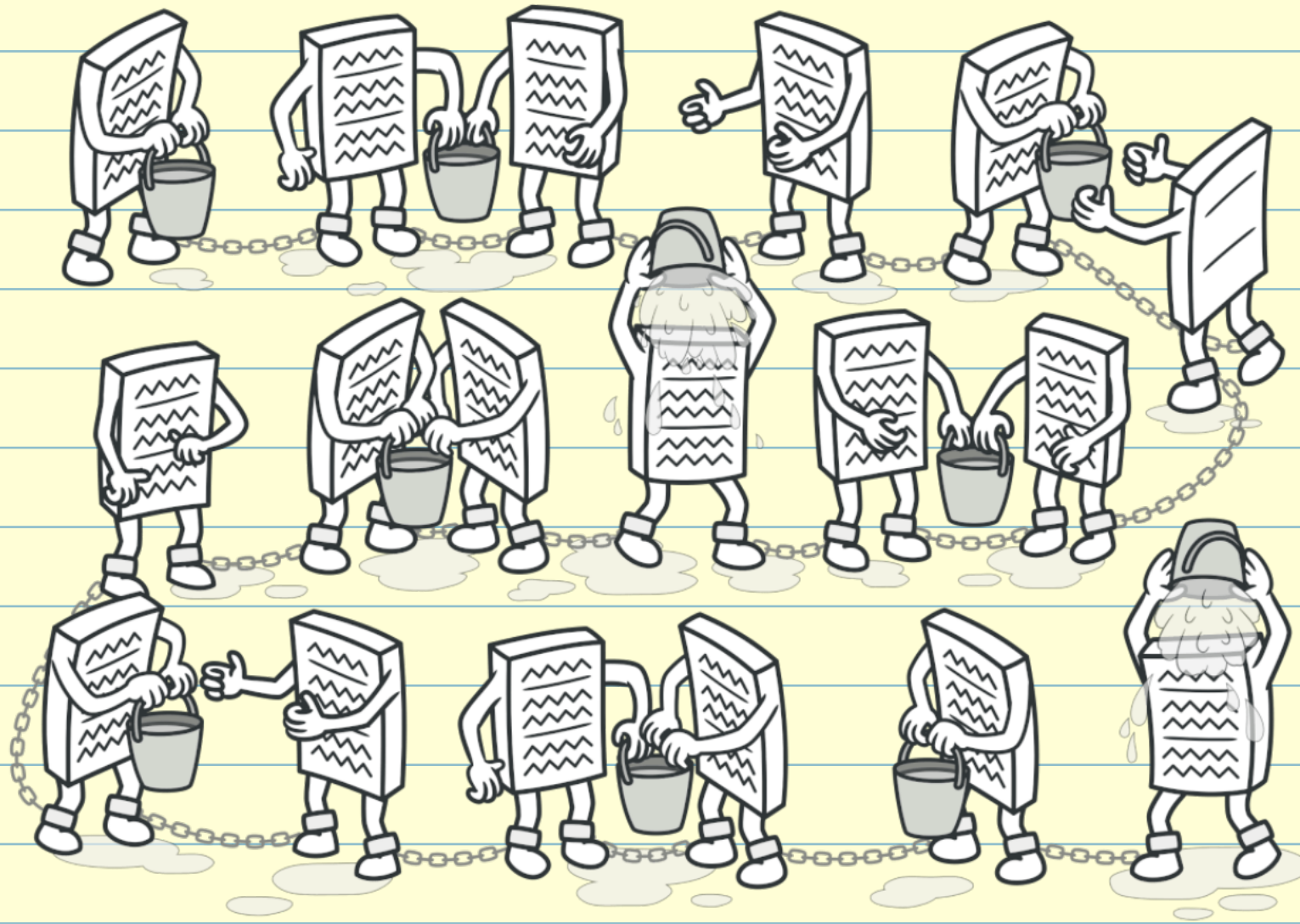
Behavioral Design Patterns

- Command
- Chain of Responsibility
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method

Chain of Responsibility



- lets you pass requests along a chain of handlers. Upon recieving a request, each handler deside either to process the request or to pass it to the next hardler in the chain.

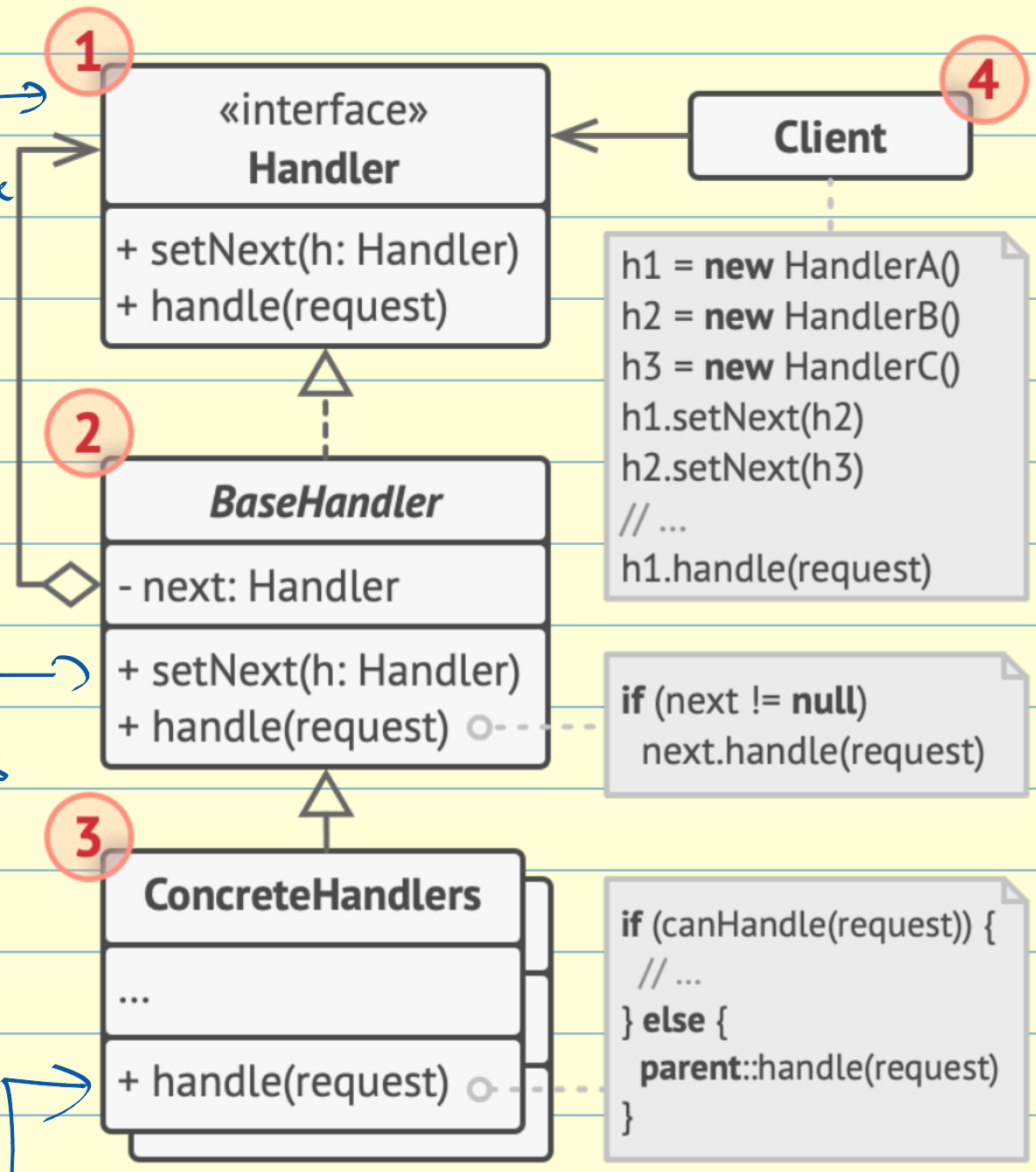
- This pattern encourages loose coupling between sender and recieven, providing freedom in handling the request.

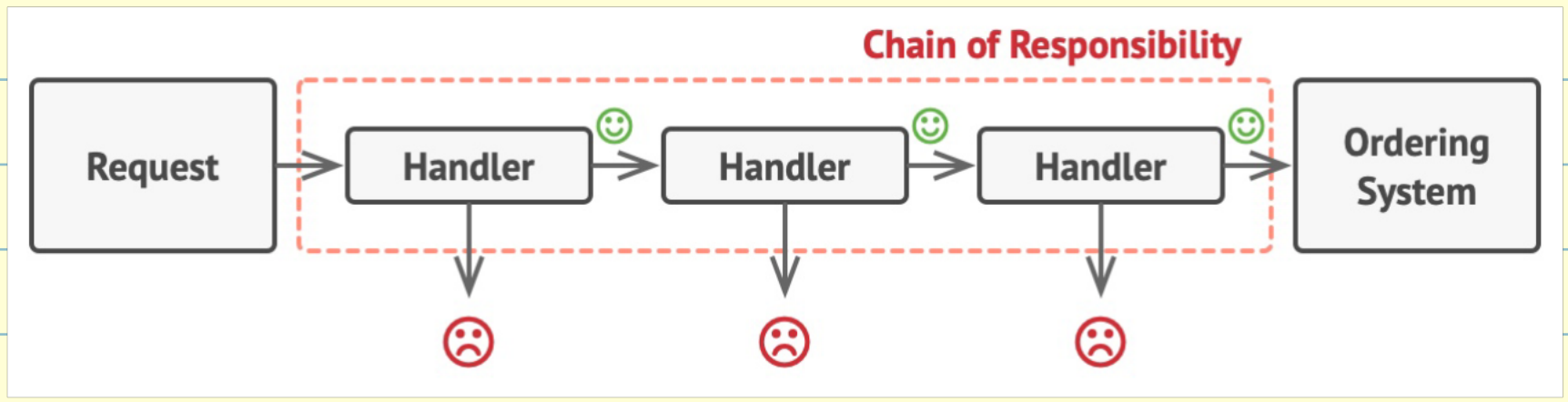
- requests are passed sequentially along the chain, ensuring that each request handled in a predefined order.

common interface for all concrete handlers.

optional class, usually this class defines to store reference to the next handler.

