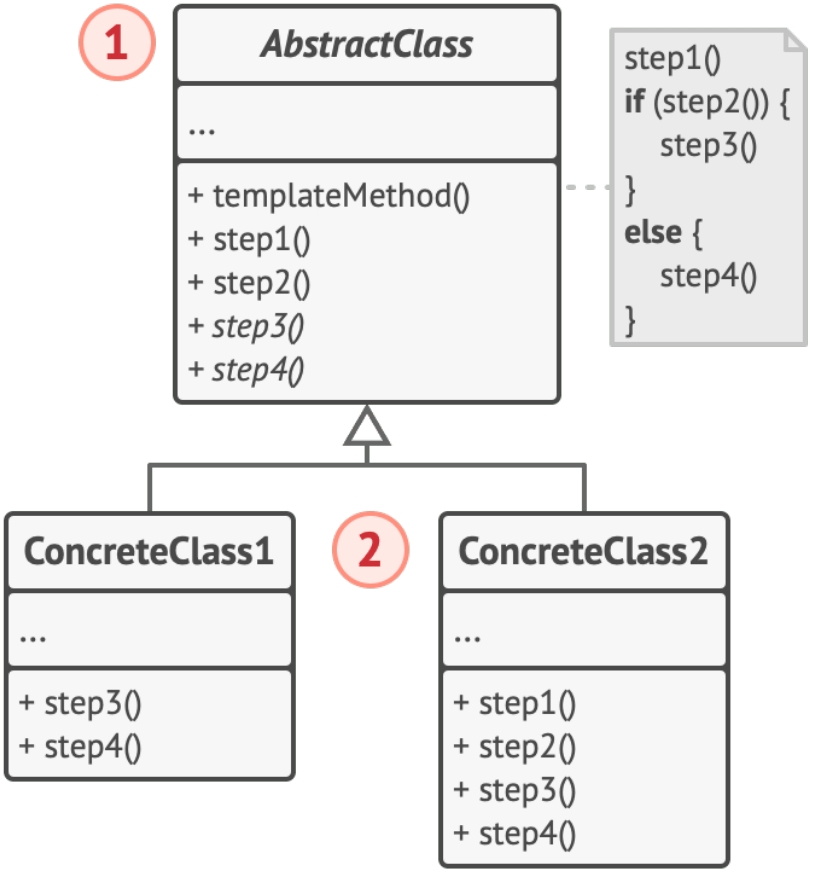
Driver

Code #

```
# Driver
b = HouseBuild()
b.build()

print("<Building a small house>")
s = SmallHouseBuild()
s.build()
```

```
Design before building
Building basement
Building walls
Building roof
Building chimney
<Building a small house>
Building walls
Building roof
```

Structure


1
AbstractClass
...
+ templateMethod()
+ step1()
+ step2()
+ step3()
+ step4()
step1()
if (step2()) {
step3()
}
else {
step4()
}
ConcreteClass1
...
+ step3()
+ step4()
2
ConcreteClass2
...
+ step1()
+ step2()
+ step3()
+ step4()

C++ Standard Template Library

- C++ supports containers in the form of Standard Template Libary (STL).
- The containers are objects that store data; The standard sequence containers include vector, deque, and list.

Code #

```
1 array arr{ 1, 2, 3, 4 };
2 // initialize a vector from an array
3 vector<int> numbers(cbegin(arr), cend(arr));
4 // insert more numbers into the vector
5 numbers.push_back(5);
6 ...
```

Applicability

- Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.
- Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify all classes when the algorithm changes.

