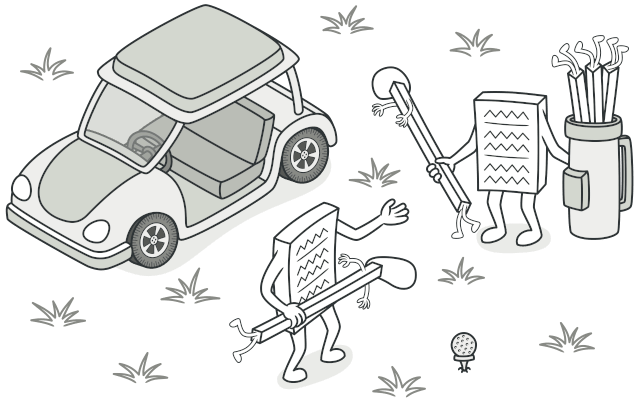
**Strategy**

**Intent**

**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.



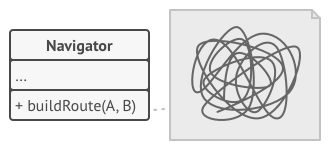
**Problem**

One day you decided to create a navigation app for casual travelers. The app was centered around a beautiful map which helped users quickly orient themselves in any city.

One of the most requested features for the app was automatic route planning. A user should be able to enter an address and see the fastest route to that destination displayed on the map.

The first version of the app could only build the routes over roads. People who traveled by car were bursting with joy. But apparently, not everybody likes to drive on their vacation. So with the next update, you added an option to build walking routes. Right after that, you added another option to let people use public transport in their routes.

However, that was only the beginning. Later you planned to add route building for cyclists. And even later, another option for building routes through all of a city’s tourist attractions.



The code of the navigator became bloated.

While from a business perspective the app was a success, the technical part caused you many headaches. Each time you added a new routing algorithm, the main class of the navigator doubled in size. At some point, the beast became too hard to maintain.

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.

In addition, teamwork became inefficient. Your teammates, who had been hired right after the successful release, complain that they spend too much time resolving merge conflicts. Implementing a new feature requires you to change the same huge class, conflicting with the code produced by other people.

**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.

The context isn’t responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context doesn’t know much about strategies. It works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.

This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones without changing the code of the context or other strategies.

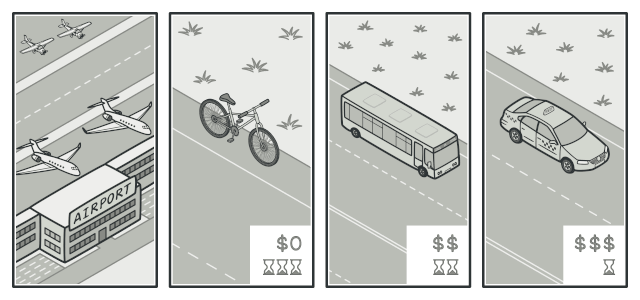


Route planning strategies.

In our navigation app, each routing algorithm can be extracted to its own class with a single buildRoute method. The method accepts an origin and destination and returns a collection of the route’s checkpoints.

Even though given the same arguments, each routing class might build a different route, the main navigator class doesn’t really care which algorithm is selected since its primary job is to render a set of checkpoints on the map. The class has a method for switching the active routing strategy, so its clients, such as the buttons in the user interface, can replace the currently selected routing behavior with another one.

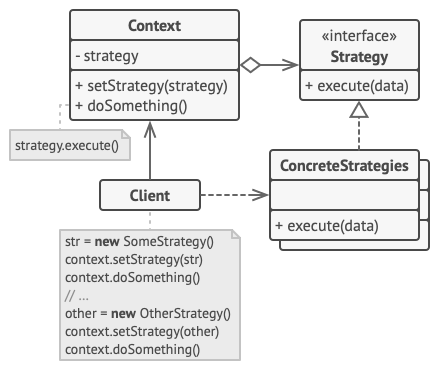
**Real-World Analogy**



Various strategies for getting to the airport.

Imagine that you have to get to the airport. You can catch a bus, order a cab, 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.

**Structure**



1. The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.
2. The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.
3. **Concrete Strategies** implement different variations of an algorithm the context uses.
4. The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn’t know what type of strategy it works with or how the algorithm is executed.
5. The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

**Pseudocode**

In this example, the context uses multiple **strategies** to execute various arithmetic operations.

// The strategy interface declares operations common to all

// supported versions of some algorithm. The context uses this

// interface to call the algorithm defined by the concrete

// strategies.

**interface** **Strategy** **is**

**method** execute(a, b)

// Concrete strategies implement the algorithm while following

// the base strategy interface. The interface makes them

// interchangeable in the context.

**class** **ConcreteStrategyAdd** **implements** Strategy **is**

**method** execute(a, b) **is**

**return** a + b

**class** **ConcreteStrategySubtract** **implements** Strategy **is**

**method** execute(a, b) **is**

**return** a - b

**class** **ConcreteStrategyMultiply** **implements** Strategy **is**

**method** execute(a, b) **is**

**return** a \* b

// The context defines the interface of interest to clients.