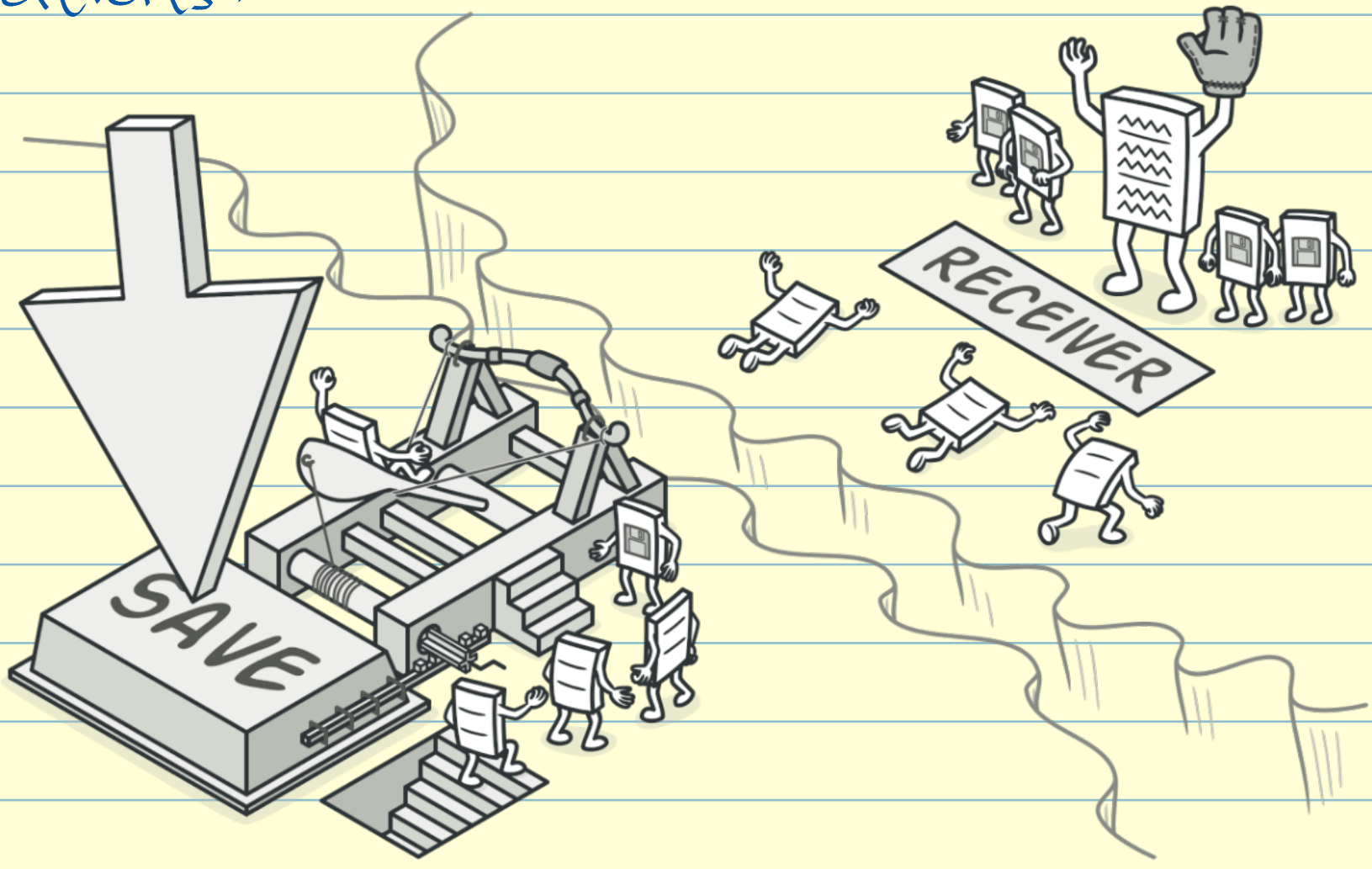
contain actual code for processing requests.





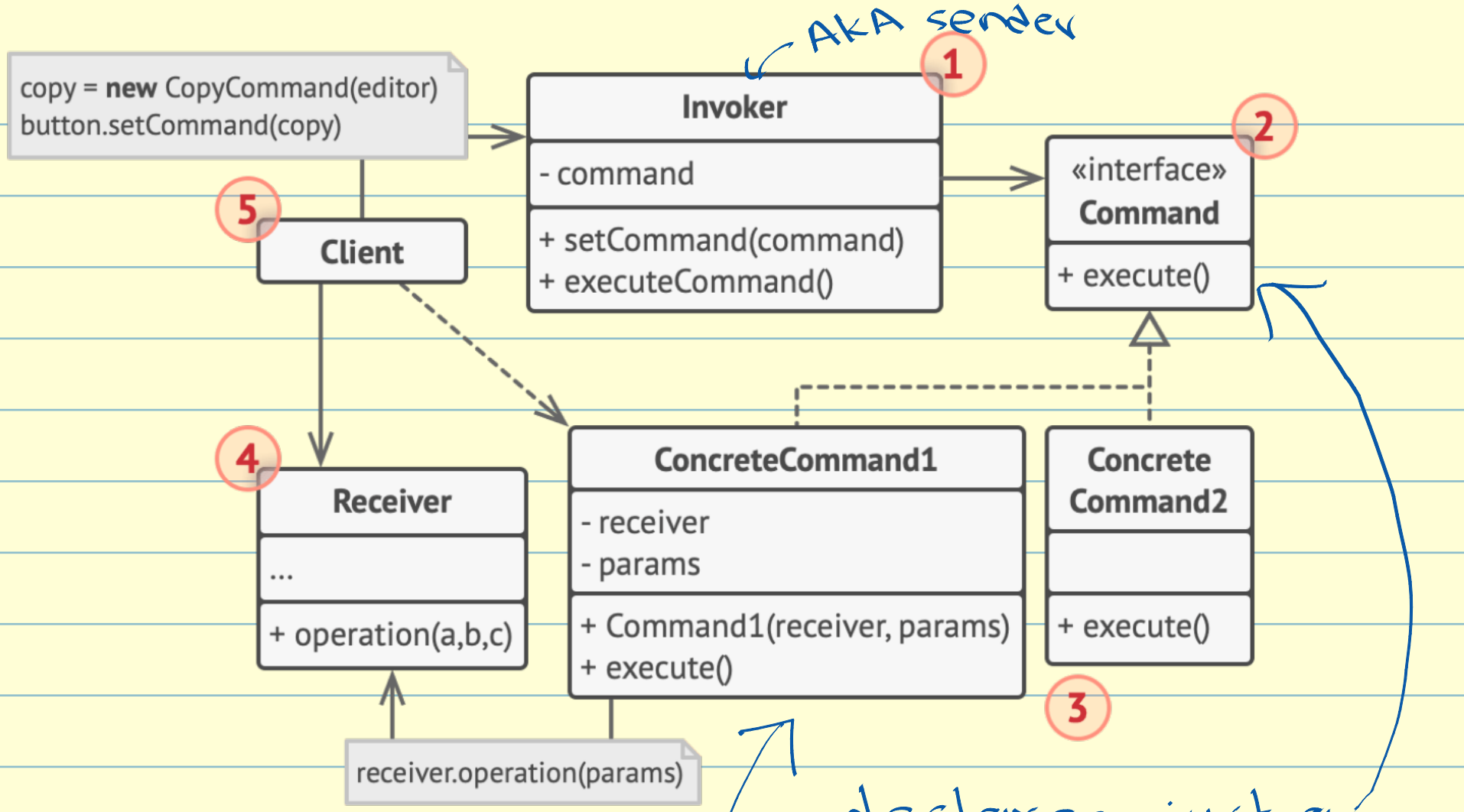
Command

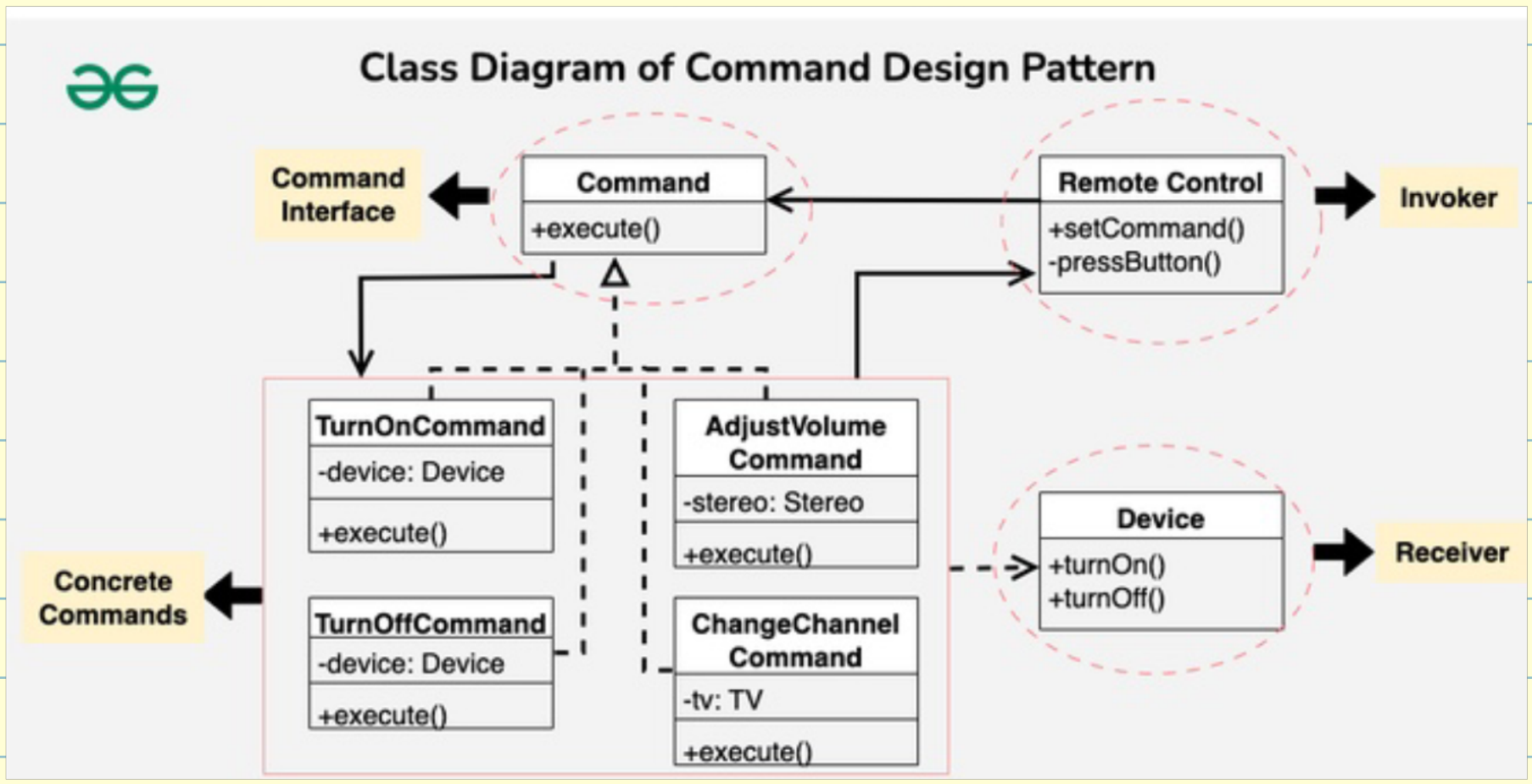
- turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.



When to use the command design patter ?

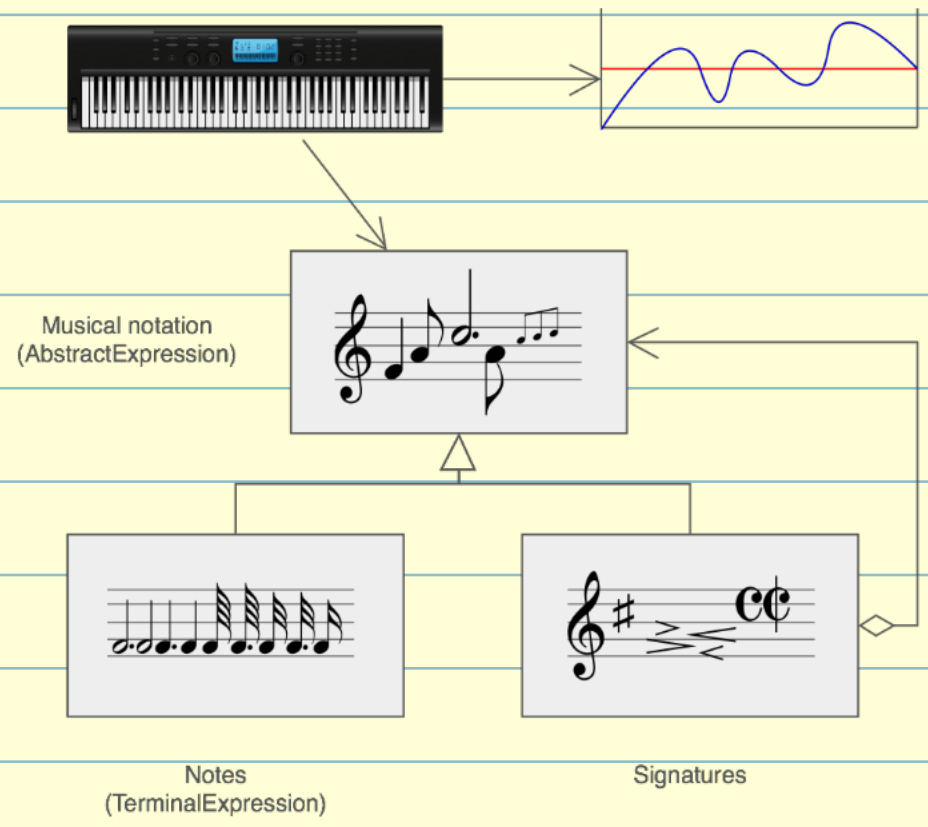
- When you need decouple sender of a request from the object that performs the request.
- When Redo / Undo functionality is required
- Support for queue and logging.

