Topic 7 - Behavioural Design Patterns

References:

Design Patterns Elements of Reusable Object-Oriented Software

Java Design Patterns A Tutorial

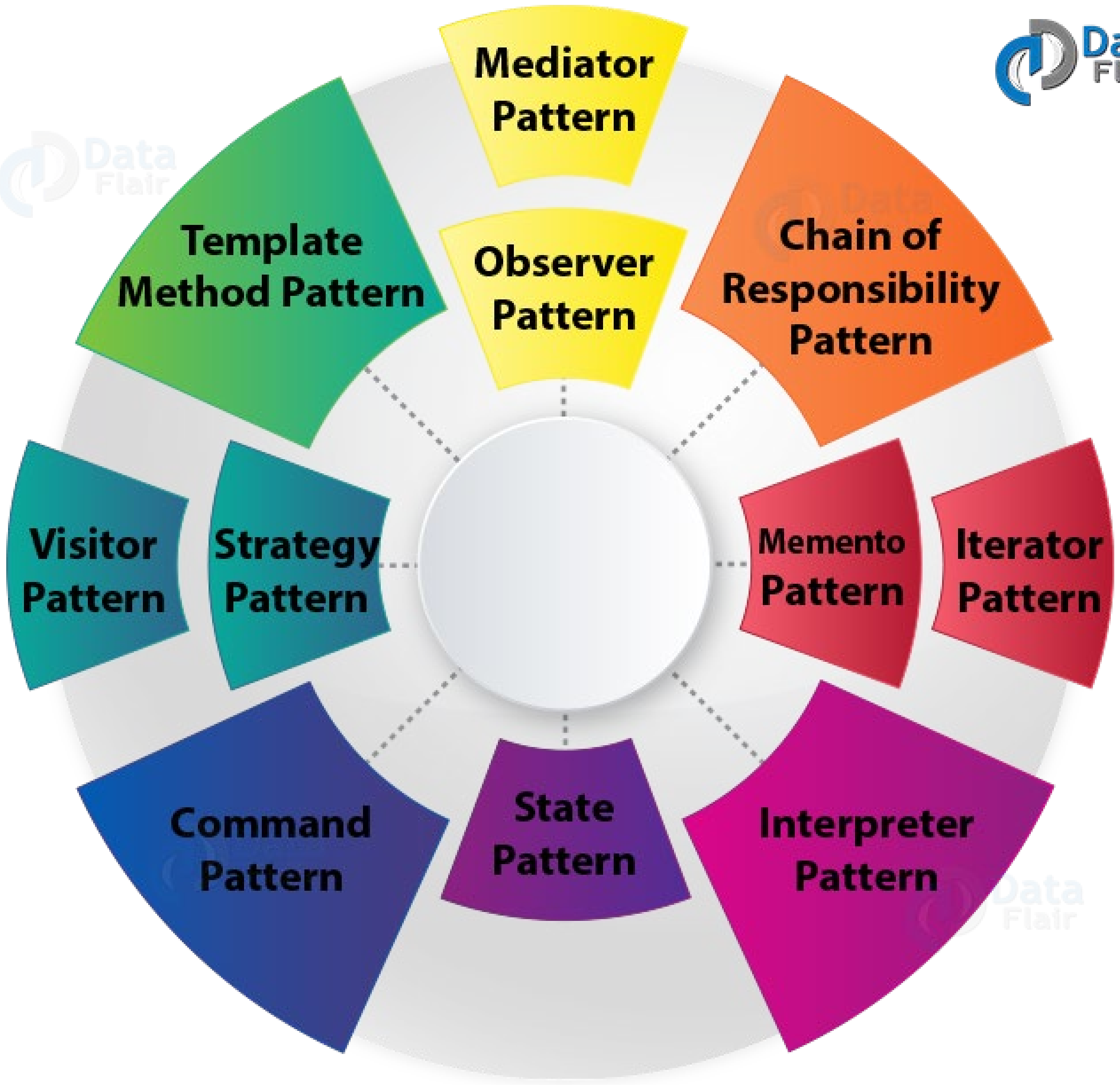
Head First Design Patterns

Mira Balaban Design Patterns

Design Patterns Explained

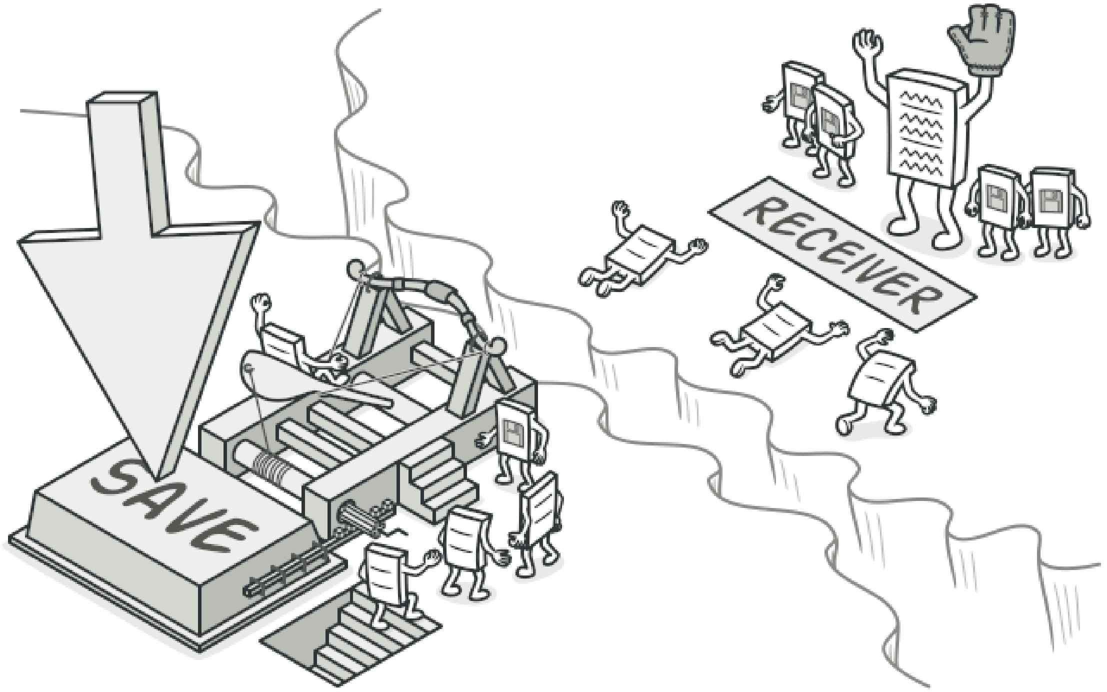
Behavioral Patterns

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects
- Behavioral class patterns use inheritance to distribute behavior between classes
- Behavioral object patterns use composition instead



Command: Motivation

- A user interface toolkit includes objects like buttons and menus that carry out a request in response to user input.
- The operation triggered by a request is application dependent: Performed by **Domain Layer** objects.
- The toolkit does not know the receiver of the request or the operations that will carry it out.