Pros

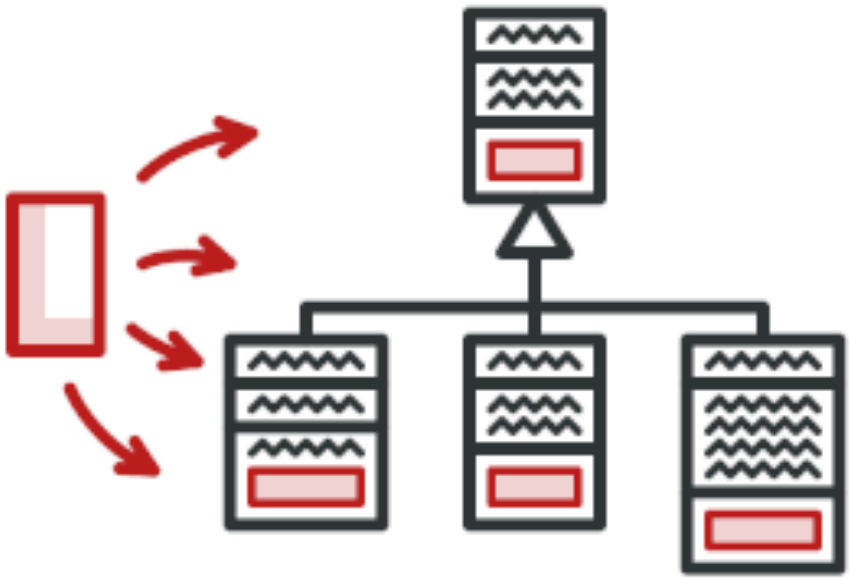
- You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.
- You can pull the duplicate code into a superclass.

Cons

- Some clients may be limited by the provided skeleton of an algorithm.
- You might violate the LSP, Liskov Substitution Principle, by suppressing a default step implementation via a subclass.
- Template methods tend to be harder to maintain the more steps they have.

Relationship with Other Patterns

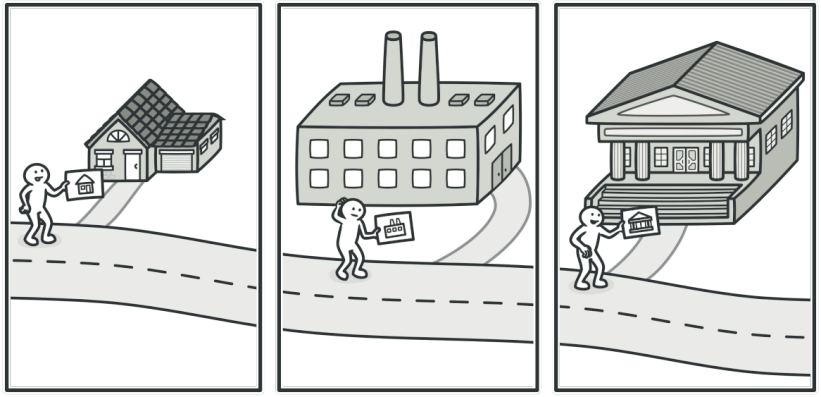
- Factory Method is a specialization of Template Method. At the same time, a Factory Method may serve as a step in a large Template Method.
- Template Method is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses. Strategy is based on composition: you can alter parts of the object's behavior by supplying it with different strategies that correspond to that behavior. Template Method works at the class level, so it's static. template works on the object level, letting you switch behaviors at runtime.



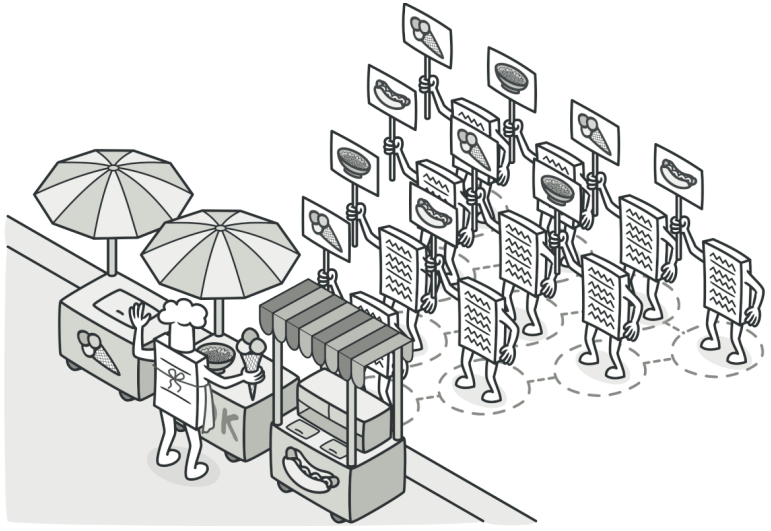
Visitor

Visitor in Real World

- Imagine an insurance agent who's eager to get new customers.
- He can visit every building in a neighborhood, trying to sell insurance to everyone he meets.
- Depending on the type of organization that occupies the building, he can offer specialized insurance policies.

