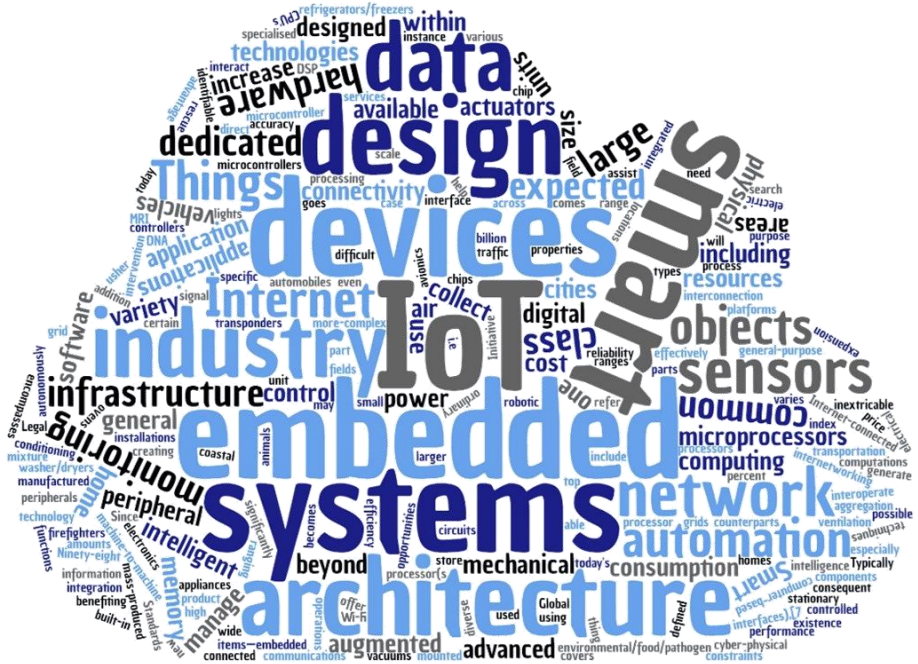
Associate Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Lecture

Software Architecture for Distributed Embedded Systems

03 - Gang of Four Patterns: *Behavioral Patterns*

Prof. Dr. Sebastian Steinhorst



Behavioral Patterns: Context

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
- Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.
- These patterns characterize complex control flow that is difficult to follow at run-time.
- They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Source: GoF book

Behavioral Patterns

Chain of responsibility
A way of passing a request between a chain of objects

Command
Encapsulate a command request as an object

Interpreter
A way to include language elements in a program

Iterator
Sequentially access the elements of a collection

Mediator
Defines simplified communication between classes

Memento
Capture and restore an object's internal state

Observer
A way of notifying change to a number of classes

State
Alter an object's behavior when its state changes

Strategy
Encapsulates an algorithm inside a class

Template method
Defer the exact steps of an algorithm to a subclass

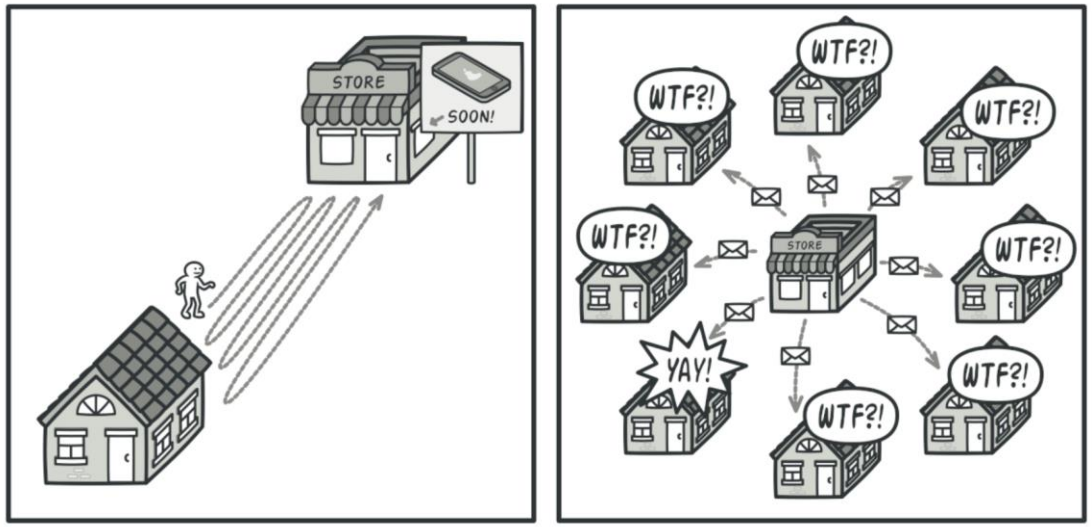
Visitor
Defines a new operation to a class without change

Source: GoF Book, https://sourcemaking.com/design_patterns

Behavioral Pattern: Observer (a.k.a. Dependents, Publish-Subscribe*)

Problem

- A large monolithic design does not scale well as new graphing or monitoring requirements are levied.



Intent

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

- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.

* As we will see in the pattern structure, the observer pattern has similarities to the general publish-subscribe architecture pattern, but does not provide the features of a distributed system publish-subscribe architecture

Source: GoF book, https://sourcemaking.com/design_patterns/observer, https://refactoring.guru/design-patterns/observer

Observer Applicability

- When an abstraction has two aspects, one dependent on the other.
  - Encapsulating these aspects in separate objects lets you vary and reuse them independently.

- When a change to one object requires changing others, and you don't know how many objects need to be changed.

- When an object should be able to notify other objects without making assumptions about who these objects are.
  - In other words, you don't want these objects tightly coupled.

Source: https://www.dofactory.com/net/observer-design-pattern

Discussion

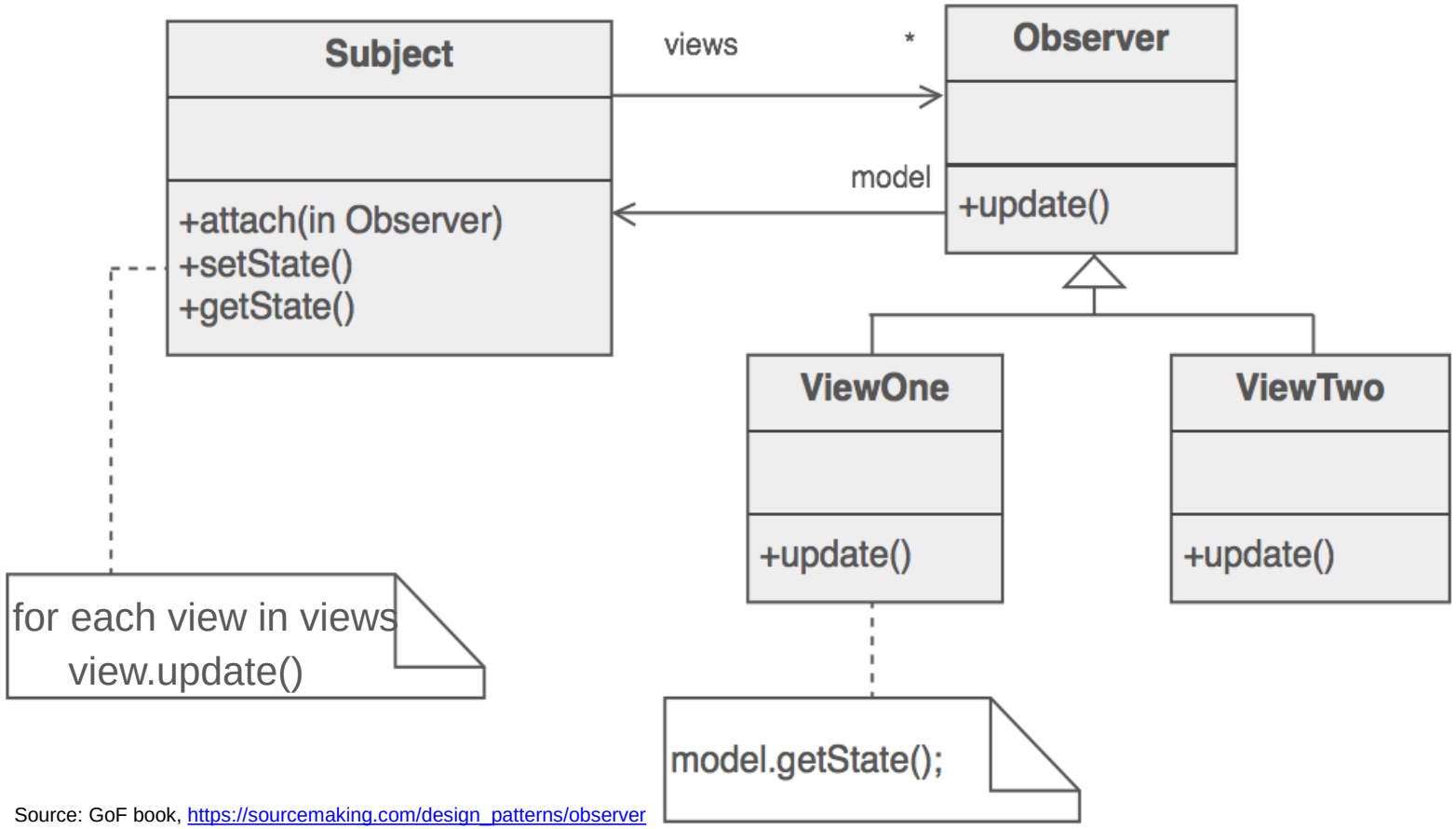
- Define an object that is the "keeper" of the data model and/or business logic (the Subject).
- Delegate all "view" functionality to decoupled and distinct Observer objects.
- Observers register themselves with the Subject as they are created.
- Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.
- This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.