Command: Solution

- make requests/behaviours on unspecified application objects by turning the behaviour itself into an object.
- Key abstraction: an abstract *Command* class or interface.
  - Declares an interface for executing operations.
  - The *Command* interface includes an abstract *execute* operation.
- *Concrete Command* subclasses specify a receiver-action pair by storing the receiver, and by implementing *execute*.
  - The receiver has the knowledge required to carry out the request.

Command: Intent and context

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Apply *Command* for the following purposes:
  - parameterize objects by an action to perform.
  - execute same command from multiple contexts (menu, button etc.)
  - specify, queue, and execute requests at different times.
    - A *Command* object can have a lifetime independent of the original request.
  - support undo/redo and logging changes.
    - The *execute* operation can store state for reversing its effects.
    - can keep a persistent log of changes.
  - Support transactions:
    - structure a system around high-level operations built on primitive operations.

Command: Class diagram

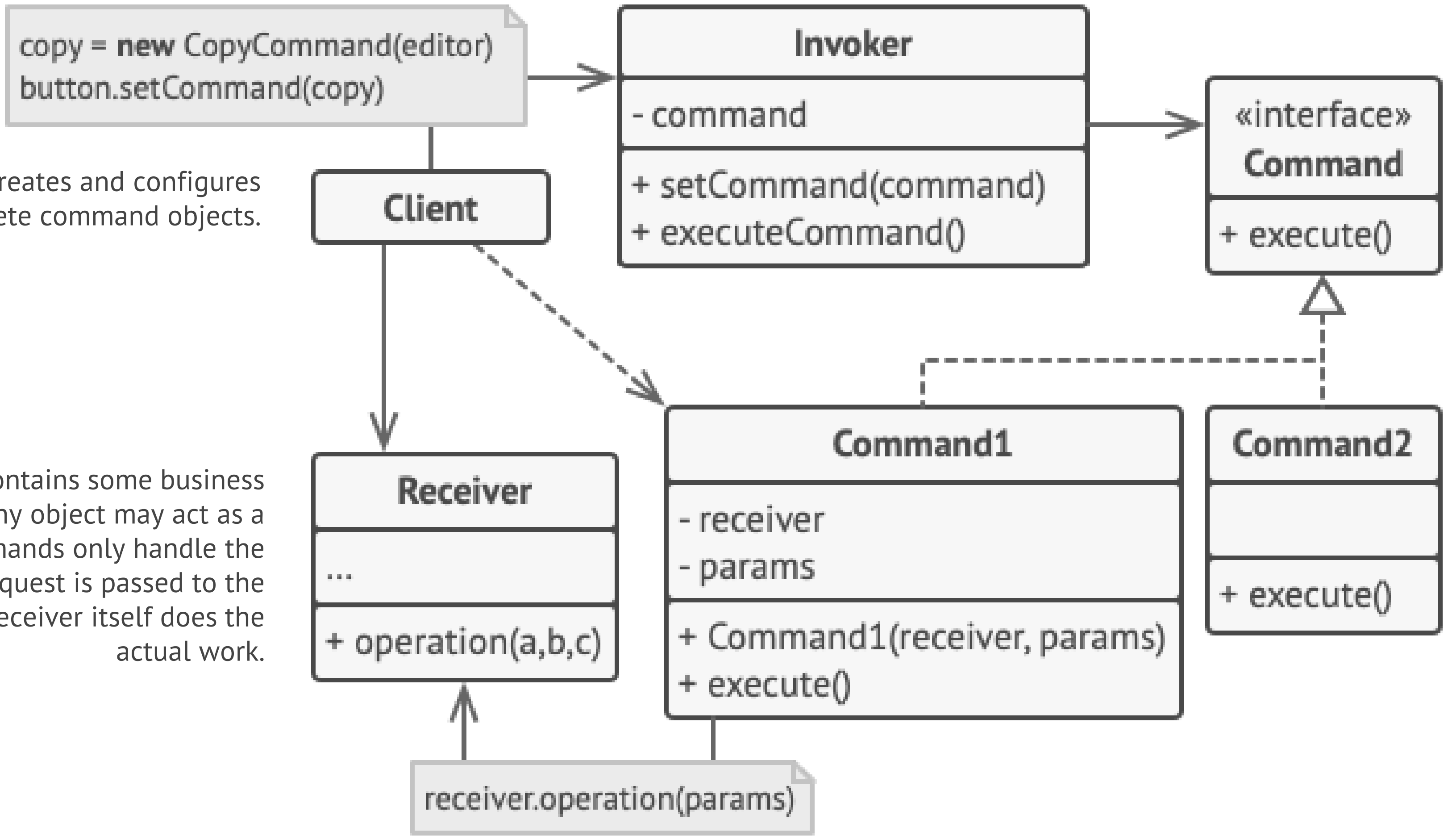
1.The **Sender** class (aka *invoker*) is responsible for initiating requests. This class must have a field for storing a reference to a command object. The sender triggers that command instead of sending the request directly to the receiver

The **Client** creates and configures concrete command objects.