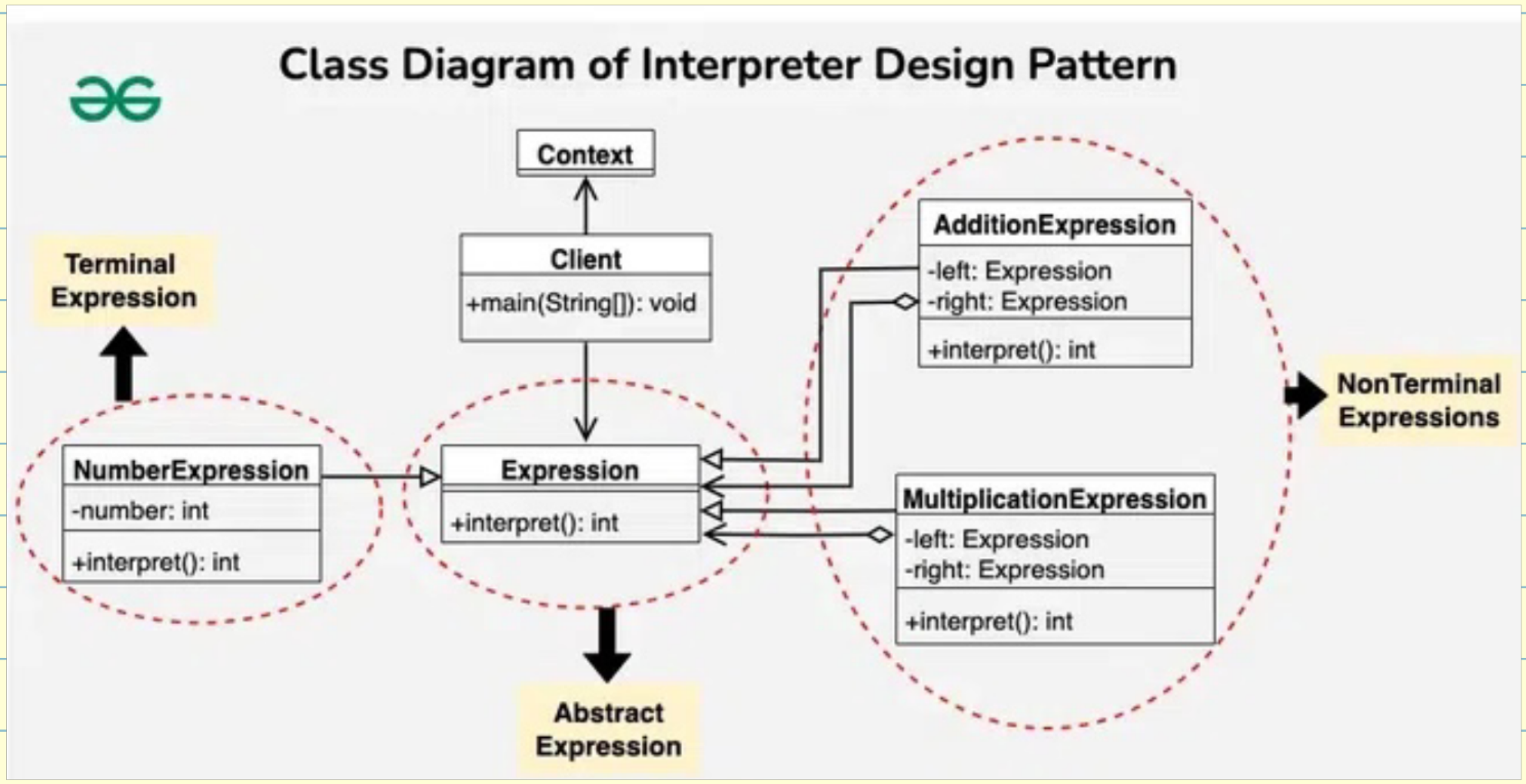

AKA sender
copy = new CopyCommand(editor)
button.setCommand(copy)
1
Invoker
- command
+ setCommand(command)
+ executeCommand()
2
«interface»
Command
+ execute()
5
Client
4
Receiver
...
+ operation(a,b,c)
ConcreteCommand1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Concrete
Command2
+ execute()
3
receiver.operation(params)
declares just a single method for excuting the command.
implement various requests.


Class Diagram of Command Design Pattern
Command Interface
Command
+execute()
Remote Control
+setCommand()
-pressButton()
Invoker
TurnOnCommand
-device: Device
+execute()
AdjustVolume Command
-stereo: Stereo
+execute()
Concrete Commands
TurnOffCommand
-device: Device
+execute()
ChangeChannel Command
-tv: TV
+execute()
Device
+turnOn()
+turnOff()
Receiver

Interpretor

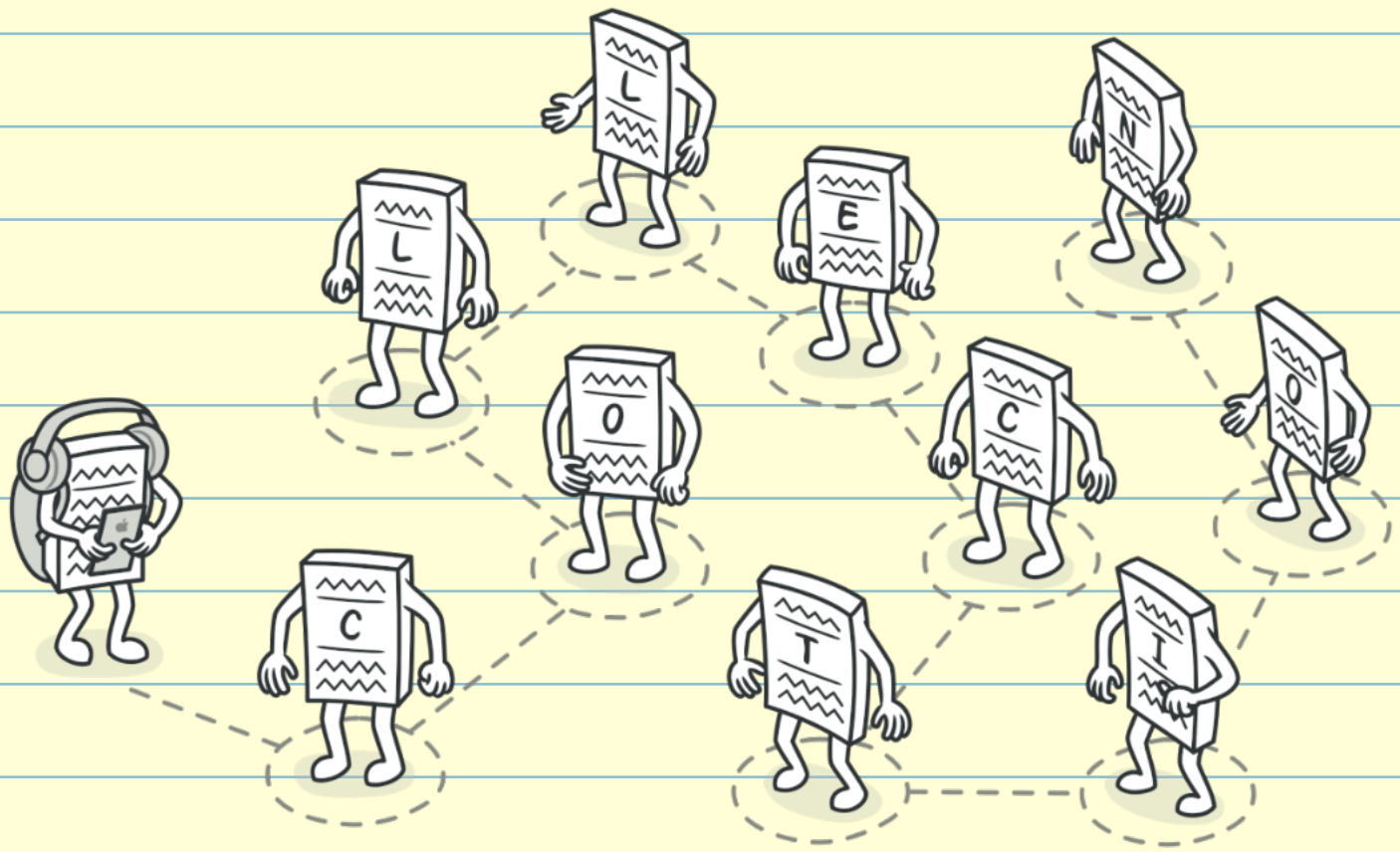
- defines a representation for its grammer along with an interpreter that uses the representation to interpret sentences in a language.