The protocol described above specifies a "pull" interaction model. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.

Source: GoF book, https://sourcemaking.com/design_patterns/observer

Observer Pattern Structure

- Subject represents the core (or independent or common or engine) abstraction.
- Observer represents the variable (or dependent or optional or user interface) abstraction.
- The Subject prompts the Observer objects to do their thing.
- Each Observer can call back to the Subject as needed.



Source: GoF book, https://sourcemaking.com/design_patterns/observer

Participants

Subject

- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

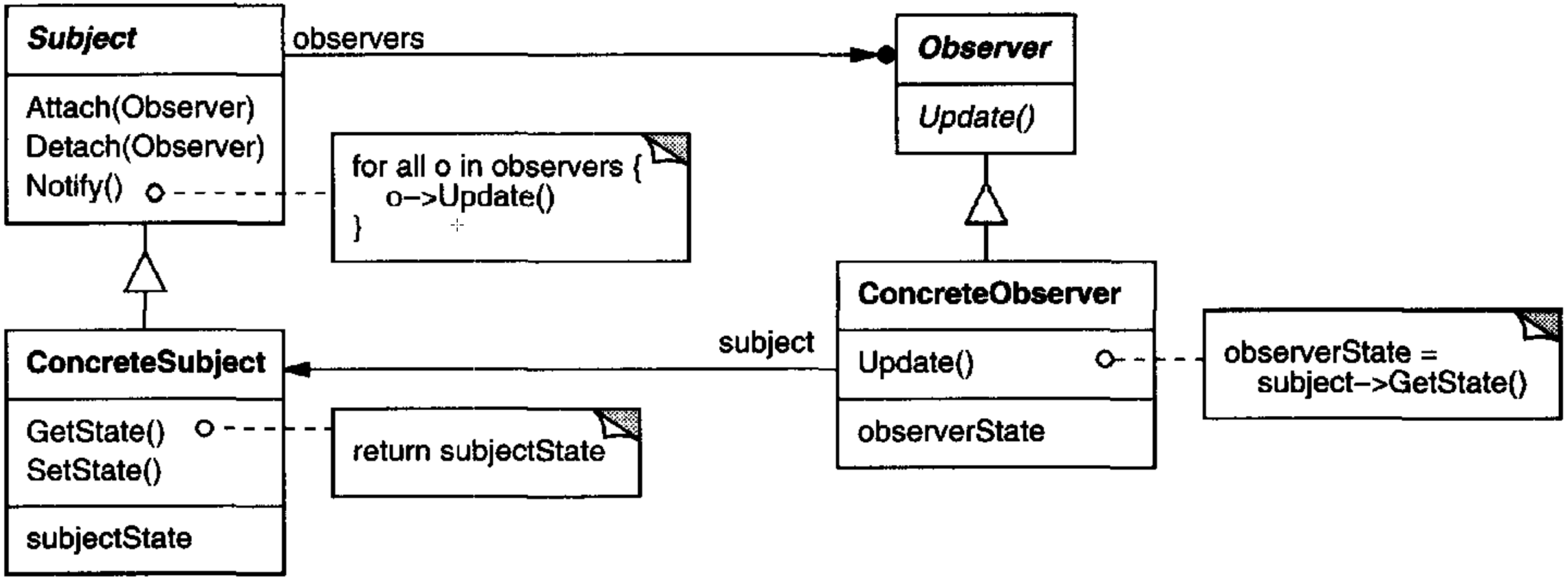
Observer

- defines an updating interface for objects that should be notified of changes in a subject.

ConcreteObserver (ViewOne, ViewTwo)

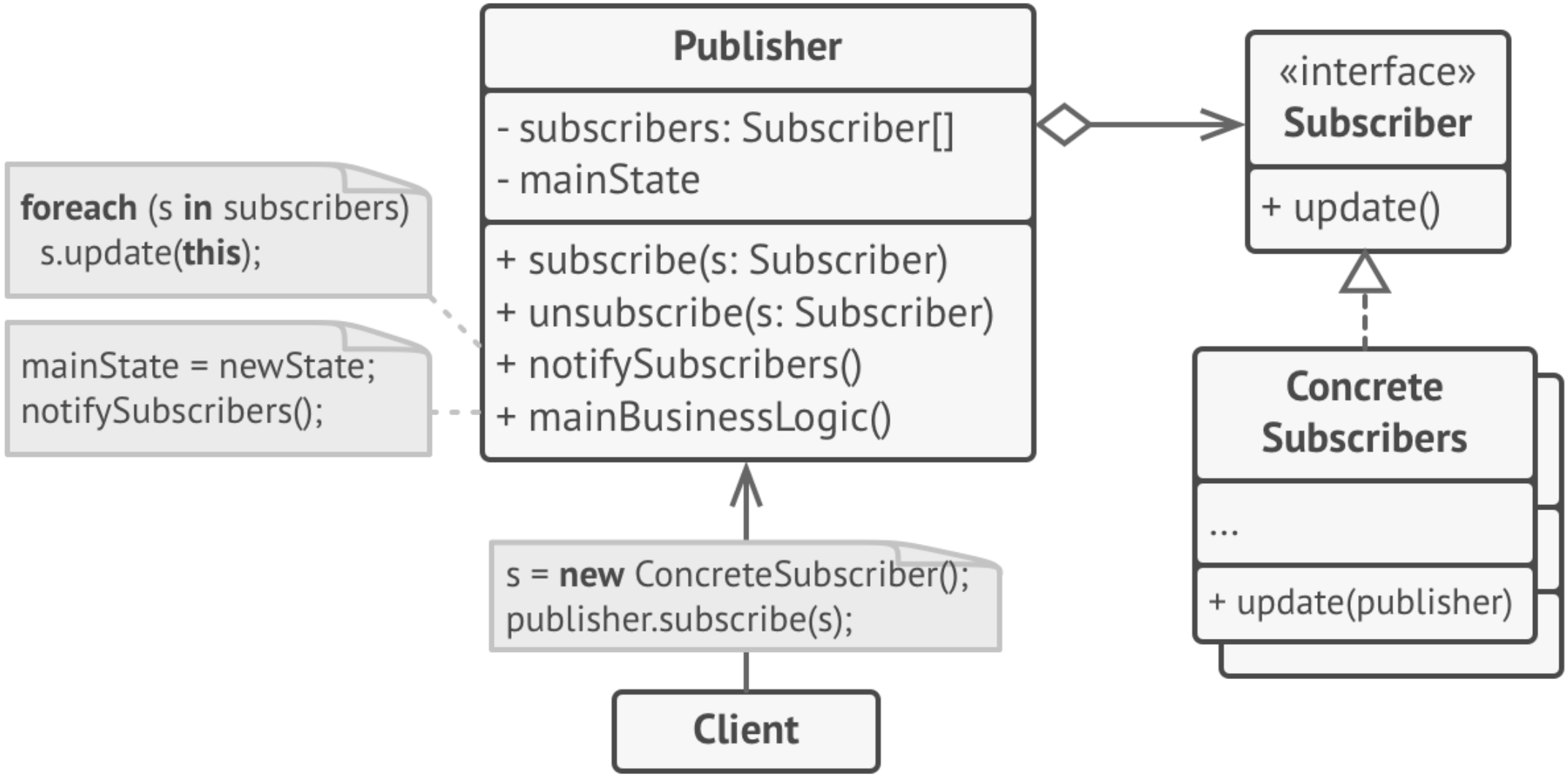
- maintains a reference to a Subject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's