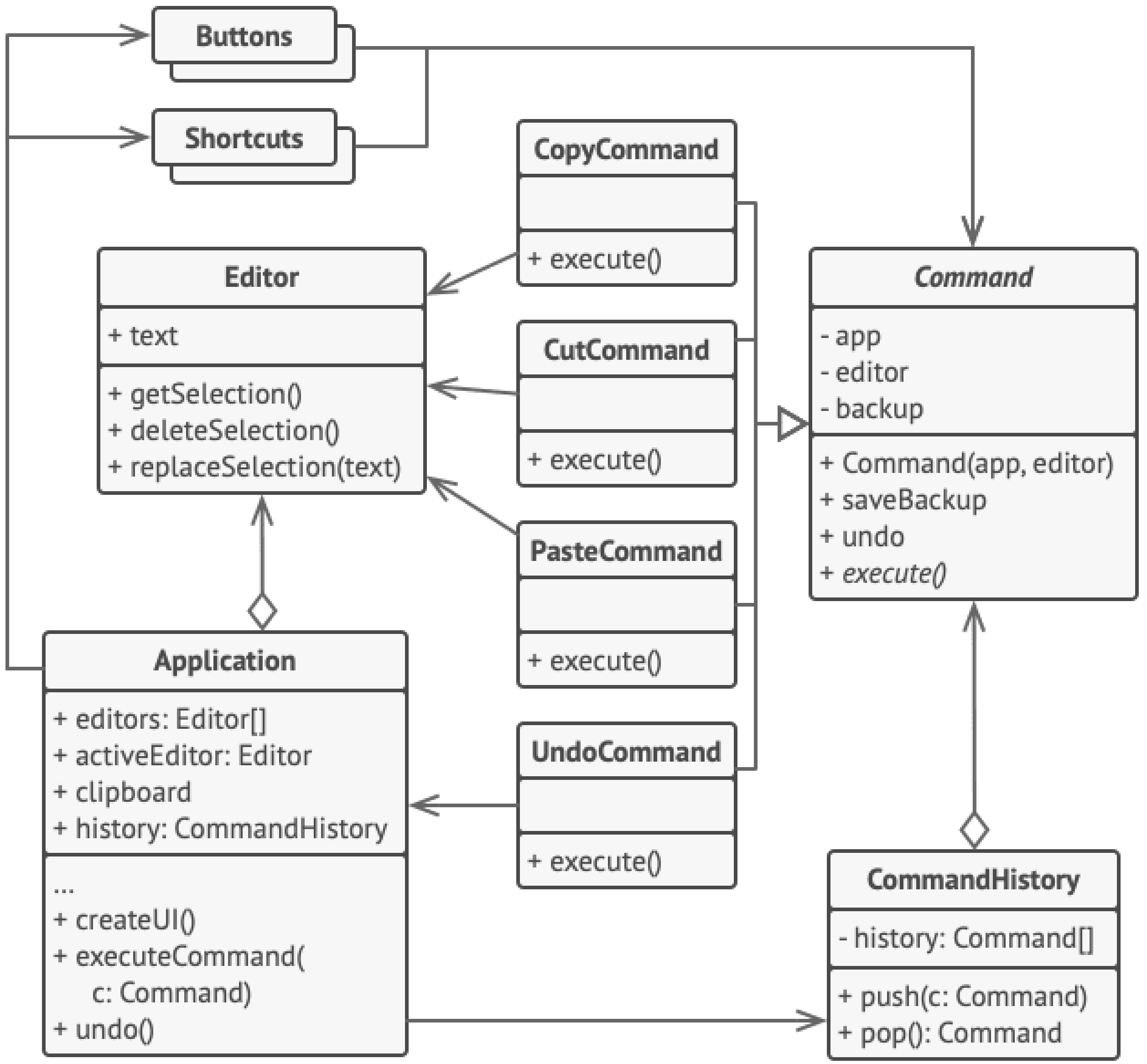
2. The **Command** interface usually declares just a single method for executing the command.

The **Receiver** class contains some business logic. Almost any object may act as a receiver. Most commands only handle the details of how a request is passed to the receiver, while the receiver itself does the actual work.

Concrete Commands implement various kinds of requests. A concrete command isn't supposed to perform the work on its own, but rather to pass the call to one of the business logic objects.



Command: Scenario



Command: Participants

- *Command*
  - declares an interface for executing an operation.
- *ConcreteCommand* (PasteCommand, OpenCommand)
  - defines a binding between a *Receiver* object and an action.
  - implements *execute* by invoking the corresponding operation(s) on *Receiver*.
- *Client* (Application)
  - creates a *ConcreteCommand* object and sets its receiver.
- *Invoker* (MenuItem)
  - asks the command to carry out the request.
- *Receiver* (Document, Application)
  - knows how to perform the operations associated with carrying out a request. Any class may serve as a *Receiver*.
- `java.lang.Runnable or javax.swing.Action/java.awt.event.ActionListener` for basic Java API examples

Providing an Undo

- A benefit of using the Command pattern is that it provides a convenient way to execute an undo.
- Each command object can be made to remember what it did and restore the previous state (as long as computational and memory requirements are reasonable).
- The command interface can be redefined as in:

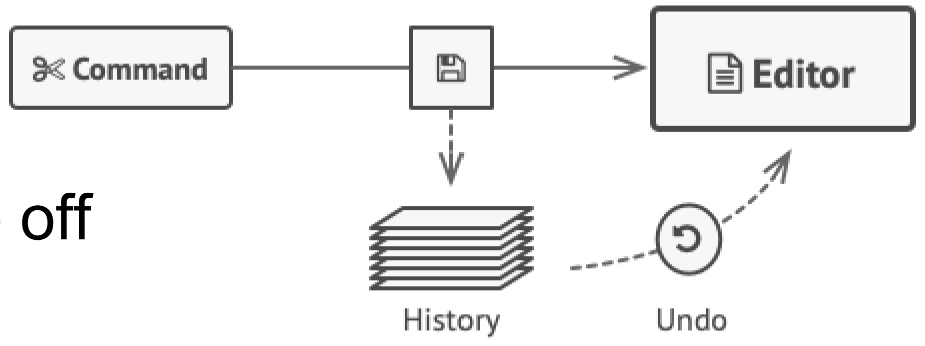
```
public interface Command
{
    public void execute()
    public void undo();
}
```

- To allow undo command, either
  - object must be made to record the previous actions/**state** or provide an **algorithm** to apply the reverse action
  - use a history manager that performs the undo
  - generally must undo in opposite order of execution
  - see example `command.undo.*;`

Command: Consequences

- Decouples the object that invokes the operation from the one that knows how to perform it.
- *Command*s are first-class objects. They can be manipulated and extended like any other object.
- *Command*s can be assembled into a composite command.
  - *Composite command*s are an instance of the *Composite* pattern. (See MacroCommand in class diagram)
- Easy to add new *Command*s: no need to change existing classes.

Memento: Motivation



- If undo operations are provided the internal state off dependent objects must be recorded.
- Without a memento the originating class much either:
  - manage its own state storage/retrieval (which reduces cohesion)
  - or expose its state so someone else can manage it (which reduces encapsulation/information hiding)
  - may lead to code duplication if different objects require state/undo management or excessive coupling to an external history manager