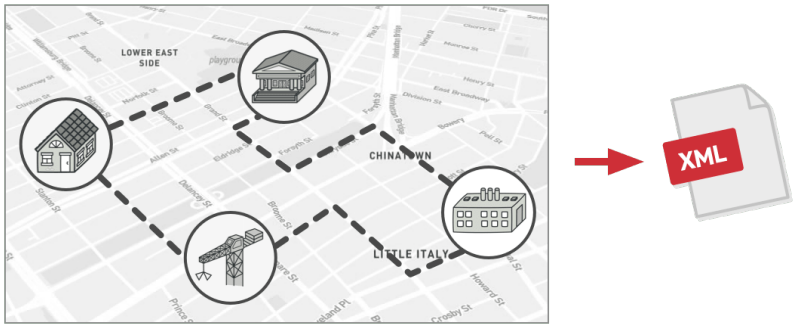


Visitor Design Pattern



- Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

Problem



- Imagine that your team develops an app which works with geographic information in the form of nodes.
- The node is represented by a class, and XML is generated from each node.
- Someone from the marketing department would ask you to provide the ability to export into a different format such as JSON.

Solution

- The Visitor pattern suggests that you place the new behavior into a separate class called visitor, instead of trying to integrate it into existing classes.
- The original object that had to perform the behavior is now passed to one of the visitorâĂŹs methods as an argument, providing the method access to all necessary data contained within the object.

Code

- In this example, we have a House (a Visitable) with Rooms (also a Visitable).
- We use the Visitor to visit the House (and the rooms) to get the total size of the size.

Interfaces

- Vistable accepts visitor.
- Visitor visits Visitable.

Code #

```
# interface
class Visitable(object):
  def accept(self, visitor): pass
class Visitor(object):
  def visit(self, element): pass
```

Visitables

- When the House (Visitable) accepts its Visitor, it invokes the Visitor's visit() method to iteratively visits all of its rooms.

Code #

```
# Implementations
class Room(Visitable):
  def __init__(self, size):
    self.size = size
  def accept(self, visitor):
      visitor.visitRoom(self)

class House(Visitable):
  def __init__(self, roomSizes):
   self.rooms = []
   for roomSize in roomSizes:
       room = Room(roomSize)
       self.rooms.append(room)
  def accept(self, visitor):
    visitor.visit(self)
```

Visitor

- Visitor knows all of Vistable's information.
- So, it can visitRoom or visit the house (visit each room iteratively).
- Python does not support ad-hoc polymorphism (different call from different arguments), so we should use visitRoom, not visit.

Code #

```
1  class Visitor(Visitor):
2    def __init__(self):
3      self.totalSize = 0
4
5    def visit(self, house):
6      self.totalSize = 0 # make sure the initial size is 0
7      # get the size of rooms
8      for room in house.rooms:
9          room.accept(self)
10   # Python doesn't support ad-hoc polymorphism
11   # It can be simply visit with Java
12   def visitRoom(self, room):
13     self.totalSize += room.size
```