Original GoF Structure



Source: GoF book

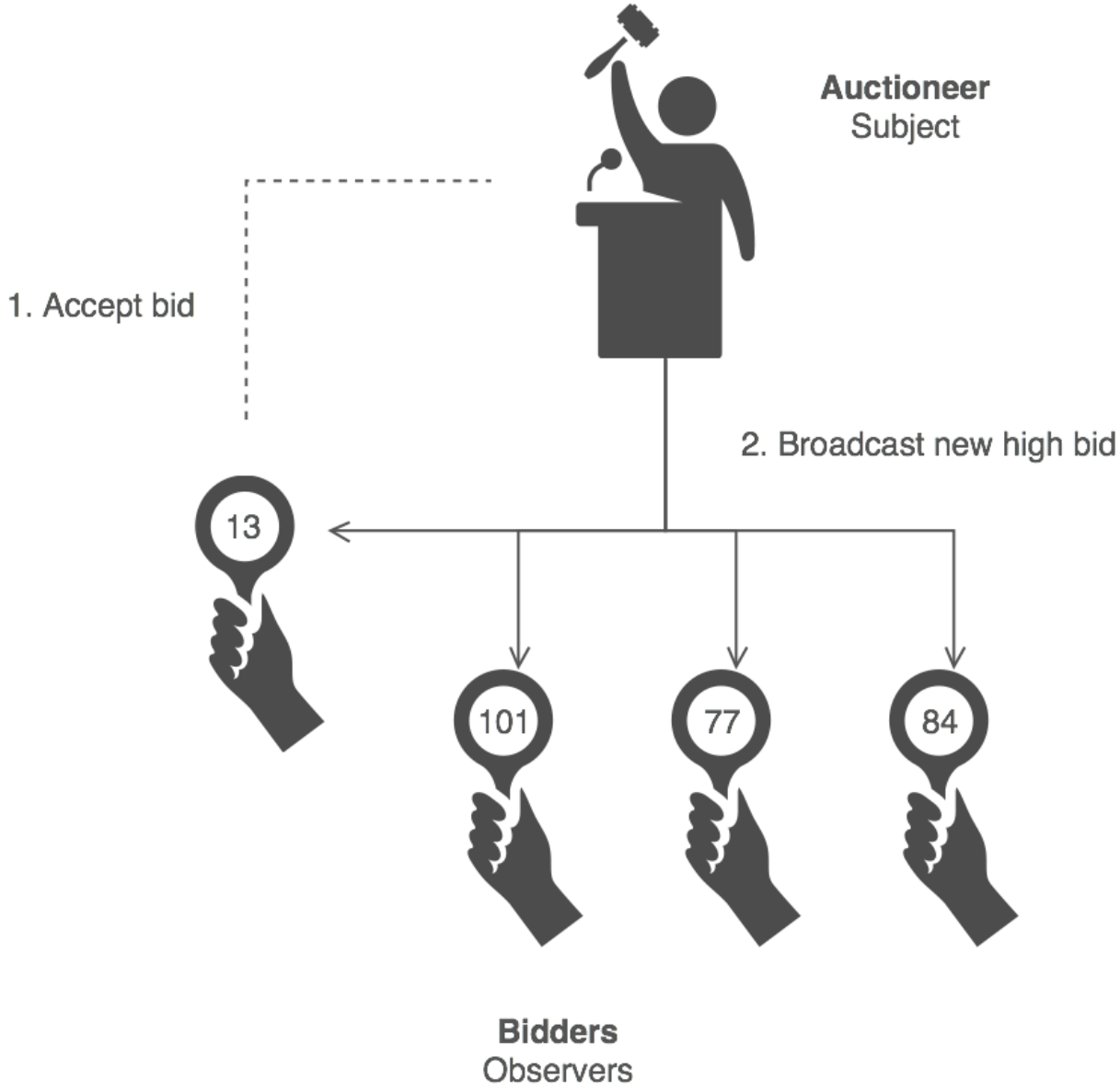
Observer Structure with Publisher and Subscriber Example



Source: https://refactoring.guru/design-patterns/observer

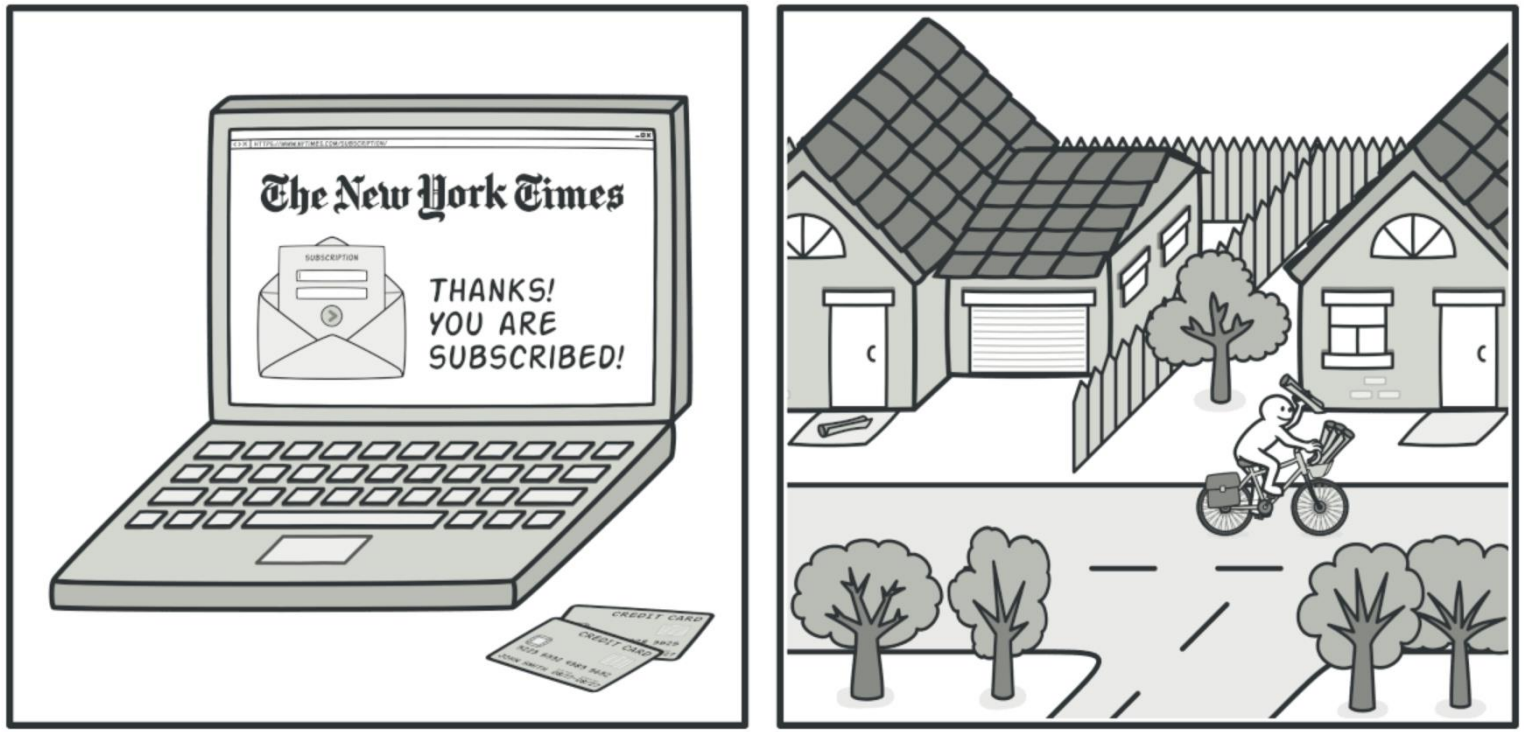
Observer Pattern Auction Intuition Example

- The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically.
- Some auctions demonstrate this pattern.
- Each bidder possesses a numbered paddle that is used to indicate a bid.
- The auctioneer (subject) starts the bidding, and reacts when a paddle is raised to accept the bid.
- The acceptance of the bid changes the bid price which is broadcast to all of the bidders (observers) in the form of a new bid.



Source: GoF book, https://sourcemaking.com/design_patterns/observer

Observer Pattern Subscription Analogy

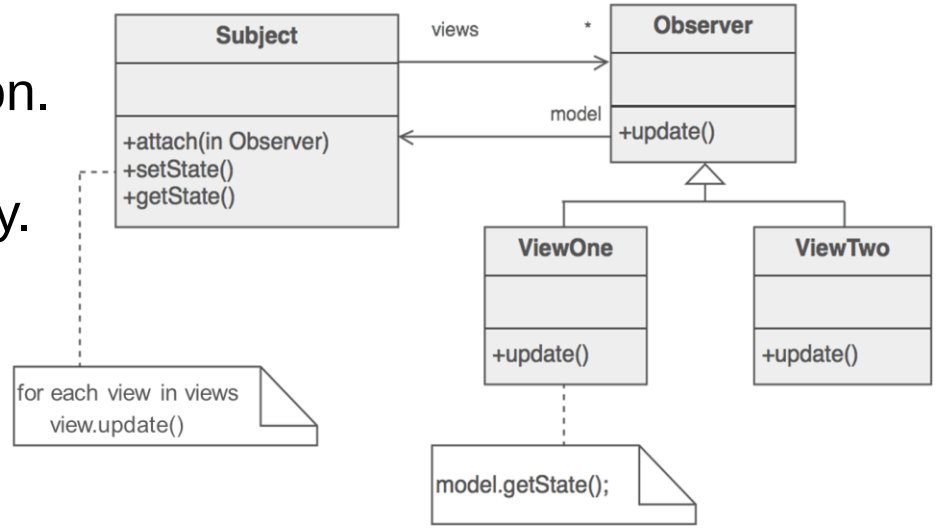


- If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance.
- The publisher maintains a list of subscribers and knows which magazines they're interested in. Subscribers can leave the list at any time when they wish to stop the publisher sending new magazine issues to them.

Source: https://refactoring.guru/design-patterns/observer

Observer Pattern: Check List

1. Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
2. Model the independent functionality with a "subject" abstraction.
3. Model the dependent functionality with an "observer" hierarchy.
4. The Subject is coupled only to the Observer base class.
5. The client configures the number and type of Observers.
6. Observers register themselves with the Subject OR they are registered by the client.
7. The Subject broadcasts events to all registered Observers.
8. The Subject may "push" information at the Observers, or, the Observers may "pull" the information they need from the Subject.



Source: GoF book, https://sourcemaking.com/design_patterns/observer

Relation to other Patterns

- Chain of Responsibility, Command, Mediator, and Observer address how you can decouple senders and receivers, but with different trade-offs.
- Chain of Responsibility passes a sender request along a chain of potential receivers.
- Command normally specifies a sender-receiver connection with a subclass.
- Mediator has senders and receivers reference each other indirectly.
- Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time.
- Mediator and Observer are competing patterns.
- The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects.
- It might be easier to make reusable Observers and Subjects than to make reusable Mediators.
- On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them.