https://refactoring.guru/design-patterns/memento

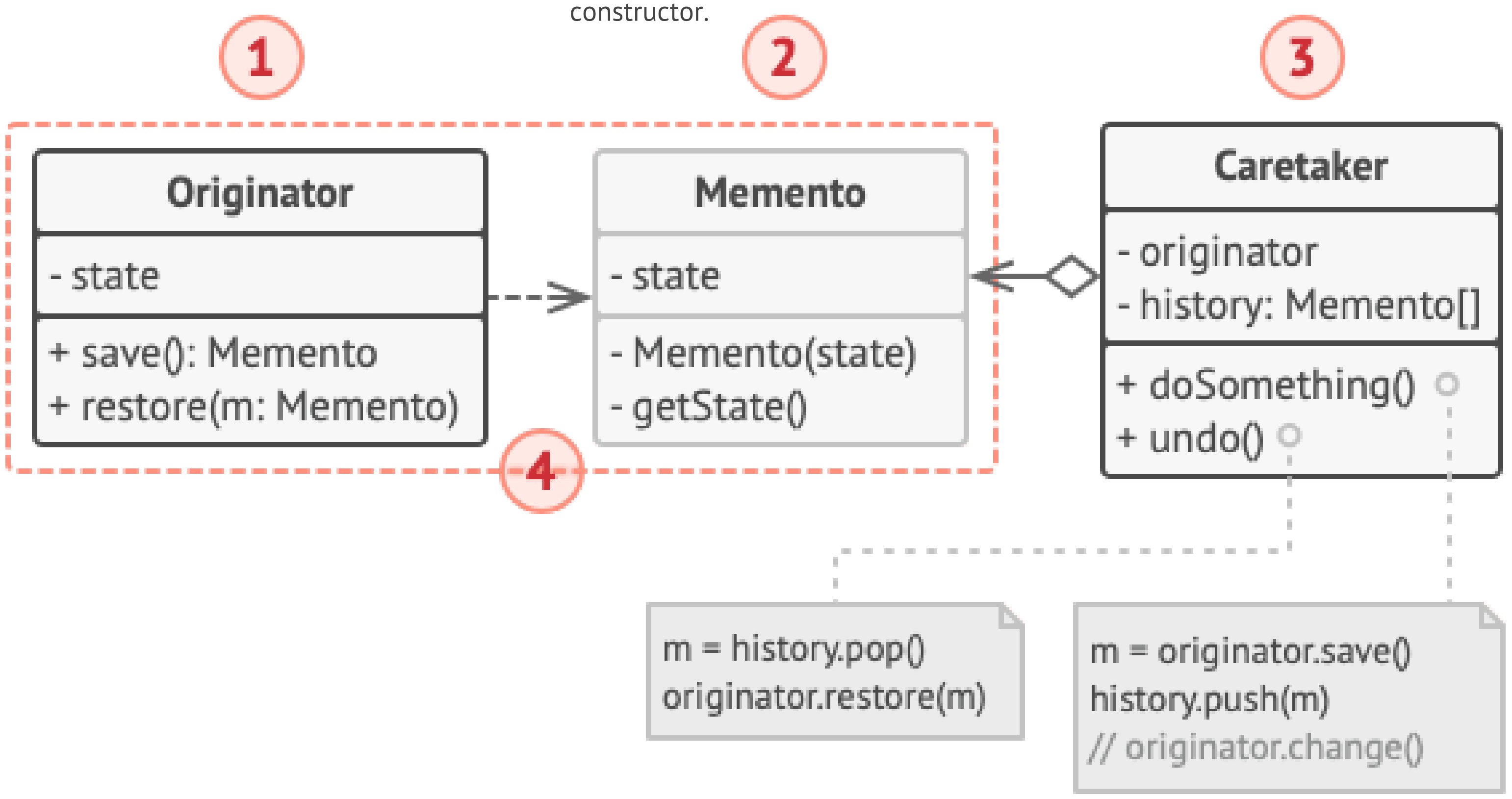
Memento: Participants

- Memento
  - Stores internal state of the Originator object – the amount of information stored is determined by the originator
  - Protects against access by objects other than Originator
- Originator
  - Creates a memento containing a snapshot of its current state
  - Uses the memento to restore internal state
- Caretaker
  - Is responsible for memento's management i.e. when to create, rollback etc.
  - is responsible for managing history
  - The state is a copy so protects encapsulation of the Originator

Memento: Structure & Collaboration

The **Memento** is a value object that acts as a snapshot of the originator's state. It's a common practice to make the memento immutable and pass it the data only once, via the constructor.

1.The **Originator** class can produce snapshots of its own state, as well as restore its state from snapshots when needed.



The **Caretaker** knows not only "when" and "why" to capture the originator's state, but also when the state should be restored.

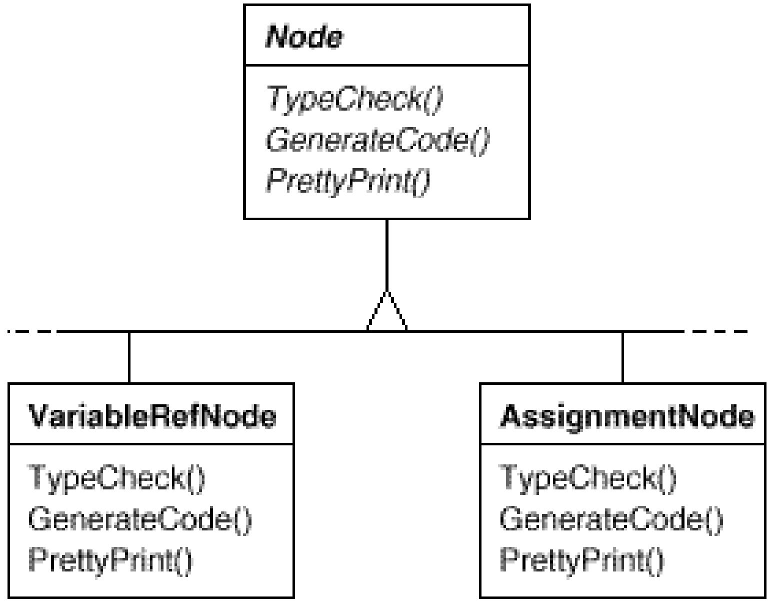
1.In this implementation, the memento class is nested inside the originator. This lets the originator access the fields and methods of the memento, even though they're declared private. On the other hand, the caretaker has very limited access to the memento's fields and methods, which lets it store mementos in a stack but not tamper with their state.

Memento: Consequences

- This is an alternative to the command pattern where commands are responsible for storing undo state
- Could also be used in conjunction with command where commands use a memento as part of the undo process
- The *CareTaker* may save state to persistant storage
- see `memento.simple.*;` example

Visitor: Example Scenario

- A compiler that represents a program as abstract syntax tree.
- The set of node classes is fixed for a language.



- Applies operations like *type-checking, code optimization, flow analysis, checking for variables being assigned values before they're used* to all nodes.
- Operations might change – depends on static semantic analysis.

Visitor: Problem

- Distributing all operations across node classes can reduce cohesion and lead to a system that's hard to understand, maintain, and change.
  - e.g. type-checking code is mixed with pretty-printing code or flow analysis code.
- Adding a new operation requires modifying and recompiling the entire node class hierarchy.
- Since operations change more often than the node class structure it would be better if:
  - each new operation could be added separately.
  - the node class structure was independent of the operations that apply to it.