Driver

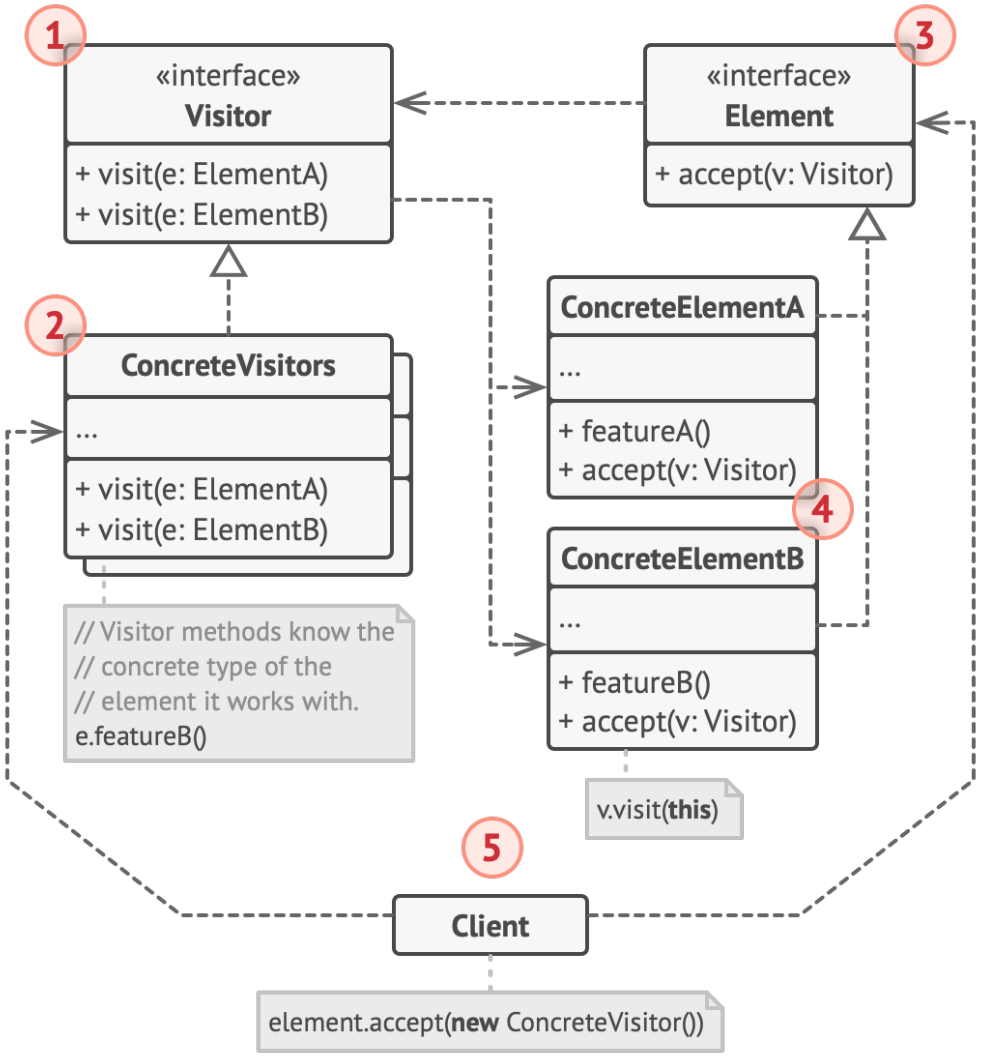
- All the business logic is hidden behind the simple visit() method.
- Once we have the roomsizes set up in the house (line 2), we can invoke visit method to iteratively adds all the room sizes.