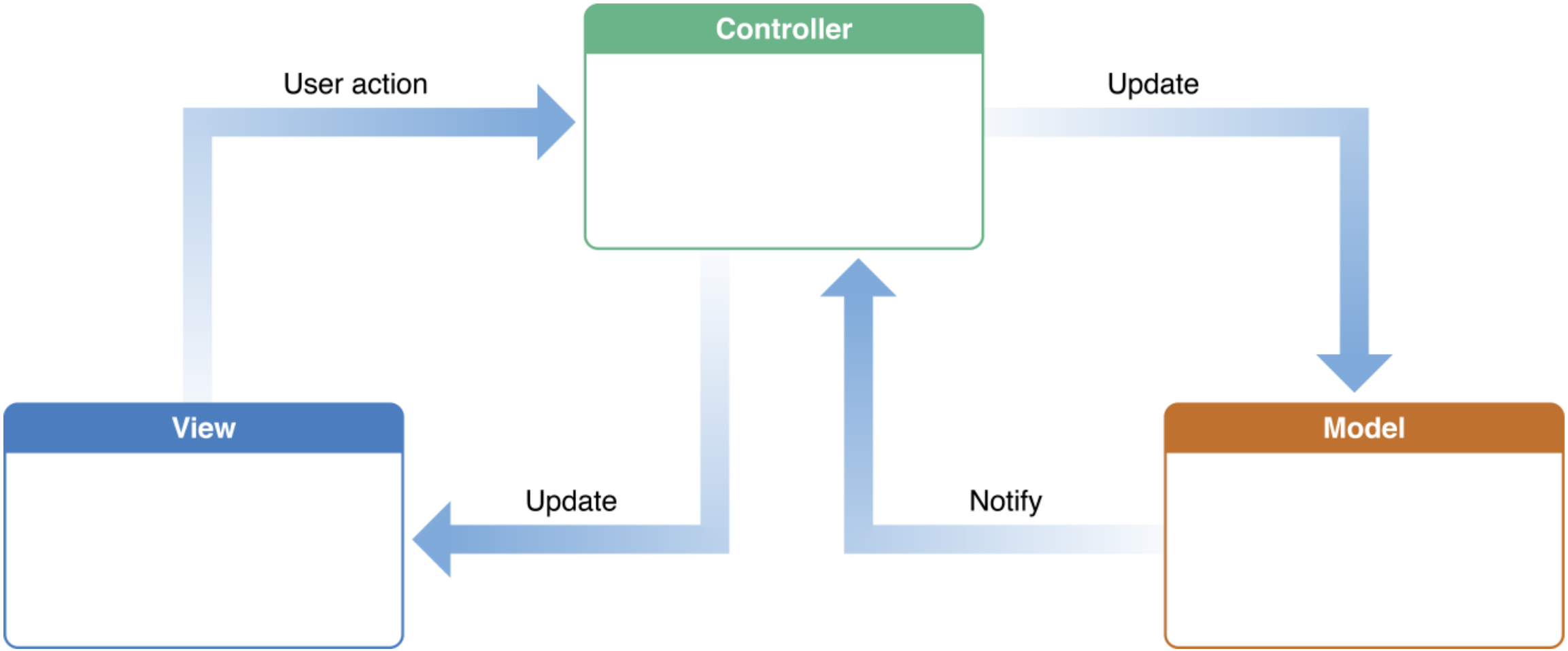
Source: GoF book, https://sourcemaking.com/design_patterns/observer

Consequences and Advanced Properties

- Issues that need to be considered:
  - implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred),
  - having a single Observer monitoring multiple Subjects,
  - and ensuring that a Subject notifies its Observers when it is about to go away.
- Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects.
- The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.

Source: GoF book, https://sourcemaking.com/design_patterns/observer

Plugin-Slide #1: Model-View-Controller



Source: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

Plugin-Slide #2: Model-View-Controller

- Observer can be related to the Model-View-Controller Pattern (from UI design).
- The Model-View-Controller (MVC) design pattern assigns objects in an application one of three roles: model, view, or controller.

Model

- Model objects encapsulate the data specific to an application and define the logic and computation that manipulate and process that data. For example, a model object might represent a character in a game or a contact in an address book.

View

- A view object is an object in an application that users can see. A view object knows how to draw itself and can respond to user actions. A major purpose of view objects is to display data from the application's model objects and to enable the editing of that data. Despite this, view objects are typically decoupled from model objects in an MVC application.

Controller

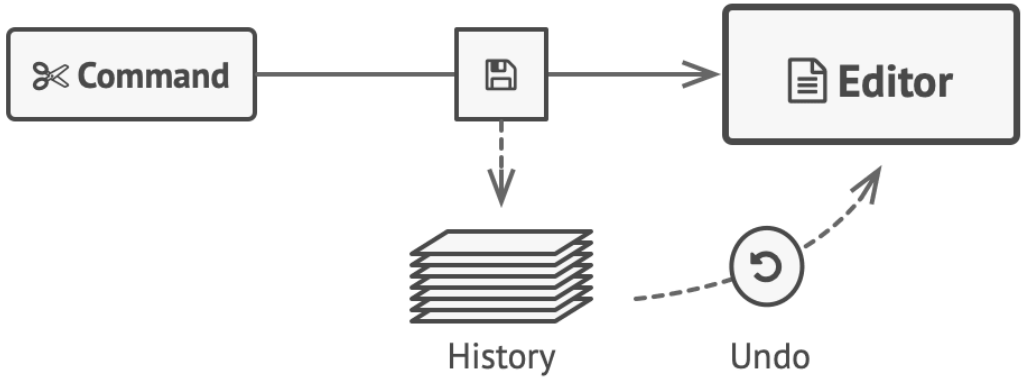
- A controller object acts as an intermediary between one or more of an application's view objects and one or more of its model objects. Controller objects are thus a conduit through which view objects learn about changes in model objects and vice versa.

Source: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

Behavioral Pattern: Memento (a.k.a. Token)

Problem

- The need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).



Intent

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
- A magic cookie that encapsulates a "check point" capability.
- Promote undo or rollback to full object status.

Applicability

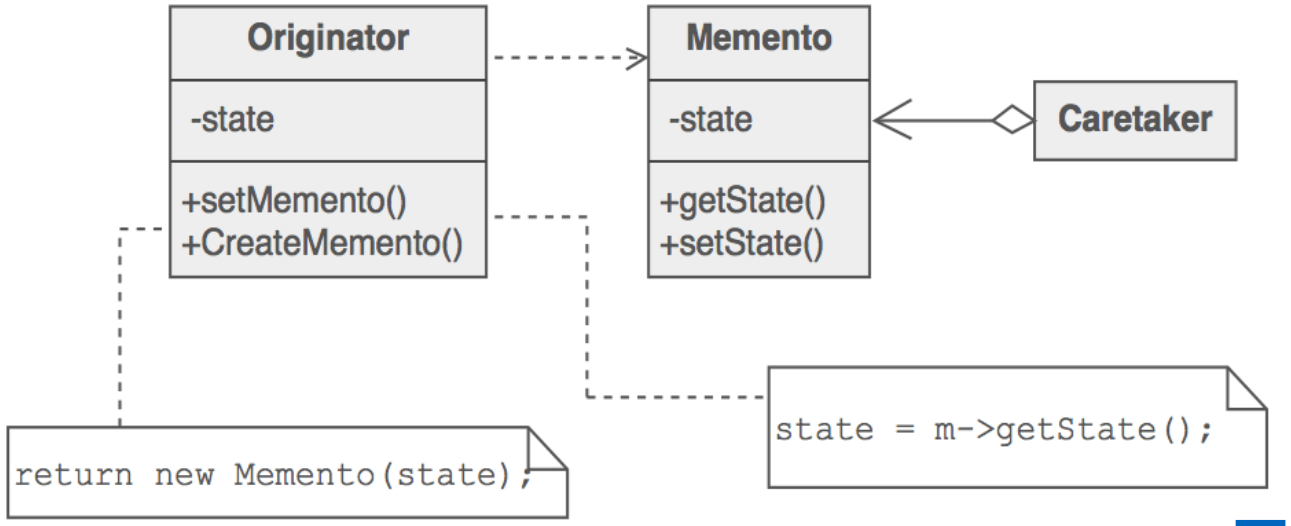
- Use the Memento pattern when
  - a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
  - a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Source: Gof Book, https://sourcemaking.com/design_patterns/memento, https://refactoring.guru/design-patterns/memento