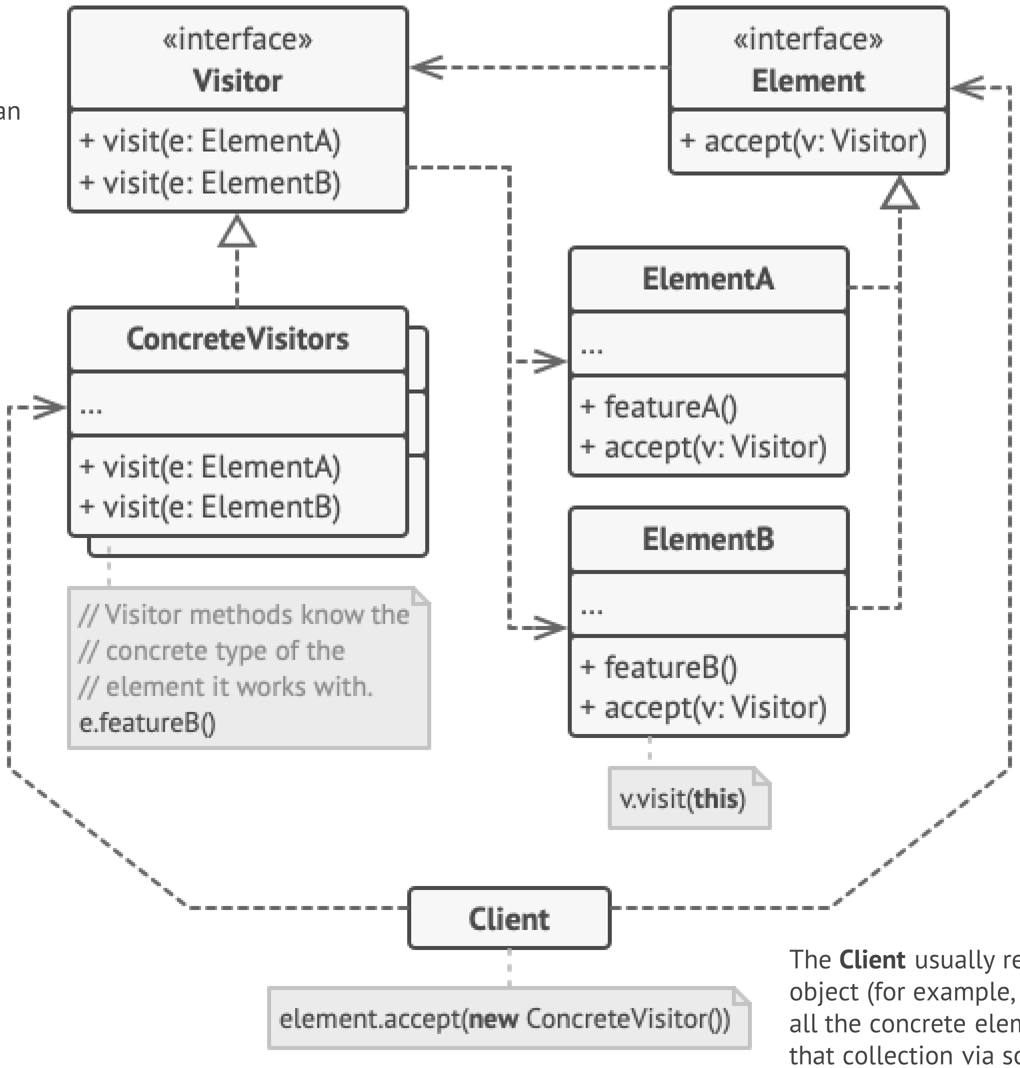
Visitor: Solution

- Package related operations (for different subclasses) in a separate object, called a ***visitor***.
- Pass visitor to elements of the class hierarchy (e.g. nodes in abstract syntax tree) as they are traversed.
- When an element "accepts a *visitor* ", it calls an operation on the *visitor* (often called **visit()**), that includes the element as an argument.
- The *visitor* executes the operation on behalf of the parameter(object) it received i.e. it visits the parameter.
- Operation code has been removed from the element to make it simpler and more cohesive.
- Adding new operations requires only the addition of a new visitor (sub)class.

Visitor: Class Diagram

The **Visitor** interface declares a set of visiting methods that can take concrete elements of an object structure as arguments.

Each **Concrete Visitor** implements several versions of the same behaviors, tailored for different concrete element classes.



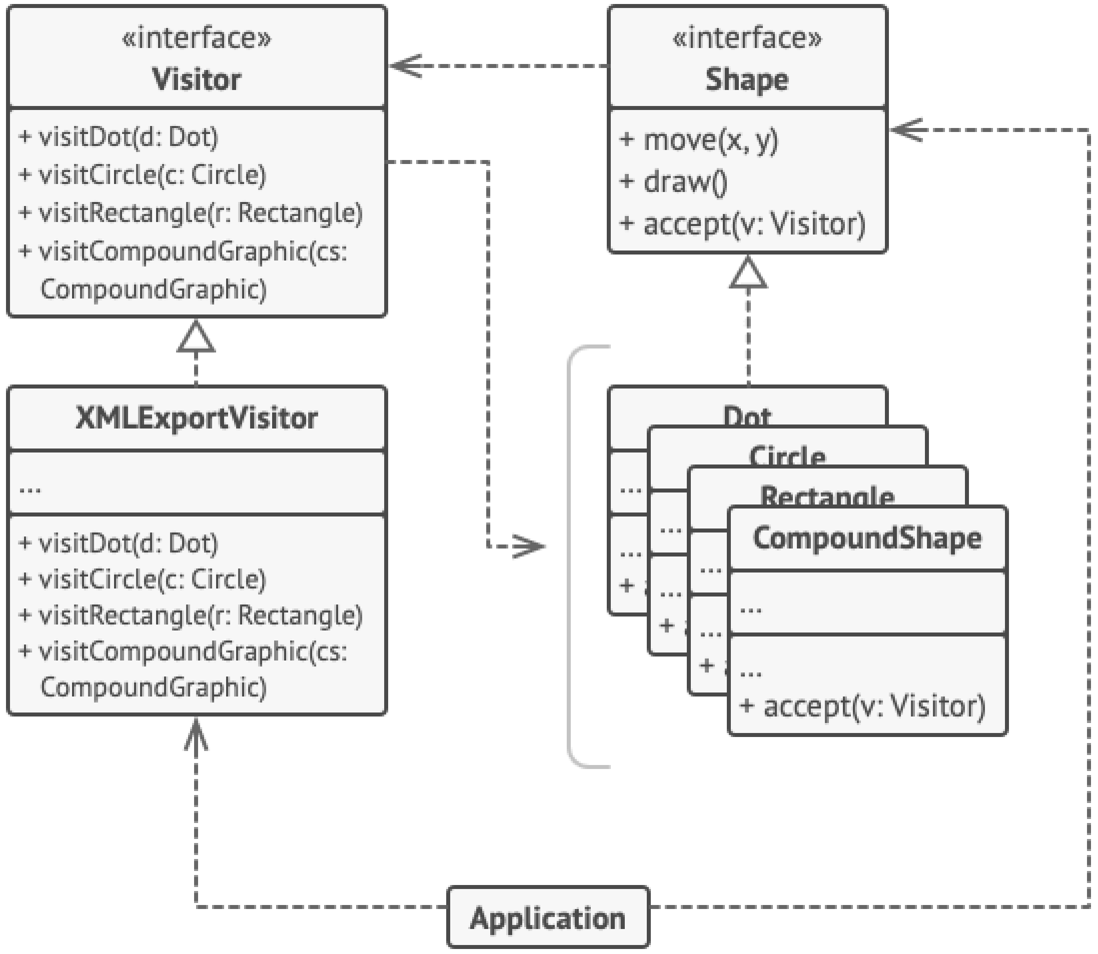
The **Element** interface declares a method for "accepting" visitors. This method should have one parameter declared with the type of the visitor interface.