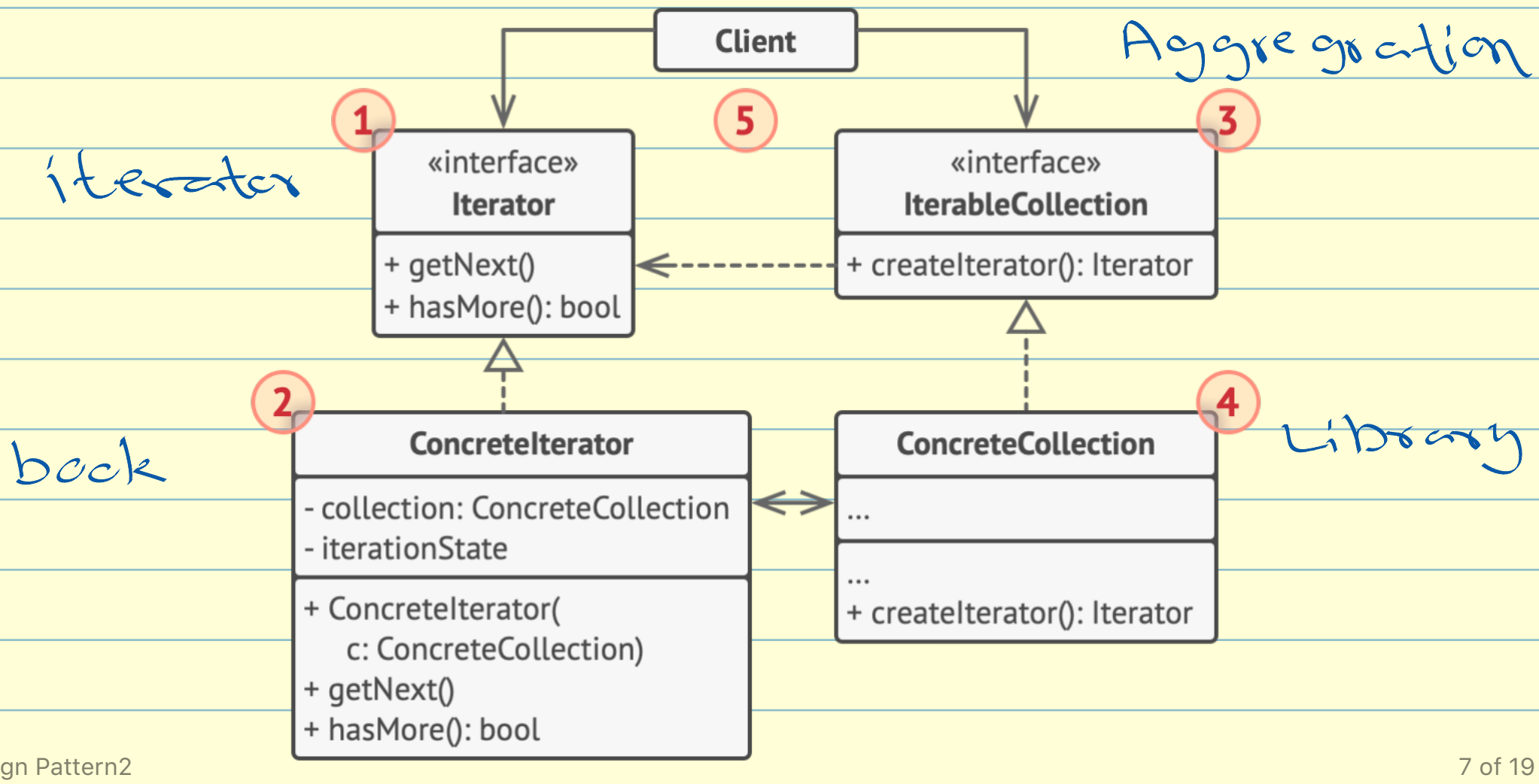




Iterator

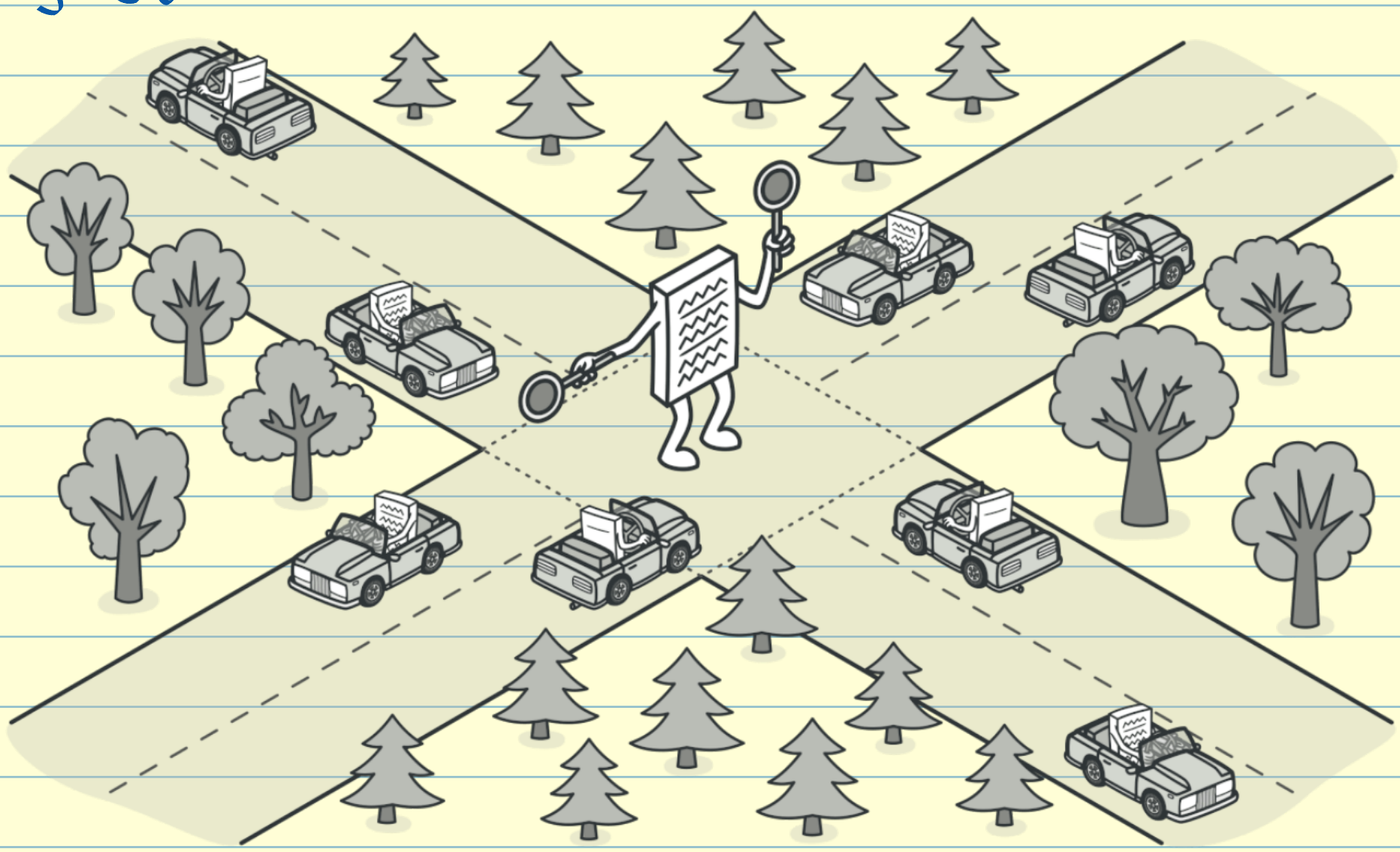
-Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation



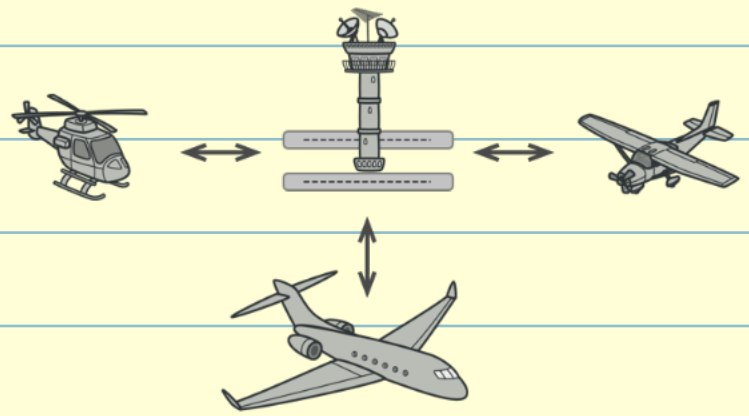


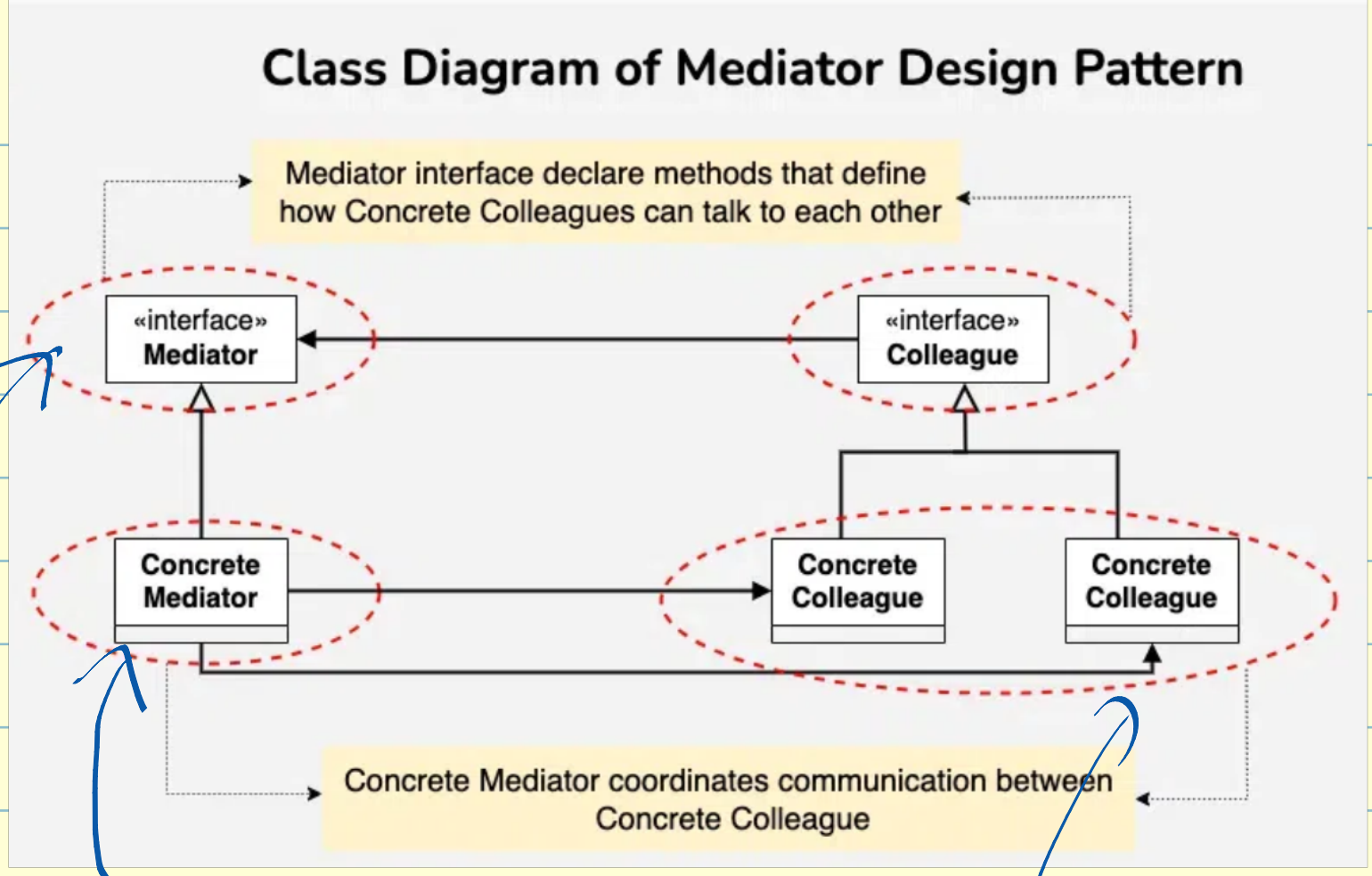
Mediator

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



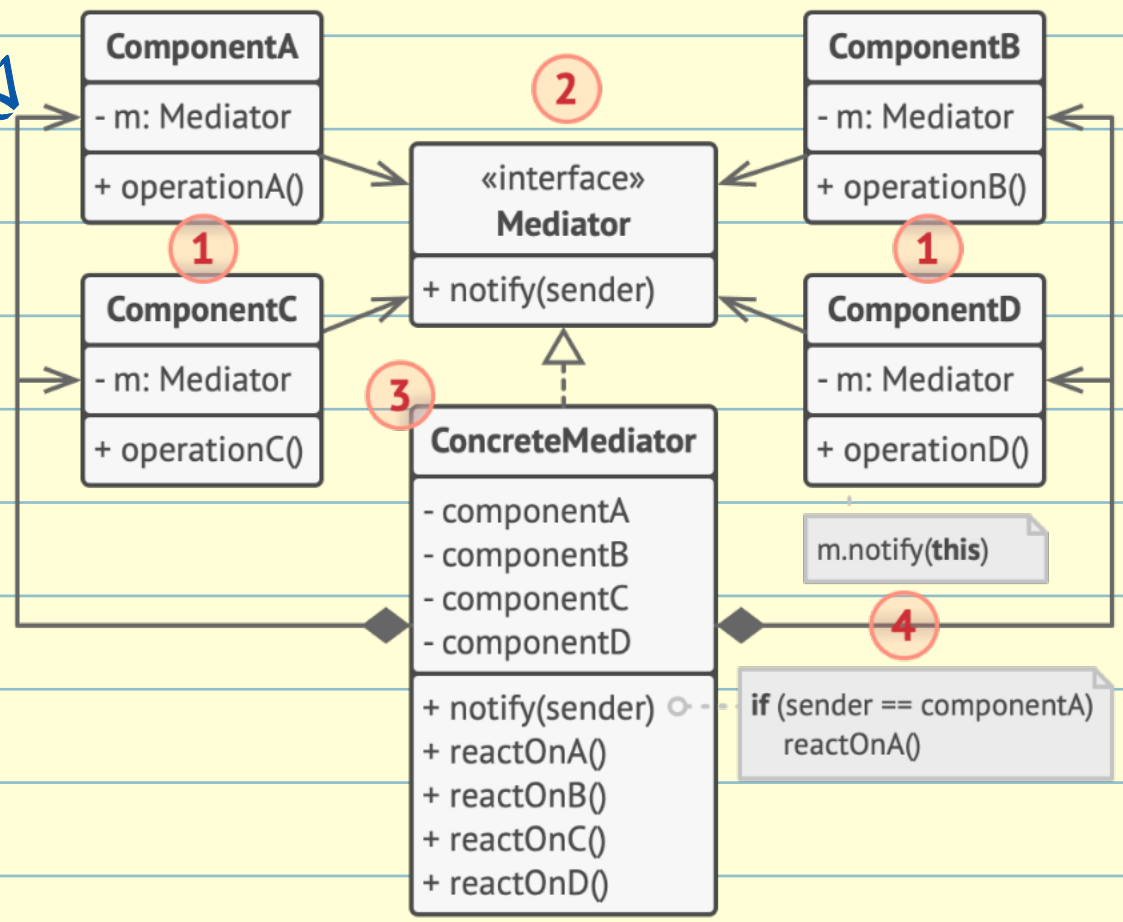
ex:- online bidding system
air traffic management system