Memento Discussion

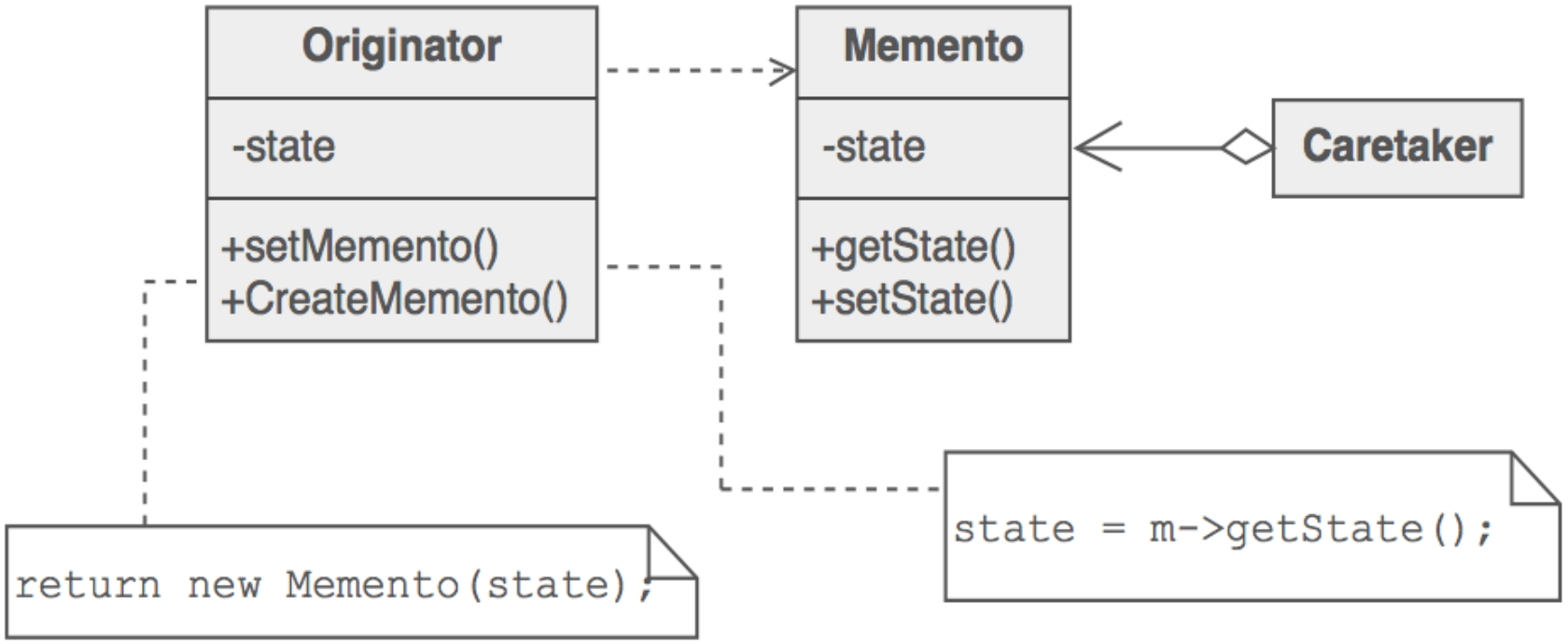
- The client requests a Memento from the source object when it needs to checkpoint the source object's state.
- The source object initializes the Memento with a characterization of its state.
- The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects).
- If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.

- An unlimited "undo" and "redo" capability can be readily implemented with a stack of Command objects and a stack of Memento objects.



Source: Gof Book, https://sourcemaking.com/design_patterns/memento

Memento Structure



Participants

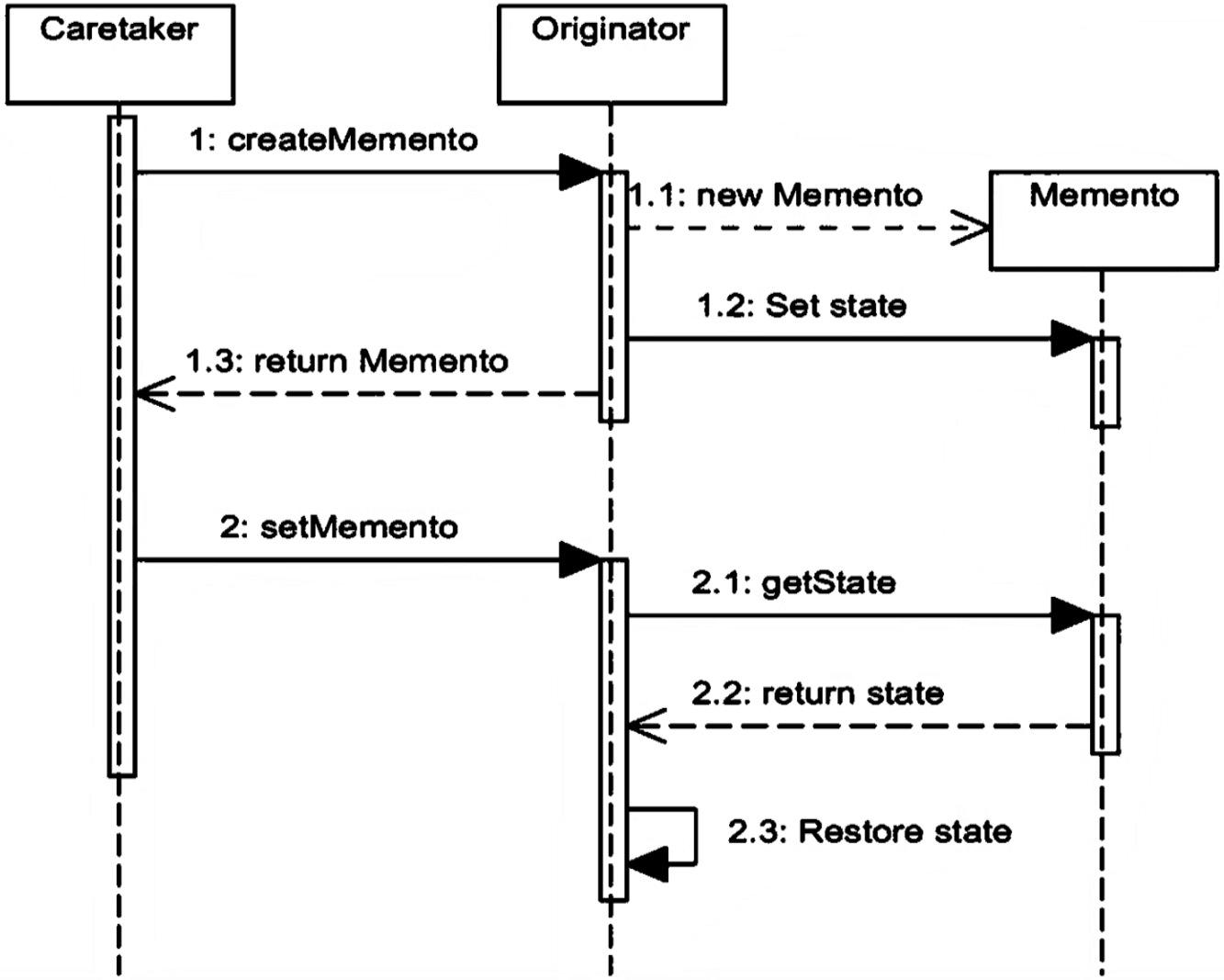
Originator
- the object that knows how to save itself.

Caretaker
- the object that knows why and when the Originator needs to save and restore itself.

Memento
- the lock box that is written and read by the Originator, and shepherded by the Caretaker.

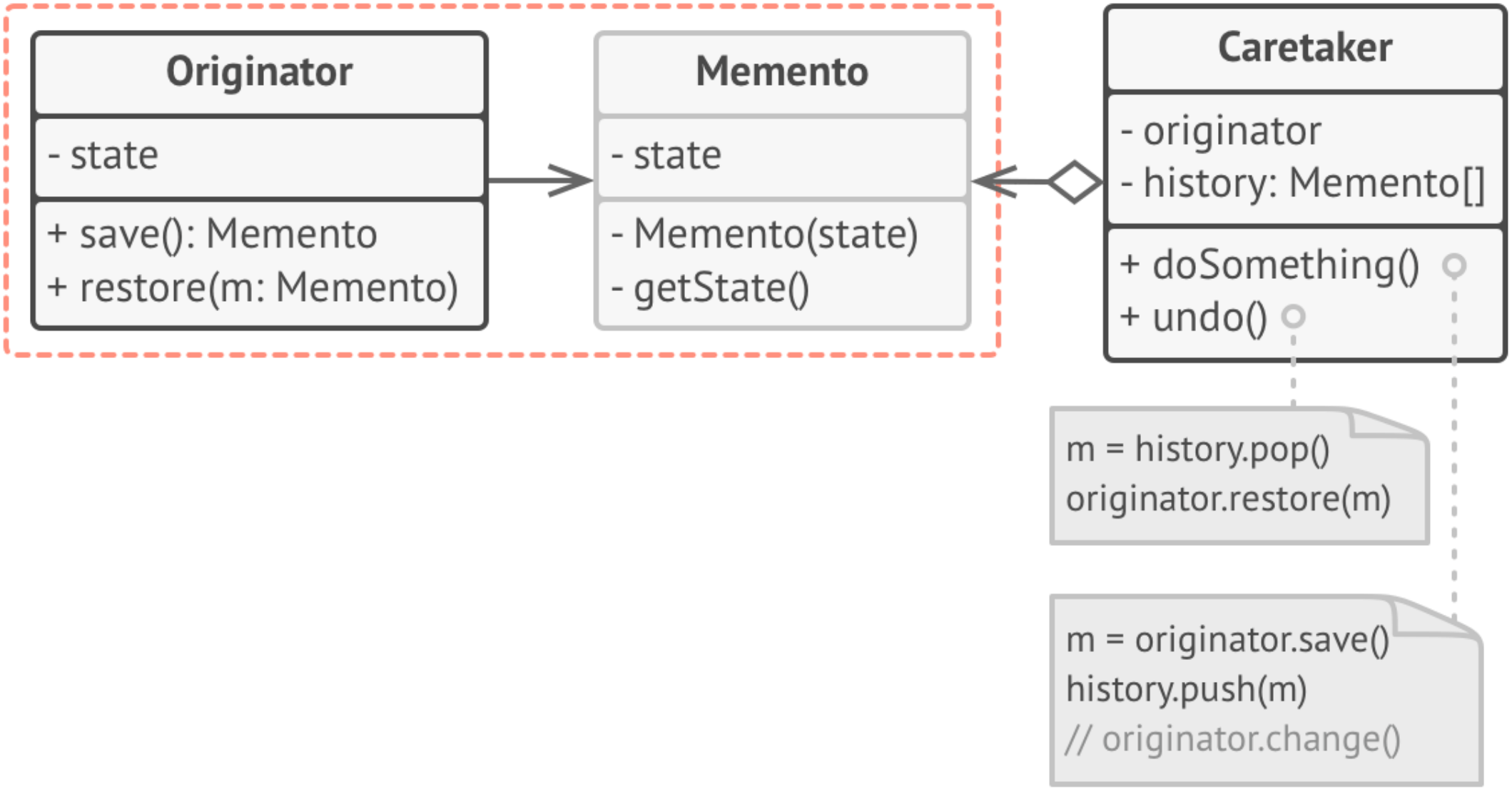
Source: Gof Book, https://sourcemaking.com/design_patterns/memento