Each **Concrete Element** must implement the acceptance method. The purpose of this method is to redirect the call to the proper visitor's method corresponding to the current element class.

The **Client** usually represents a collection or some other complex object (for example, a **Composite** tree). Usually, clients aren't aware of all the concrete element classes because they work with objects from that collection via some abstract interface.

https://refactoring.guru/design-patterns/visitor

Visitor: Specific Example



https://refactoring.guru/design-patterns/visitor

Visitor: Participants

- *Visitor*
  - declares a *visit()* operation for each class of *ConcreteElement* in the object structure
- *ConcreteVisitor*
  - implements each operation declared by *Visitor*
- *Element*
  - defines an operation *accept(Visitor)*
- *ConcreteElement*
  - implements operation *accept(Visitor)*

IMPORTANT: Java, Polymorphism and Binding

- Overloaded methods are not called polymorphically via parameters
  - They are statically bound (i.e. at compile time)
- Polymorphism only occurs on the object type the method is being called on
  - e.g. for a call `a.b(c);`
  - polymorphism only occurs based on the type of `a` given the available overrides of method `b()`
  - parameter `c` is bound at compile time based on the declared type of parameter `c`
- This behaviour is why the visitor is required and is the basis for how it works
  - This behaviour is the same in C++

Visitor: Considerations

- a *visitor* gathers related operations and separates unrelated ones
- adding new operations is easy
- however adding new *ConcreteElement* classes is hard
  - gives rise to new abstract operation on *Visitor*
  - ...and requires implementation in every *ConcreteVisitor*
- *Visitor* not limited to classes in a (sub)hierarchy, can be applied to any collection of classes
  - provided they define *accept()* methods
  - can provide a *Visitable* interface for this purpose

Visitor: Considerations

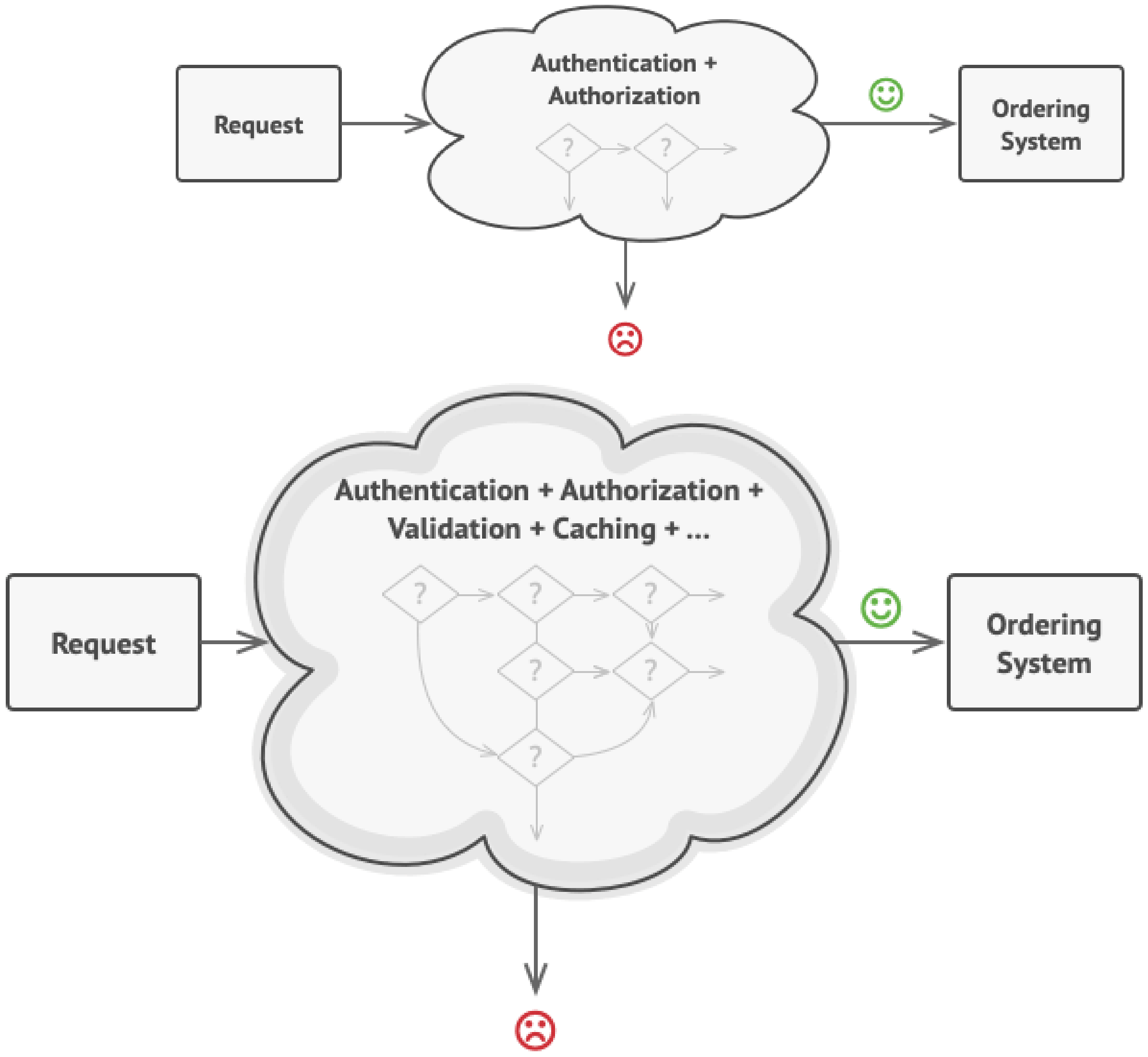
- *Visitor* may require that *ConcreteElement* classes expose state so *Visitor* can do its job
  - breaks encapsulation
  - better to provide operations on the Element
- The Visitor may not be cohesive if the subclasses it implements operations for are very different
- **Is a trade-off since favouring cohesion over other guidelines since it does go against both *Information Expert* and *Feature Envy* principles!**
- For a more complex double dispatch example see `visitor.dbl.pegs.*;`

Visitor: Summary

- Apply *Visitor* when:
  - a hierarchy contains many classes with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
    - avoids *instance of* checks
    - but still better to try to limit diverging interfaces in a class hierarchy
  - many distinct and unrelated operations need to be performed on objects, and you want to avoid polluting their classes with these operations.
  - the classes in the object structure rarely change, but you frequently want to add new operations on the structure.