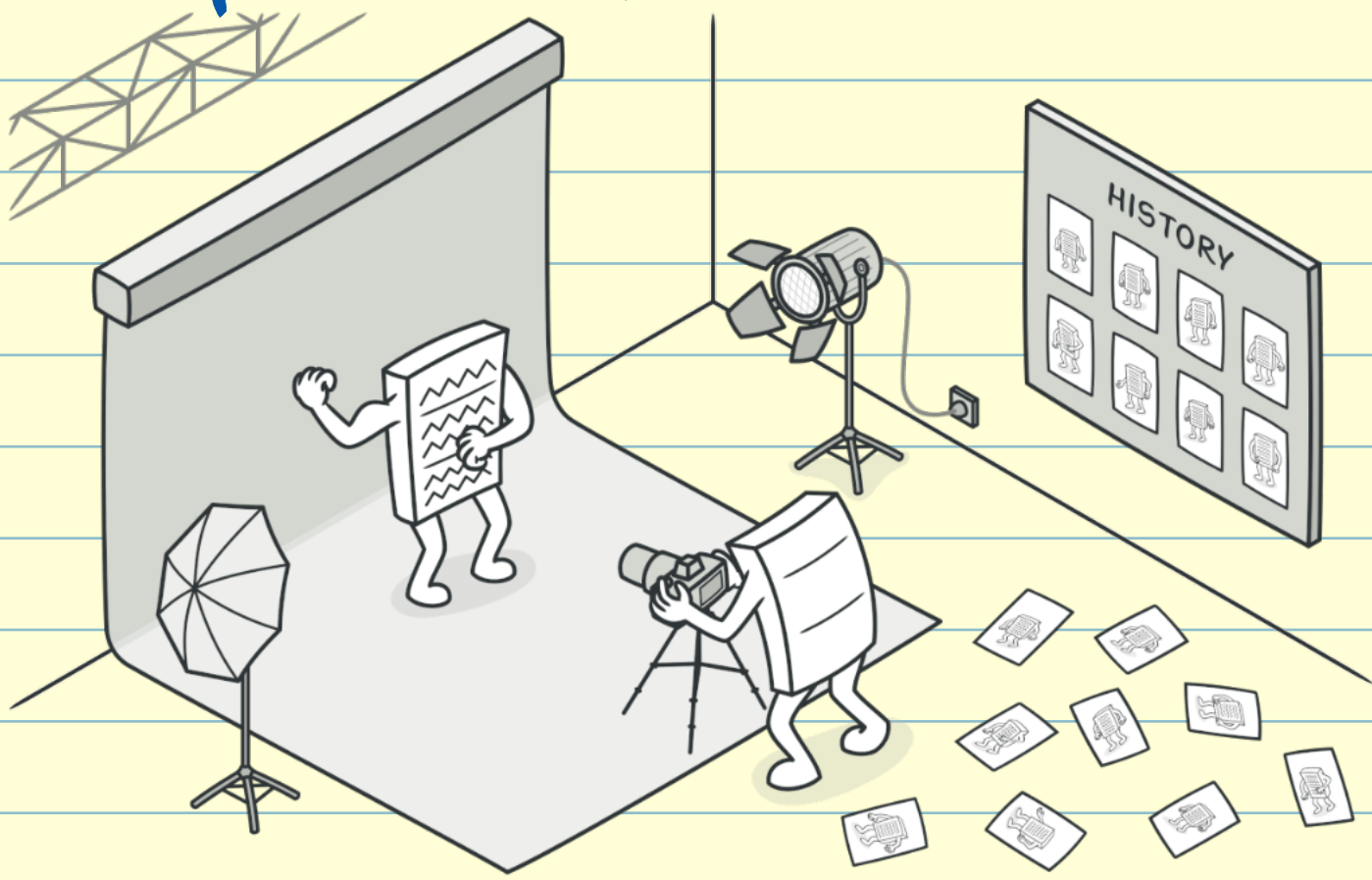
declares methods of communication with components, which usually include single notification method

to mediator.



Memento / Snapshot DP

- let's you save and restore the previous state of an object without revealing the details of its implementation.



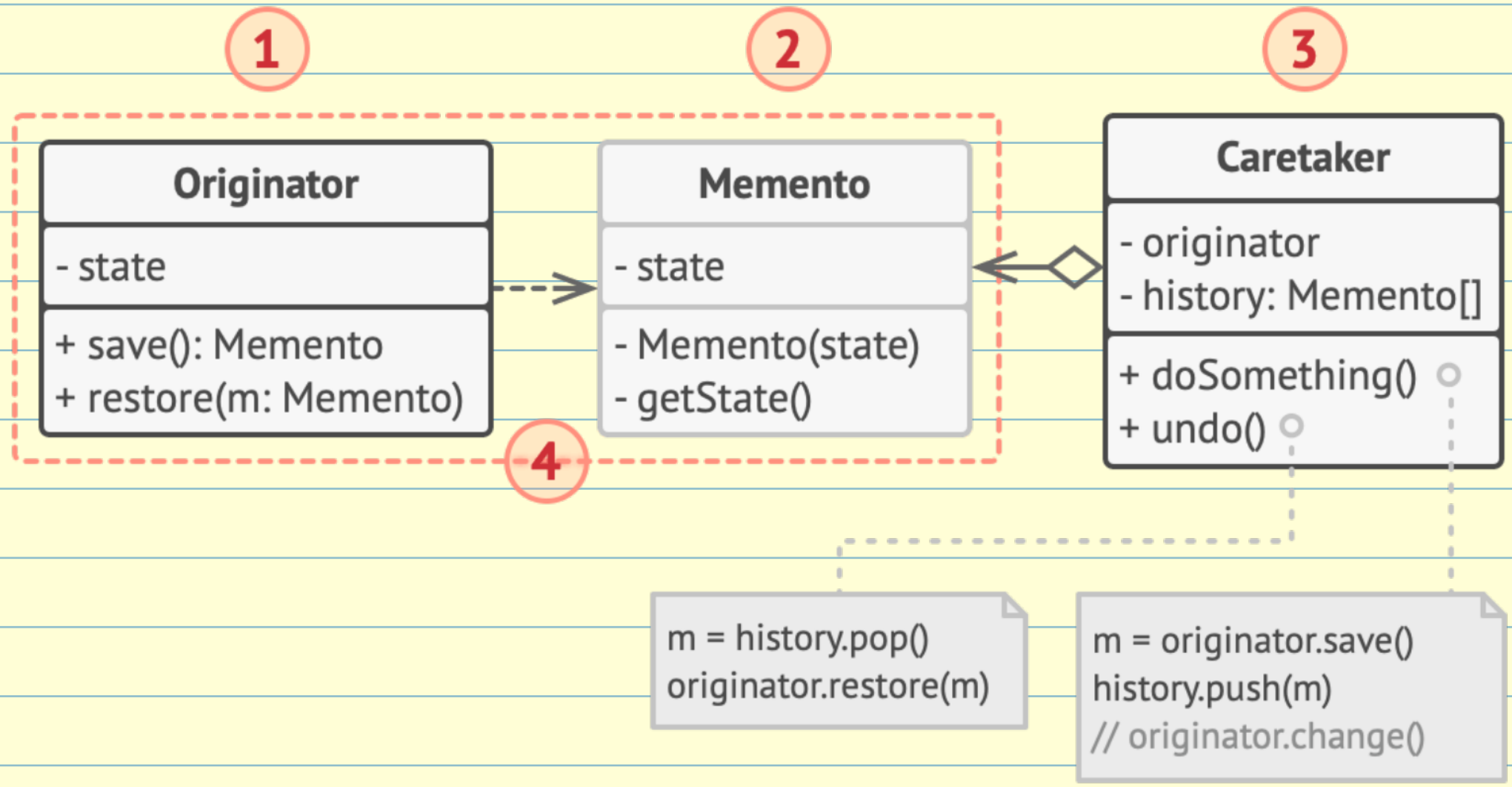
Why required & when to use?

- provides an ability to revert an object to a previous state.
  i.e. UNDO capability.
- it does not expose the object internal implementation.

Originator
- it is the object that state need to be saved and restored.
- expose methods to save and restore its state using Memento object.

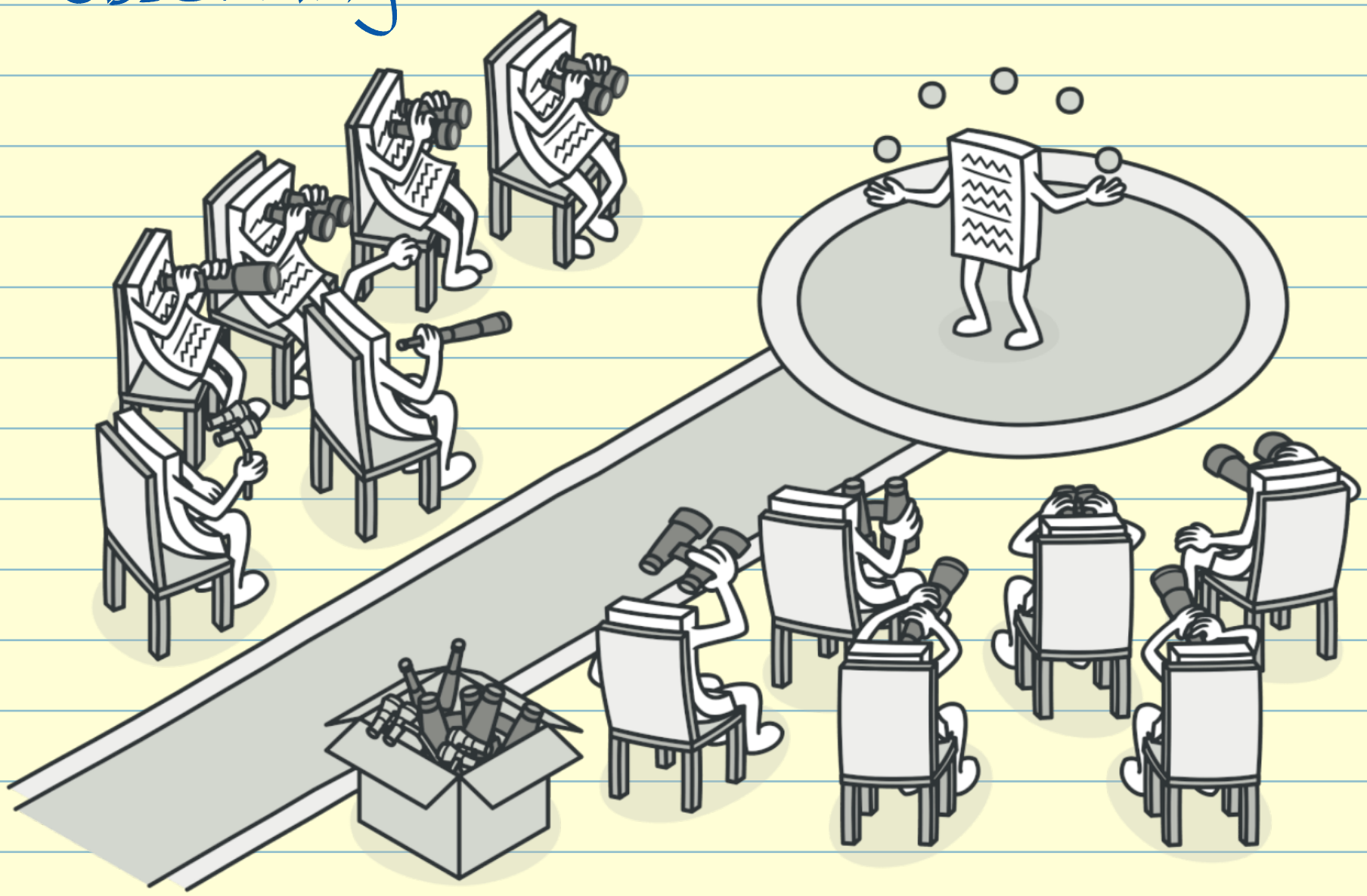
Memento
- it represent an object which holds the state of the originator

Caretaker
- manage a list of mementos.