Memento Pattern UML Sequence Diagram



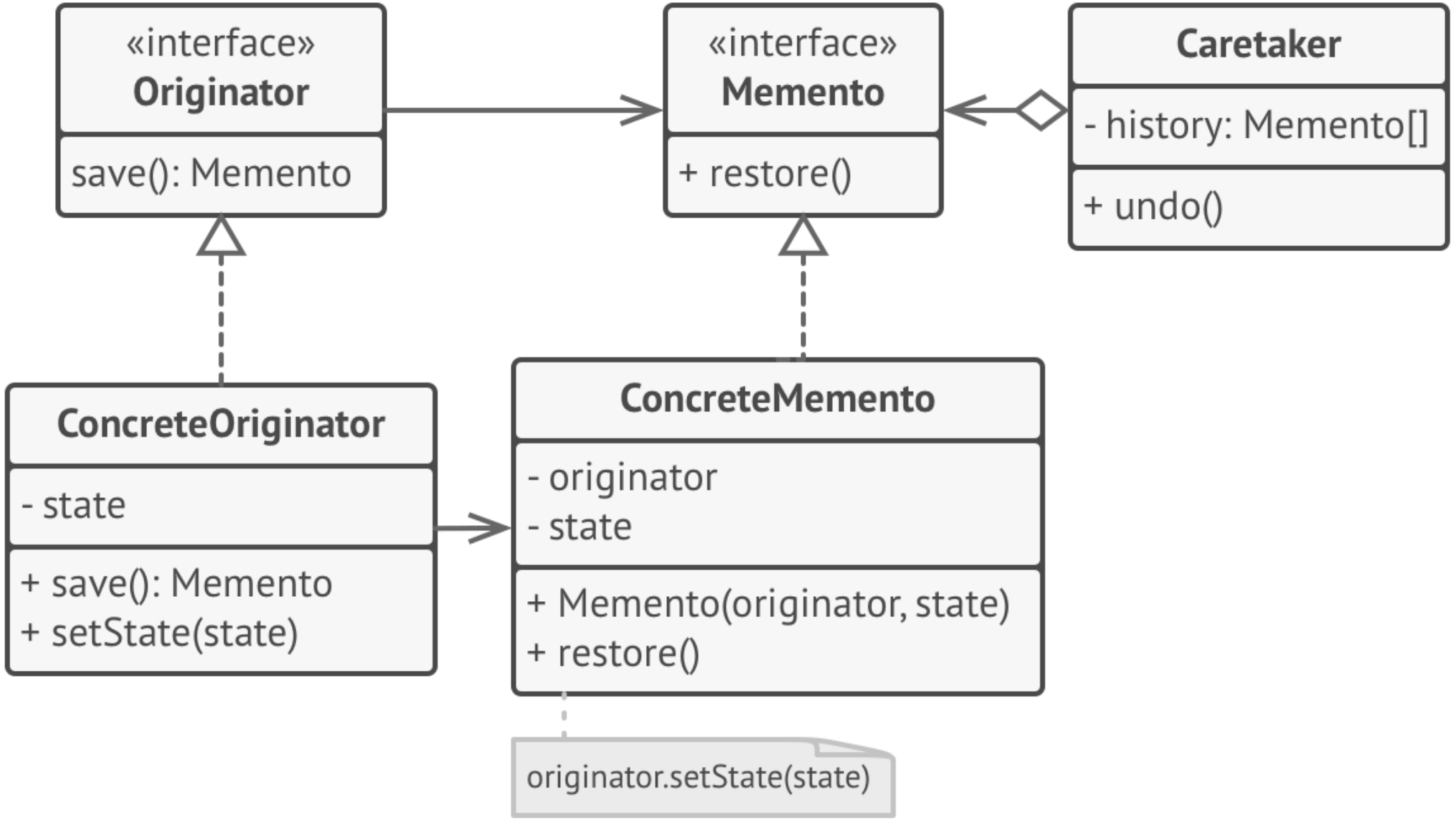
Source: https://ramj2ee.blogspot.com/2013/11/memento-design-pattern-sequence-diagram.html

Memento Undo Example



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

Memento with strict Encapsulation



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

Check List

1. Identify the roles of “caretaker” and “originator”.
2. Create a Memento class and declare the originator a friend.
3. Caretaker knows when to "check point" the originator.
4. Originator creates a Memento and copies its state to that Memento.
5. Caretaker holds on to (but cannot peek into) the Memento.
6. Caretaker knows when to "roll back" the originator.
7. Originator reinstates itself using the saved state in the Memento.

Source: GoF Book, https://sourcemaking.com/design_patterns/memento

Memento Consequences

- **Preserving encapsulation boundaries**

Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator. The pattern shields other objects from potentially complex Originator internals, thereby preserving encapsulation boundaries.

- **Using mementos might be expensive**

Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough. Unless encapsulating and restoring Originator state is cheap, the pattern might not be appropriate. Incremental Mementos might be a solution.

Source: GoF Book, https://sourcemaking.com/design_patterns/memento

Memento Consequences

- **Defining narrow and wide interfaces**

It may be difficult in some languages to ensure that only the originator can access the memento's state.

- **Hidden costs in caring for mementos**

A caretaker is responsible for deleting the mementos it cares for. However, the caretaker has no idea how much state is in the memento. Hence an otherwise lightweight caretaker might incur large storage costs when it stores mementos.

Source: GoF Book, https://sourcemaking.com/design_patterns/memento

Memento: Relation to other Patterns

- Command and Memento act as magic tokens to be passed around and invoked at a later time.
- In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time.
- Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value.
- Command can use Memento to maintain the state required for an undo operation.
- Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