Code #

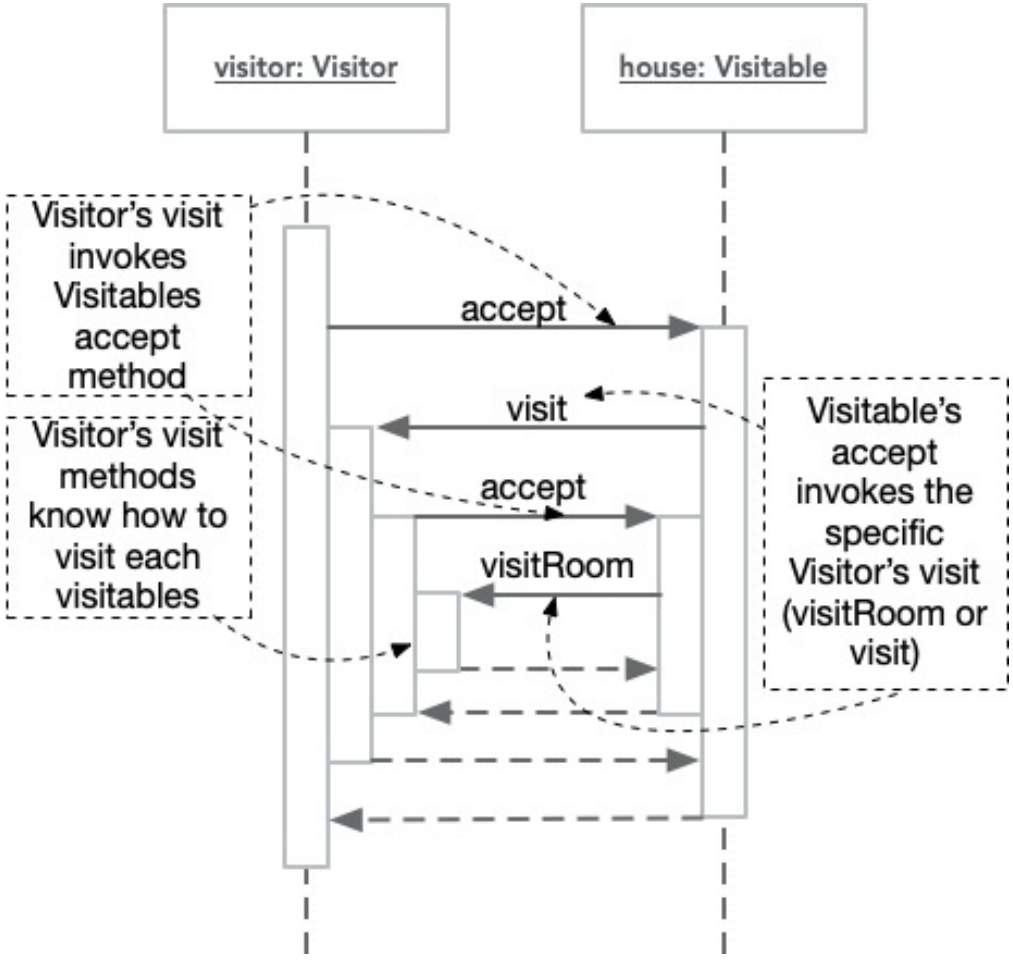
```
1 roomSizes = [200, 300, 500]
2 house = House(roomSizes)
3 visitor = Visitor()
4 visitor.visit(house)
5
6 print(visitor.totalSize) # 200 + 300
    ↪  + 500 = 1000
```

```
1000
```

Structure


1
«interface»
Visitor
+ visit(e: ElementA)
+ visit(e: ElementB)
3
«interface»
Element
+ accept(v: Visitor)
2
ConcreteVisitors
...
+ visit(e: ElementA)
+ visit(e: ElementB)
// Visitor methods know the
// concrete type of the
// element it works with.
e.featureB()
ConcreteElementA
...
+ featureA()
+ accept(v: Visitor)
4
ConcreteElementB
...
+ featureB()
+ accept(v: Visitor)
v.visit(this)
5
Client
element.accept(new ConcreteVisitor())

Sequence Diagram



Applicability

- Use the Visitor when you need to perform an operation on all elements of a complex object structure (for example, an object tree).
- Use the Visitor to clean up the business logic of auxiliary behaviors.

Pros

- Open/Closed Principle. You can introduce a new behavior that can work with objects of different classes without changing these classes.
- Single Responsibility Principle. You can move multiple versions of the same behavior into the same class.
- A visitor object can accumulate some useful information while working with various objects. This might be handy when you want to traverse some complex object structure, such as an object tree, and apply the visitor to each object of this structure.

Cons

- You need to update all visitors each time a class gets added to or removed from the element hierarchy.
- Visitors might lack the necessary access to the private fields and methods of the elements that they're supposed to work with.

Relationship with Other Patterns

- You can treat Visitor as a powerful version of the Command pattern. Its objects can execute operations over various objects of different classes.
- You can use Visitor to execute an operation over an entire Composite tree.
- You can use Visitor along with Iterator to traverse a complex data structure and execute some operation over its elements, even if they all have different classes.