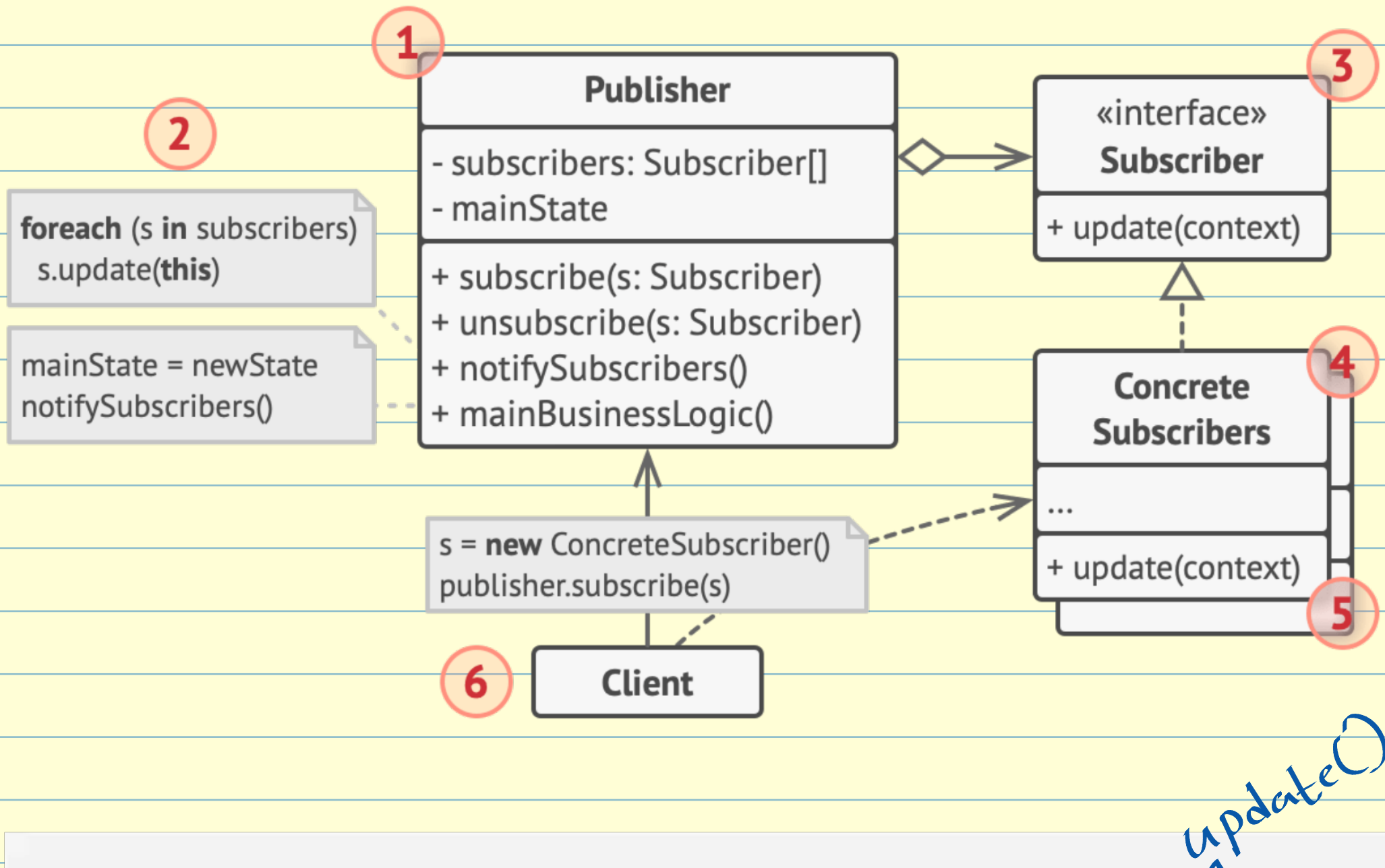


Observer

- lets you define a subscription method to notify multiple objects about any events that happen to the object they are observing.

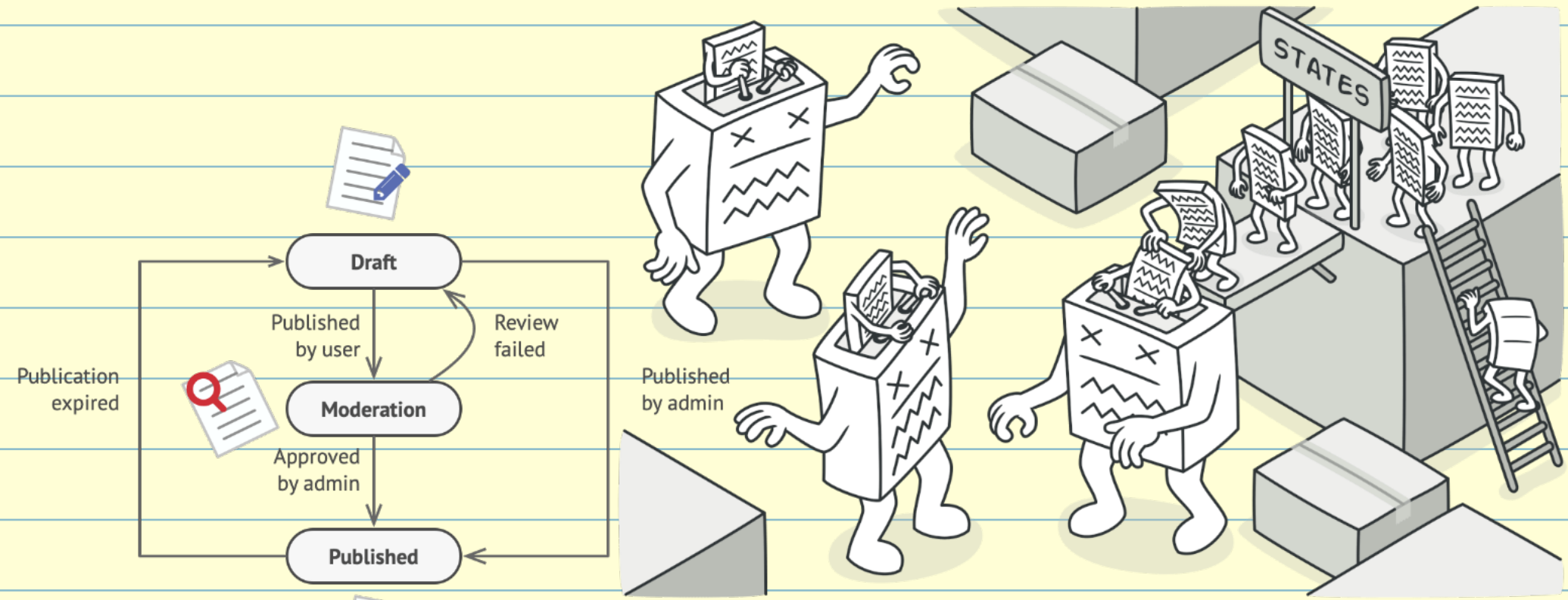


- the subject object automatically notifies the dependent observer objects whenever it's state change by calling those methods.

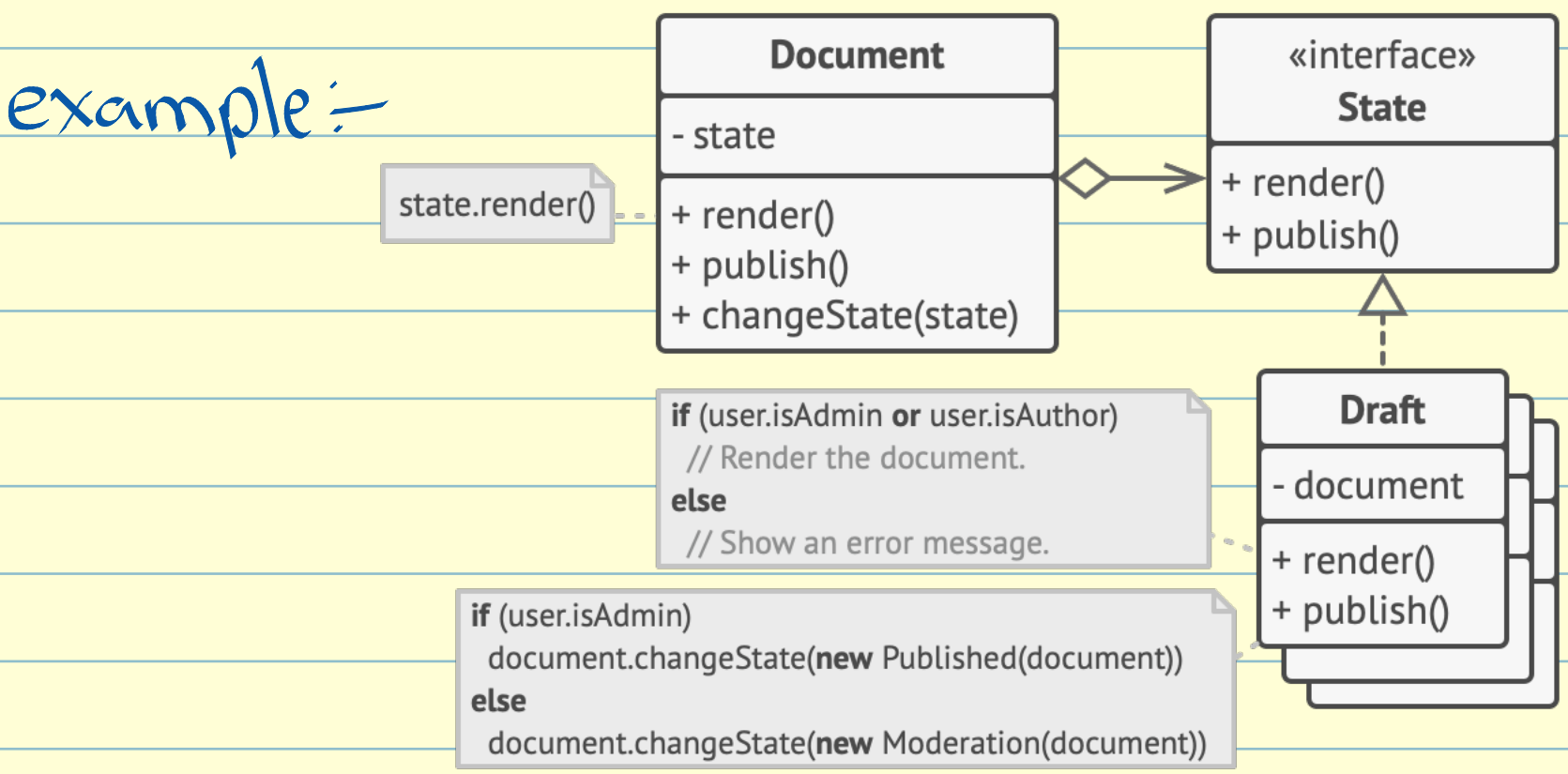

1
Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
2
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
3
«interface»
Subscriber
+ update(context)
4
Concrete
Subscribers
...
+ update(context)
5
s = new ConcreteSubscriber()
publisher.subscribe(s)
6
Client
update()
Class Diagram of Observer Design Pattern
<<interface>>
Subject
+addObserver()
+removeObserver()
+notifyObservers()
1
*
<<interface>>
Observer
+update(weather)
Subject
Observer
WeatherStation
-observers: List
+weather: String
+addObserver()
+removeObserver()
+notifyObservers()
PhoneDisplay
-weather: String
+update()
+display()
TVDisplay
-weather: String
+update()
+display()
Concrete
Subject
Concrete
Observer