Source: GoF Book, https://sourcemaking.com/design_patterns/memento

Behavioral Pattern: Strategy #1

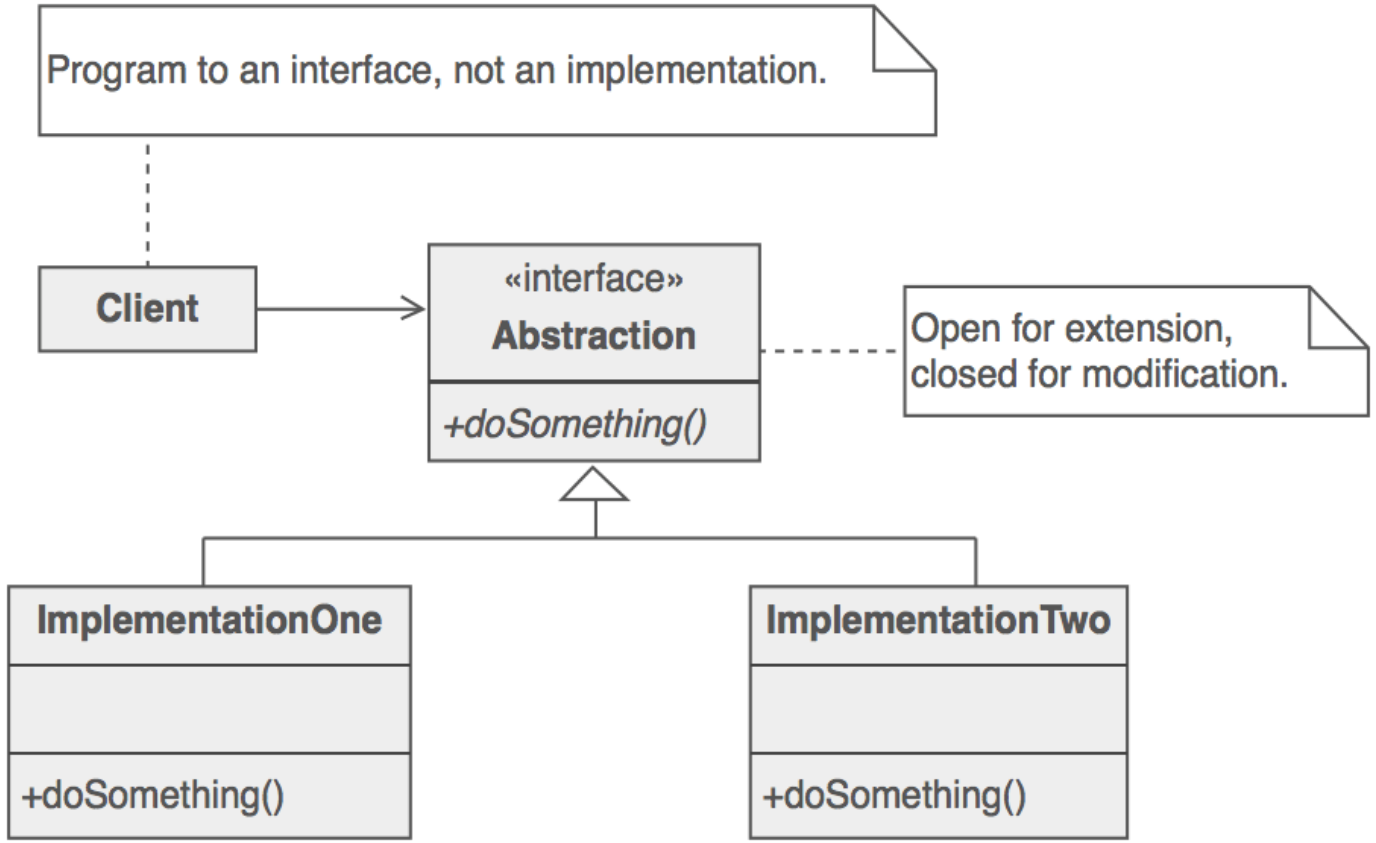
Problem

- One of the dominant strategies of object-oriented design is the "open-closed principle".

Common approach:

- Encapsulate interface details in a base class, and bury implementation details in derived classes.

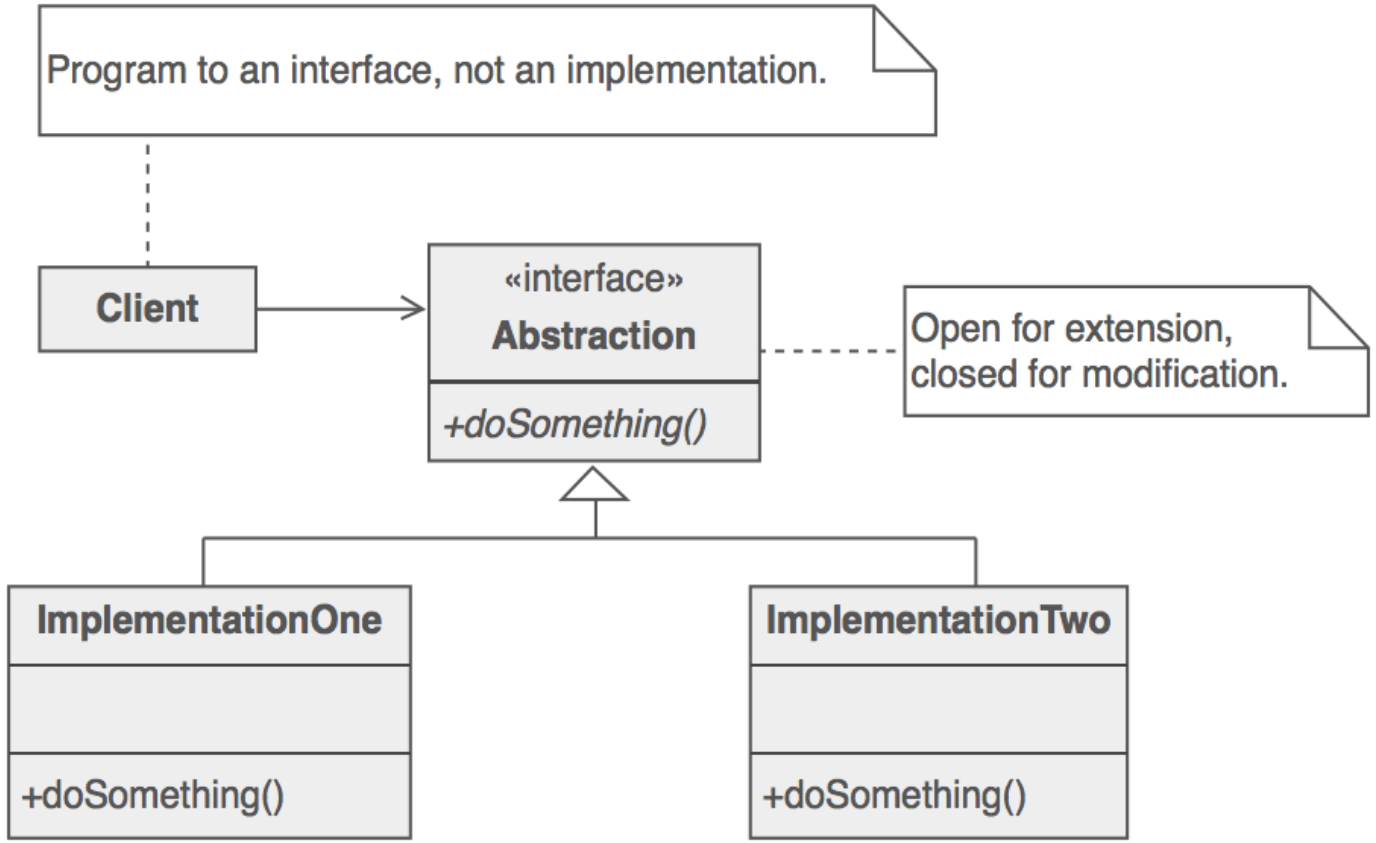
- Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.



Source: GoF Book, https://sourcemaking.com/design_patterns/strategy

Behavioral Pattern: Strategy #2

- A generic value of the software community for years has been, "maximize cohesion and minimize coupling".
- The object-oriented design approach shown here is all about minimizing coupling.
- Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing "abstract coupling", an object-oriented variant of the more generic exhortation "minimize coupling".
- **Program to an interface, not an implementation!**

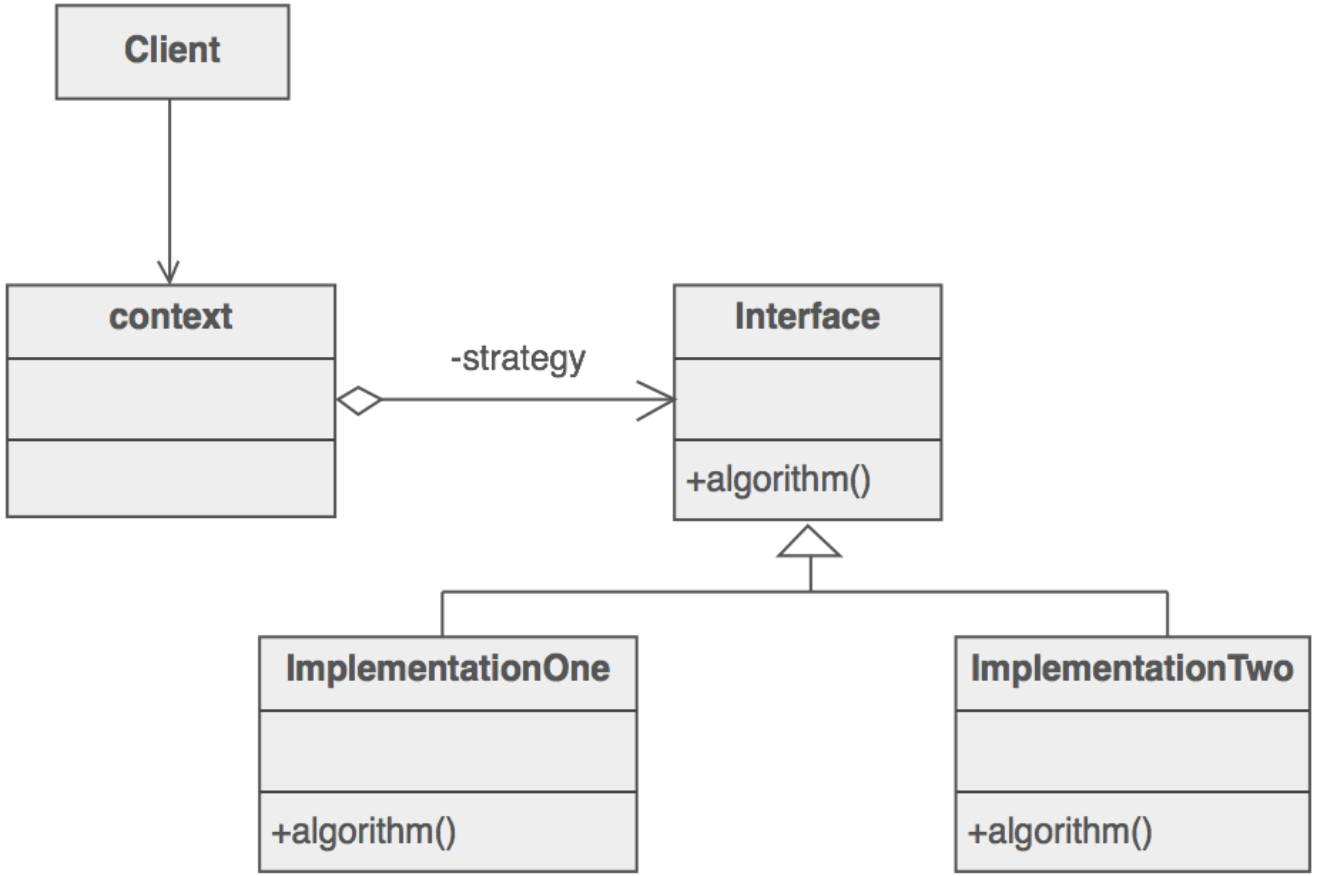


Source: GoF Book, https://sourcemaking.com/design_patterns/strategy

Strategy Pattern: Intent, Structure and Participants

Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.