Chain of Responsibility: Motivation

- Decouple the sender of the request and its receivers by giving multiple objects a chance to handle a request
- The requesting object has no explicit knowledge of how or by whom the request is handled
- Simplifies the client as it does not have to maintain references or deal with chaining
  - increases configurablity/extensibility
- Java exception handling is an example of chain of responsibility design. If no handler, exception is passed to the method caller



https://refactoring.guru/design-patterns/chain-of-responsibility

Chain of Responsibility: Participants

- *Handler*
  - Defines an interface for handling requests
  - (optional) can implement the successor link if abstract class provided
- ConcreteHandler
  - Handles requests (if possible)
  - Otherwise it forwards the request to its successor
- Client
  - Initiates the request to a ConcreteHandler object on the chain

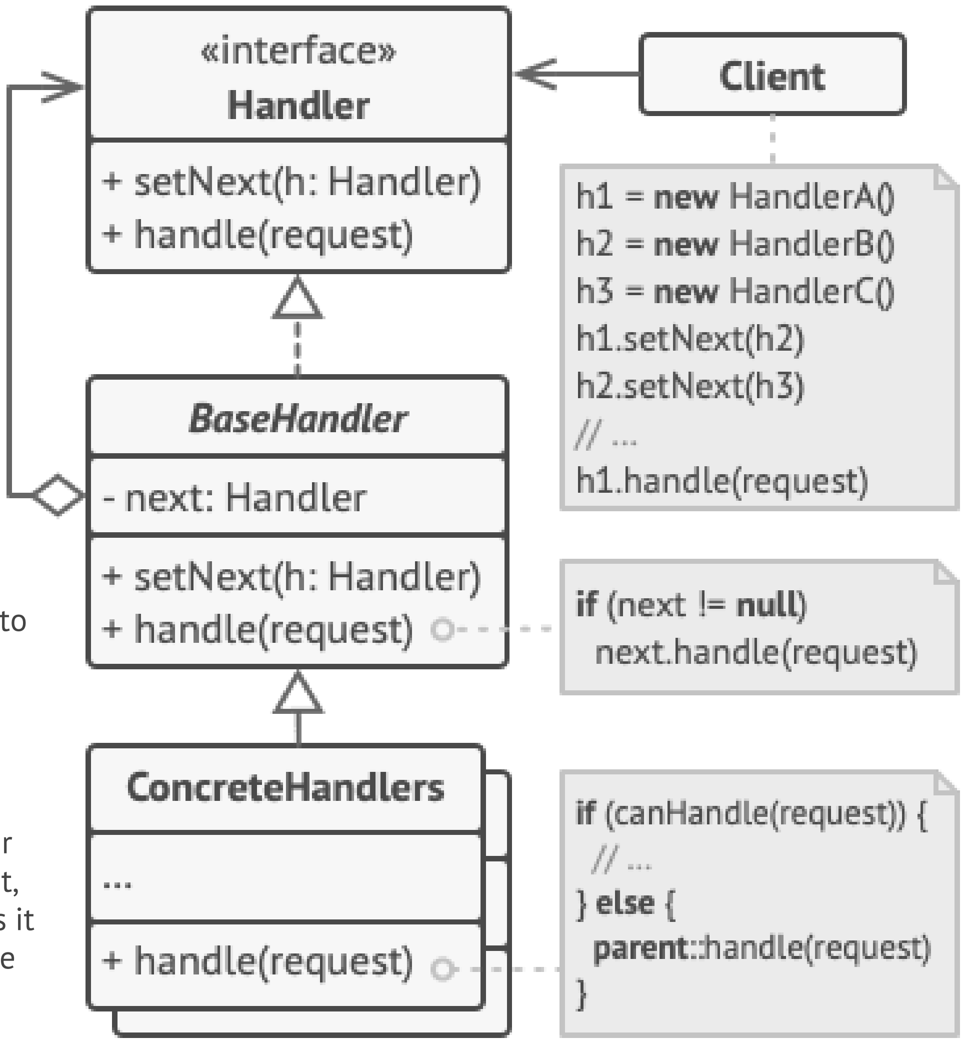
https://refactoring.guru/design-patterns/chain-of-responsibility

Structure

The **Handler** declares the interface, common for all concrete handlers. It usually contains just a single method for handling requests, but sometimes it may also have another method for setting the next handler on the chain.

The **Base Handler** is an optional class where you can put the boilerplate code that's common to all handler classes.

Concrete Handlers contain the actual code for processing requests. Upon receiving a request, each handler must decide whether to process it and, additionally, whether to pass it along the chain.



```
h1 = new HandlerA()
h2 = new HandlerB()
h3 = new HandlerC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
```