State Method

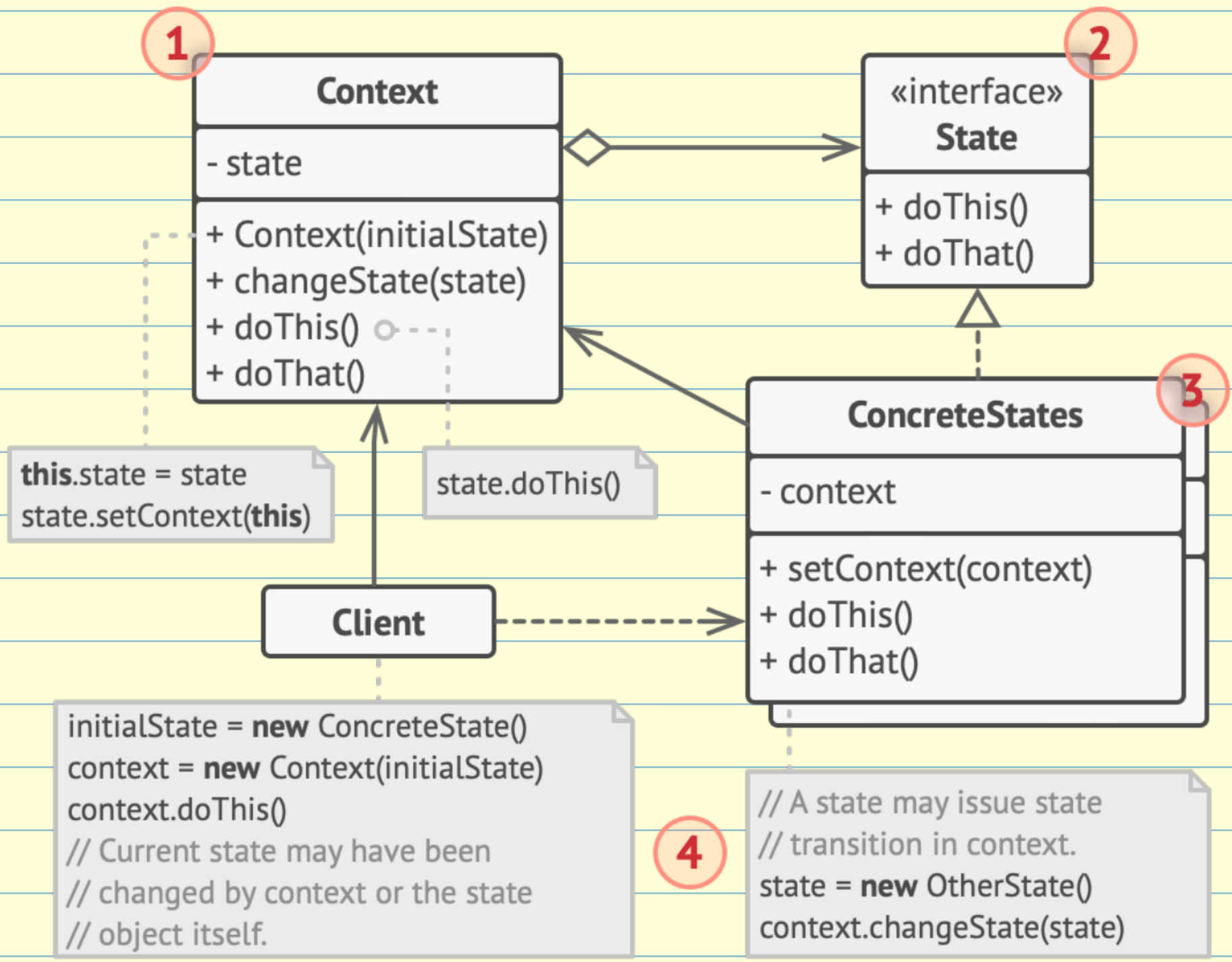
- lets an object alter his behavior when its internal state changes. It appears as if the object changed its class.



example :-



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-code changes frequently.

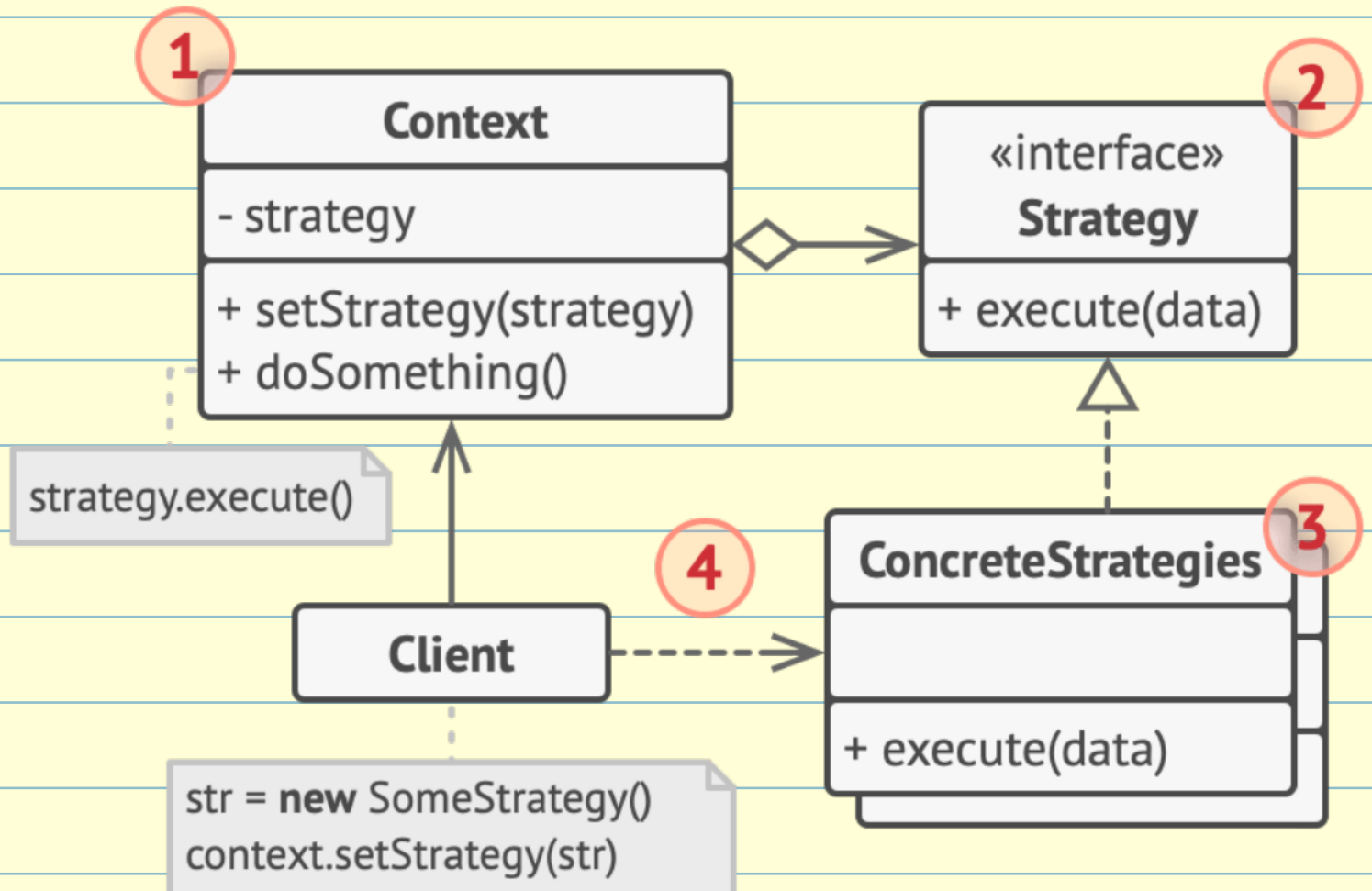


Strategy

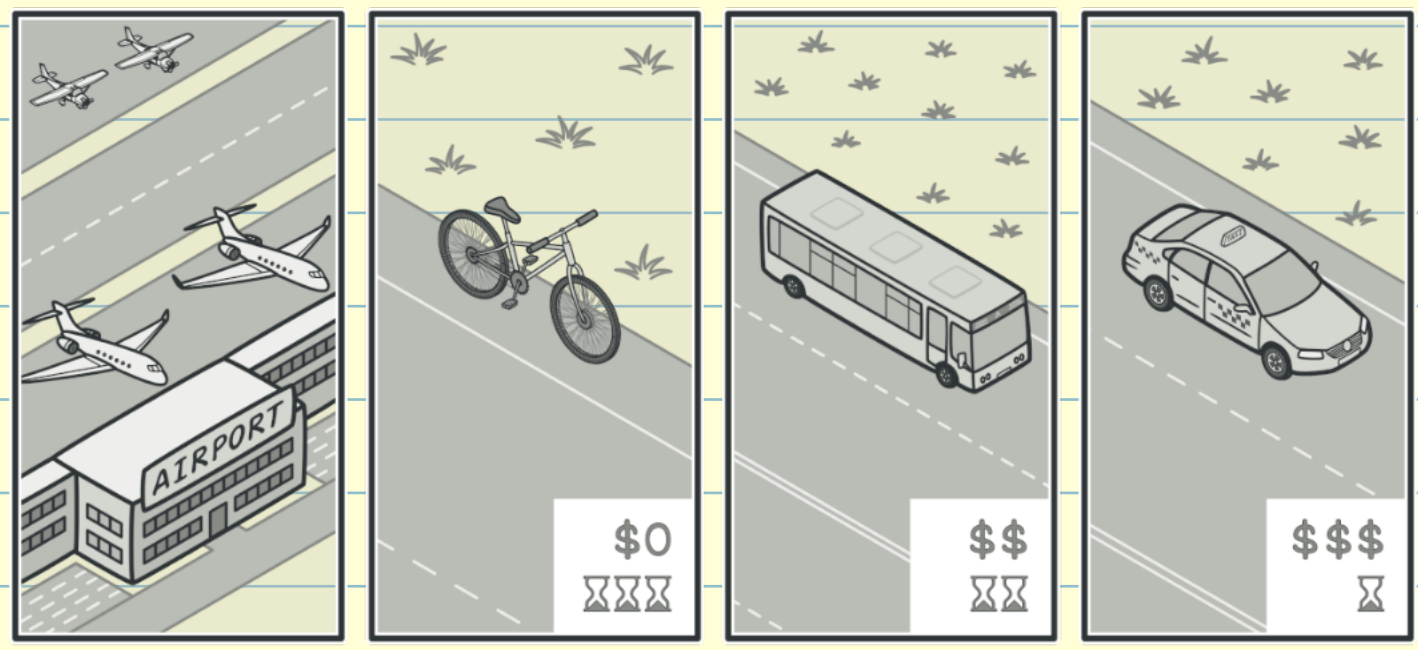
lets you define a family algorithms, put each of them into a seperate class, and make their objects interchangeable.

ex:- different sorting algorithm example

*use strategy pattern when you have a lot of similarly classes that only differ in the way they execute some behavior.