Source: GoF Book, https://sourcemaking.com/design_patterns/strategy

Participants

Interface (ABC Strategy)

- declares an interface common to all supported algorithms.
- Context uses this interface to call the algorithm defined by an Implementation (ConcreteStrategy).

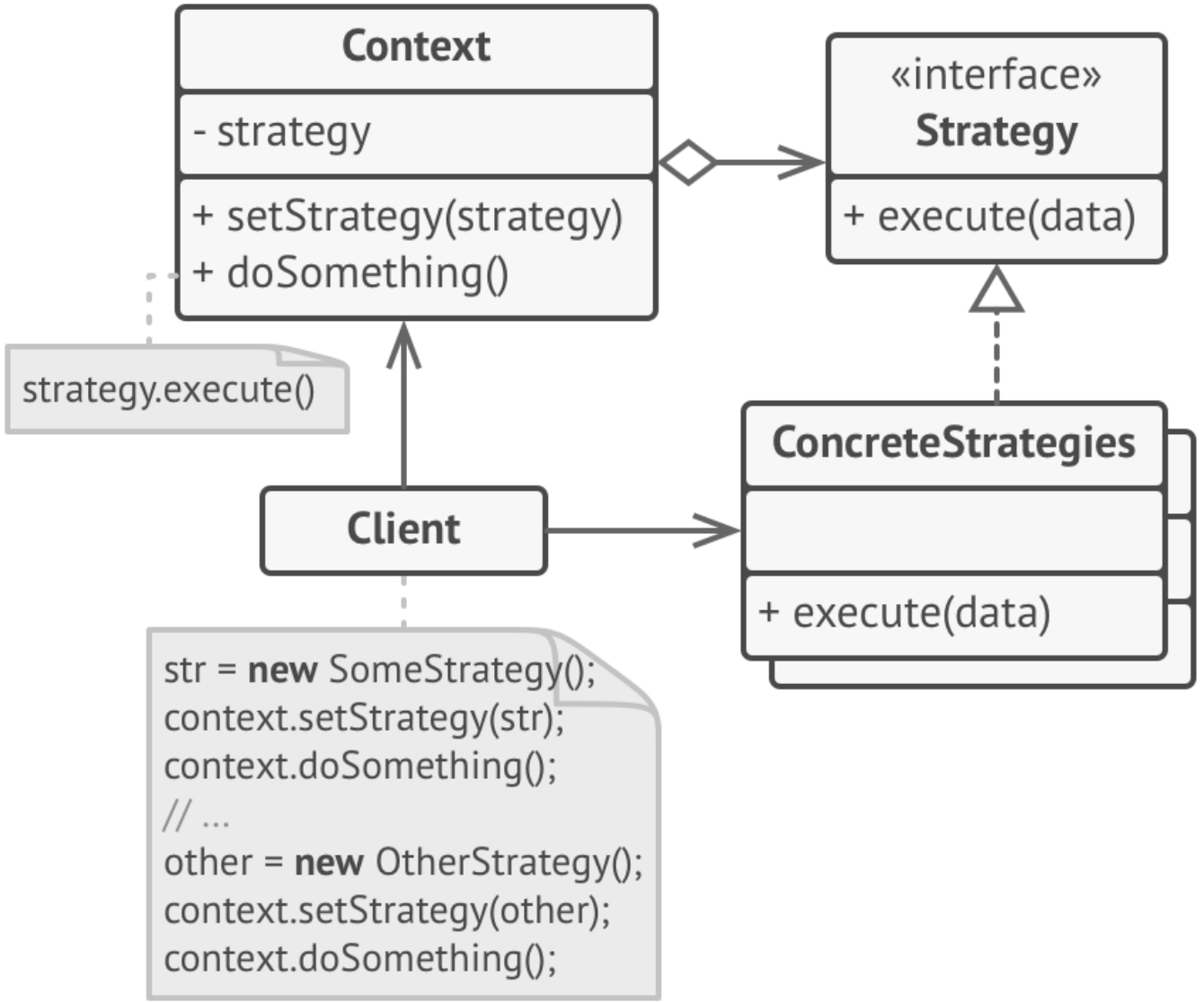
Implementation (ConcreteStrategy)

- implements the algorithm using the Strategy interface.

Context

- is configured with an Implementation (ConcreteStrategy) object.
- maintains a reference to an Interface object.

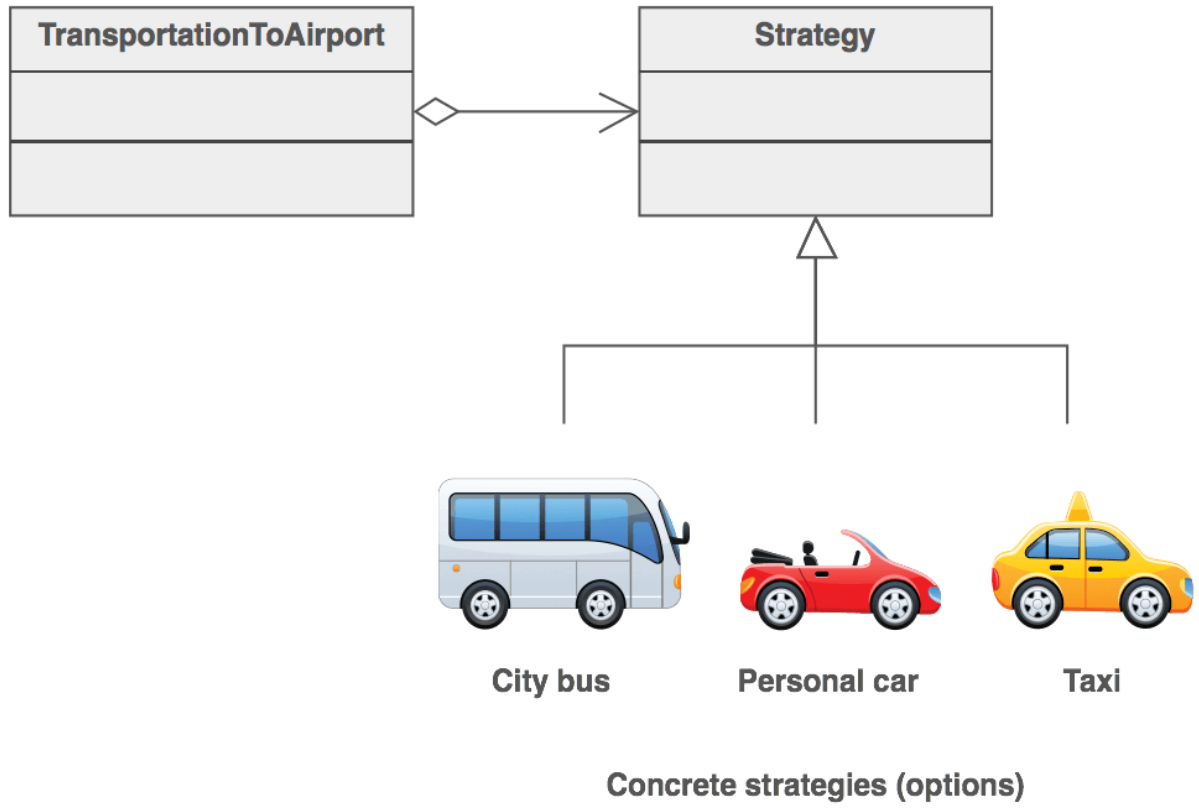
Different Perspective on same Structure



Source: https://refactoring.guru/design-patterns/strategy

Strategy Pattern Intuition Example

- A Strategy defines a set of algorithms that can be used interchangeably.
- Modes of transportation to an airport is an example of a Strategy.
- Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service.
- For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably.
- The traveler must choose the Strategy based on trade-offs between cost, convenience, and time.



Source: GoF Book, https://sourcemaking.com/design_patterns/strategy

Strategy: Check list

- Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
- Specify the signature for that algorithm in an interface.
- Bury the alternative implementation details in derived classes.
- Clients of the algorithm couple themselves to the interface.

Source: GoF Book, https://sourcemaking.com/design_patterns/strategy

Strategy: Relation to other Patterns

- Strategy is like Template Method except in its granularity.
- State is like Strategy except in its intent.
- Strategy lets you change the guts of an object. Decorator lets you change the skin.
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the 'handle/body' idiom. They differ in intent - that is, they solve different problems.
- Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).
- Strategy objects often make good Flyweights.

Source: GoF Book, https://sourcemaking.com/design_patterns/strategy

Wrap-Up Behavioral Patterns

- Behavioral design patterns are design patterns that **identify common communication patterns between objects and realize these patterns**. By doing so, these patterns increase flexibility in carrying out this communication.

- We looked at 3 behavioral design patterns: Observer, Memento and Strategy
  - **Observer** lets you define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

  - **Memento** lets you capture the object's internal state without exposing its internal structure, so that the object can be returned to this state later.

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

- Good resources: https://sourcemaking.com/design_patterns, https://refactoring.guru/design-patterns, GoF Book