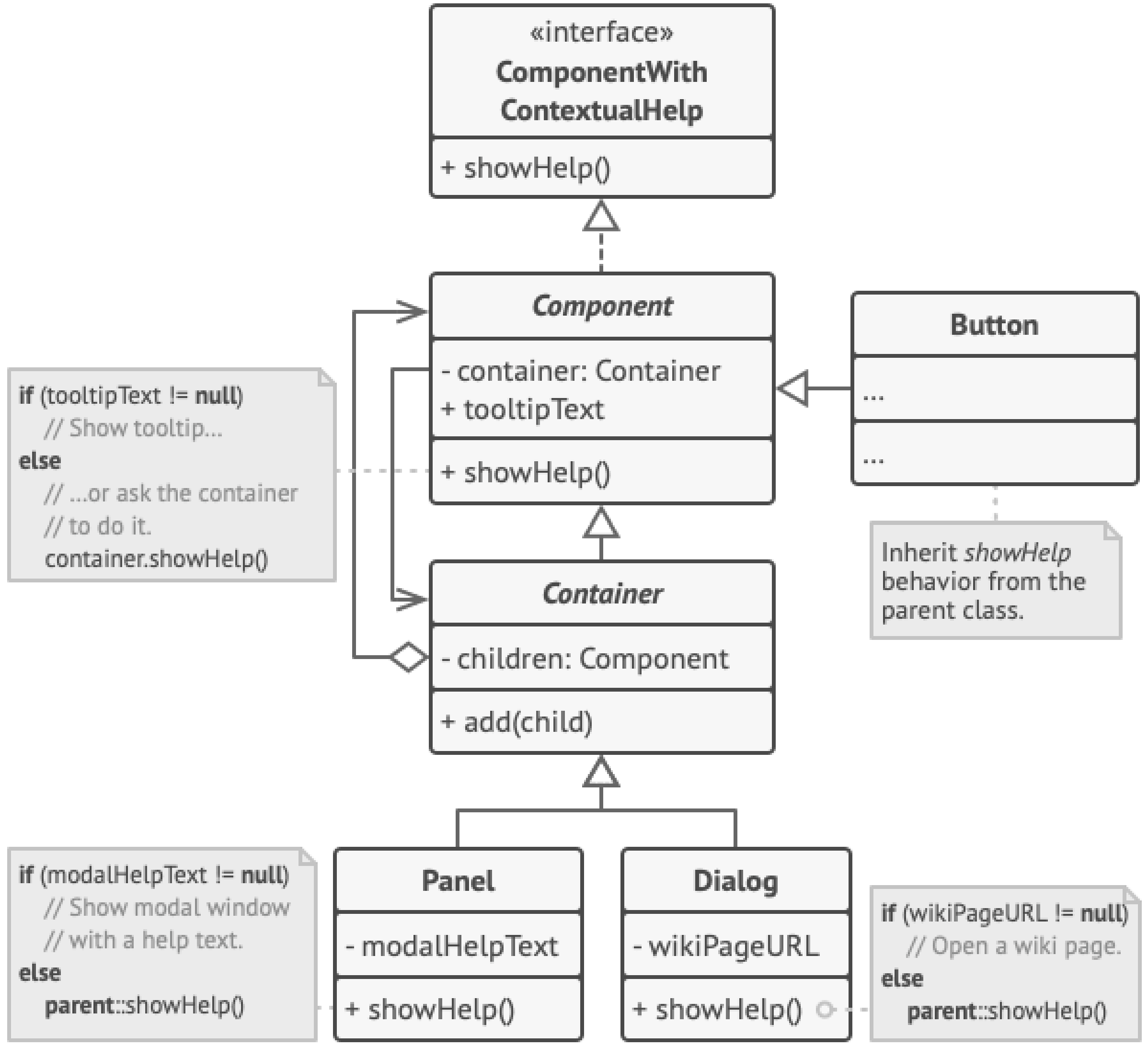
```
if (next != null)
  next.handle(request)
```

```
if (canHandle(request)) {
  // ...
} else {
  parent::handle(request)
}
```

The **Client** may compose chains just once or compose them dynamically, depending on the application's logic. Note that a request can be sent to any handler in the chain—it doesn't have to be the first one.

https://refactoring.guru/design-patterns/chain-of-responsibility

Example



```
if (tooltipText != null)
    // Show tooltip...
else
    // ...or ask the container
    // to do it.
    container.showHelp()
```

```
if (modalHelpText != null)
    // Show modal window
    // with a help text.
else
    parent::showHelp()
```

```
if (wikiPageURL != null)
    // Open a wiki page.
else
    parent::showHelp()
```

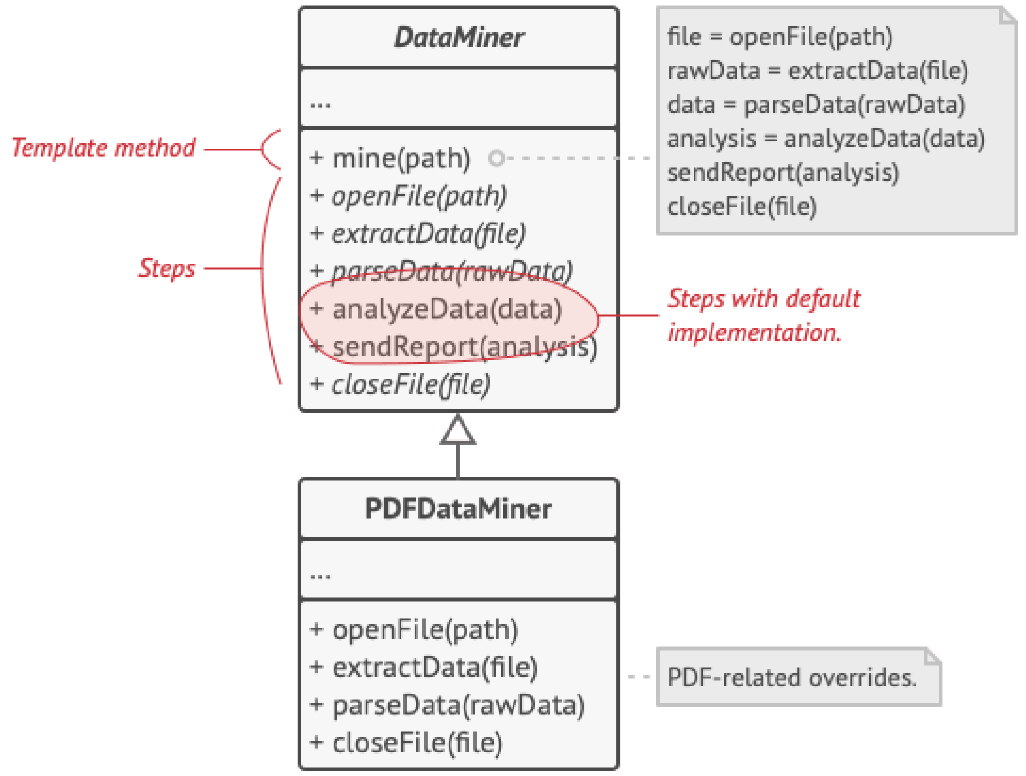
https://refactoring.guru/design-patterns/chain-of-responsibility

Consequences

- Reduced coupling to the client by delegating responsibility for which other object handles a request
- Added flexibility in assigning responsibilities to objects and changing orders etc. depending on context
- Especially useful for caching (e.g. from memory, to disk to the cloud)
  - i.e. try first to get from memory, if not get locally from persistent storage, if not then go to the cloud (or even peer machines etc.)
  - very elegant solution from client perspective who simply calls an operation and increases likelihood of getting a fast response
- Care must taken with ordering of the chain and handling exceptional cases where an operation is not handled

Template Method https://refactoring.guru/design-patterns/template-method

- You have an algorithm that is common to a group of subclasses but one or more parts of the algorithm must be specialised
- Naïve solution is to override the method with duplicate code and only some small changes (see PurchaseHandler. processRequest() overrides!)
- Better solution is to encapsulate the shared algorithm in a templateMethod() within an abstract class
  - can add an interface as well if necessary
- The template method calls abstract primitive methods that are implemented by each subclas
  - through an interface if provided
- see template.* example


DataMiner
...
+ mine(path)
+ openFile(path)
+ extractData(file)
+ parseData(rawData)
+ analyzeData(data)
+ sendReport(analysis)
+ closeFile(file)
file = openFile(path)
rawData = extractData(file)
data = parseData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
Template method
Steps
Steps with default implementation.
PDFDataMiner
...
+ openFile(path)
+ extractData(file)
+ parseData(rawData)
+ closeFile(file)
PDF-related overrides.