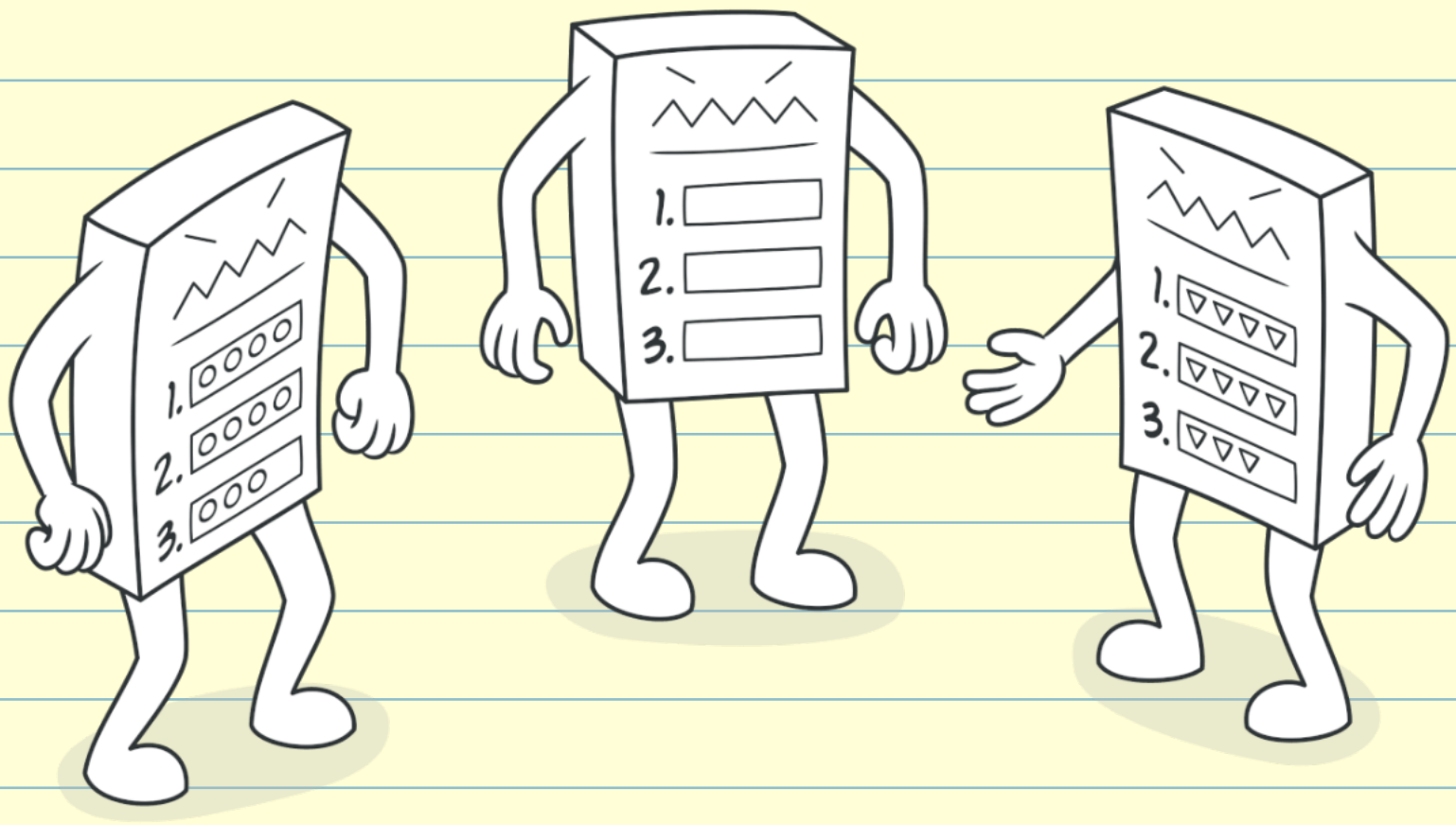


ex:- navigation app



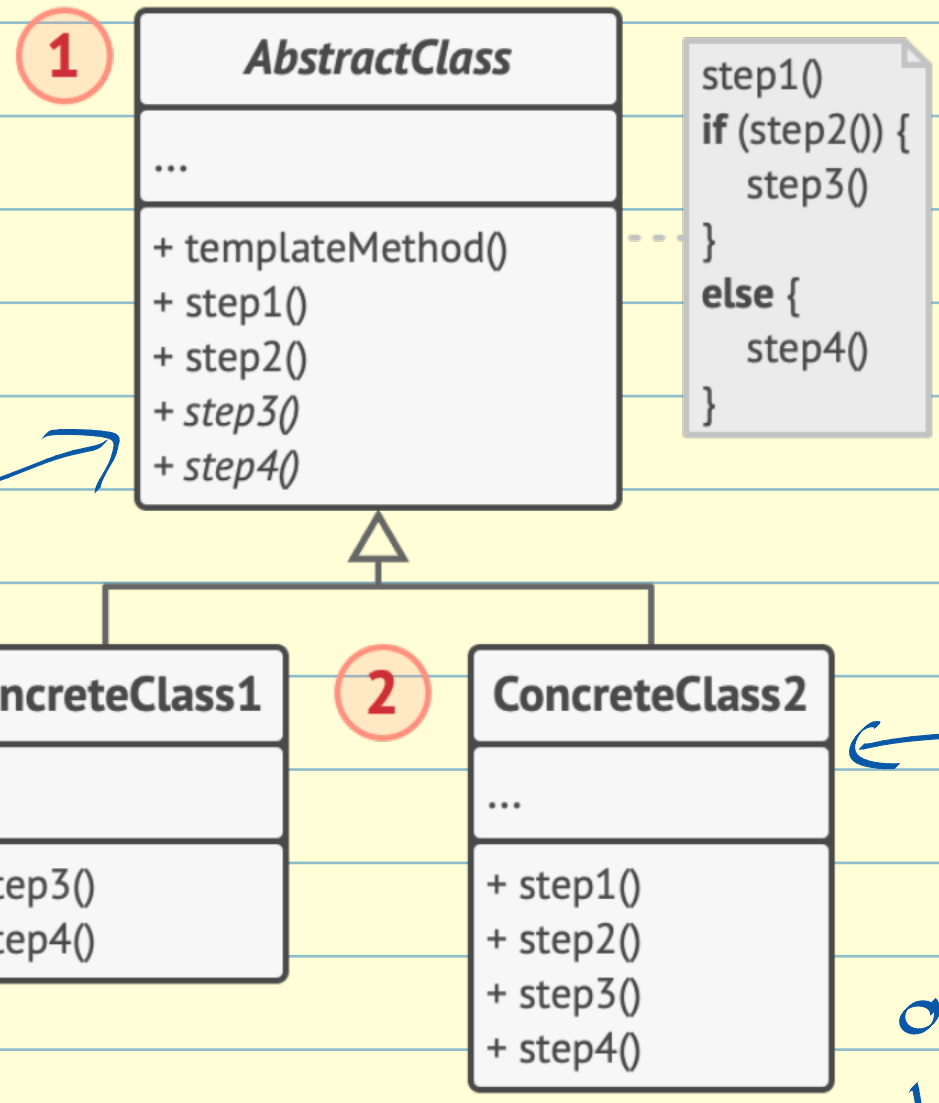
Template method

- defines the skelton of an algorithm in the superclass, but lets subclasses overide specific steps of the algorithm without changing its structure.



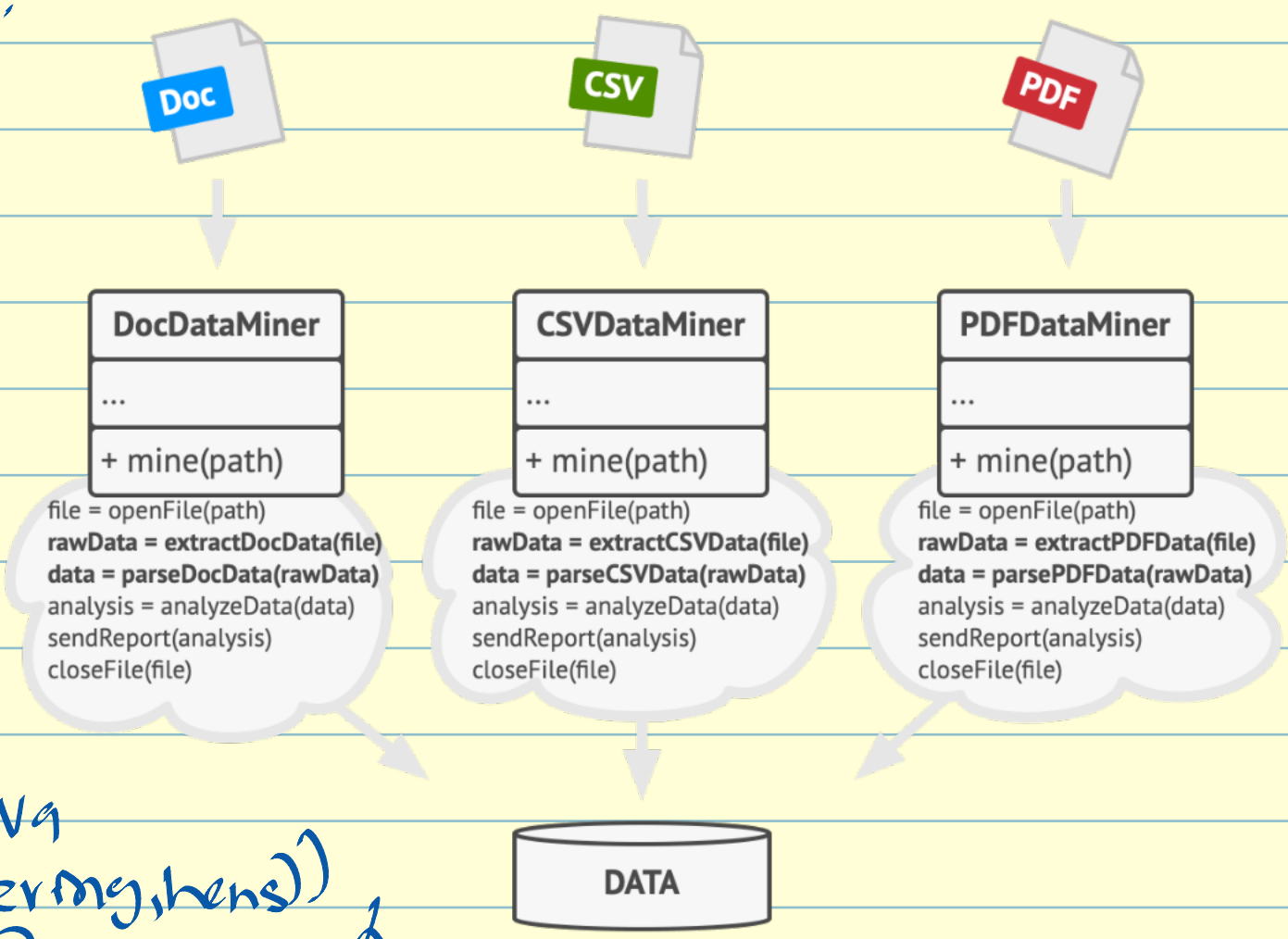
- factory method is a specialization of template method.

- use the template method when you only need to let clients extend only paticular steps of an algorithm, but not the whole algorithm.



← concrete classes can override all of the steps, but not the template method itself.

The abstract class declares methods that act as steps of an algorithm.



Dinara De Silva
(B.Sc (Engineering)Hons)
(US)