**class** **Context** **is**

// The context maintains a reference to one of the strategy

// objects. The context doesn't know the concrete class of a

// strategy. It should work with all strategies via the

// strategy interface.

**private** strategy: Strategy

// Usually the context accepts a strategy through the

// constructor, and also provides a setter so that the

// strategy can be switched at runtime.

**method** setStrategy(Strategy strategy) **is**

**this**.strategy = strategy

// The context delegates some work to the strategy object

// instead of implementing multiple versions of the

// algorithm on its own.

**method** executeStrategy(int a, int b) **is**

**return** strategy.execute(a, b)

// The client code picks a concrete strategy and passes it to

// the context. The client should be aware of the differences

// between strategies in order to make the right choice.

**class** **ExampleApplication** **is**

**method** main() **is**

Create context object.

Read first number.

Read last number.

Read the desired action from user input.

**if** (action == addition) **then**

context.setStrategy(**new** ConcreteStrategyAdd())

**if** (action == subtraction) **then**

context.setStrategy(**new** ConcreteStrategySubtract())

**if** (action == multiplication) **then**

context.setStrategy(**new** ConcreteStrategyMultiply())

result = context.executeStrategy(First number, Second number)

Print result.

**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.**

 The Strategy pattern lets you indirectly alter the object’s behavior at runtime by associating it with different sub-objects which can perform specific sub-tasks in different ways.

**Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.**

 The Strategy pattern lets you extract the varying behavior into a separate class hierarchy and combine the original classes into one, thereby reducing duplicate code.

**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.**

 The Strategy pattern lets you isolate the code, internal data, and dependencies of various algorithms from the rest of the code. Various clients get a simple interface to execute the algorithms and switch them at runtime.

**Use the pattern when your class has a massive conditional statement that switches between different variants of the same algorithm.**

 The Strategy pattern lets you do away with such a conditional by extracting all algorithms into separate classes, all of which implement the same interface. The original object delegates execution to one of these objects, instead of implementing all variants of the algorithm.

**How to Implement**

1. In the context class, identify an algorithm that’s prone to frequent changes. It may also be a massive conditional that selects and executes a variant of the same algorithm at runtime.
2. Declare the strategy interface common to all variants of the algorithm.
3. One by one, extract all algorithms into their own classes. They should all implement the strategy interface.
4. In the context class, add a field for storing a reference to a strategy object. Provide a setter for replacing values of that field. The context should work with the strategy object only via the strategy interface. The context may define an interface which lets the strategy access its data.
5. Clients of the context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job.

**Pros and Cons**

* 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.
* 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.

**Relations with Other Patterns**

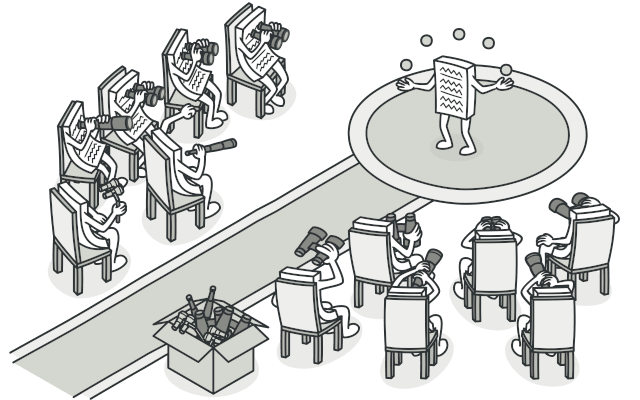
* [**Bridge**](https://refactoring.guru/design-patterns/bridge), **[State](https://refactoring.guru/design-patterns/state)**, **[Strategy](https://refactoring.guru/design-patterns/strategy)** (and to some degree **[Adapter](https://refactoring.guru/design-patterns/adapter)**) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn’t just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.
* [**Command**](https://refactoring.guru/design-patterns/command) and **[Strategy](https://refactoring.guru/design-patterns/strategy)** may look similar because you can use both to parameterize an object with some action. However, they have very different intents.
  + You can use *Command* to convert any operation into an object. The operation’s parameters become fields of that object. The conversion lets you defer execution of the operation, queue it, store the history of commands, send commands to remote services, etc.
  + On the other hand, *Strategy* usually describes different ways of doing the same thing, letting you swap these algorithms within a single context class.
* [**Decorator**](https://refactoring.guru/design-patterns/decorator) lets you change the skin of an object, while **[Strategy](https://refactoring.guru/design-patterns/strategy)** lets you change the guts.
* [**Template Method**](https://refactoring.guru/design-patterns/template-method) is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses. **[Strategy](https://refactoring.guru/design-patterns/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**](https://refactoring.guru/design-patterns/state) can be considered as an extension of **[Strategy](https://refactoring.guru/design-patterns/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.

**Observer**

**Also known as:**Event-Subscriber, Listener

**Intent**

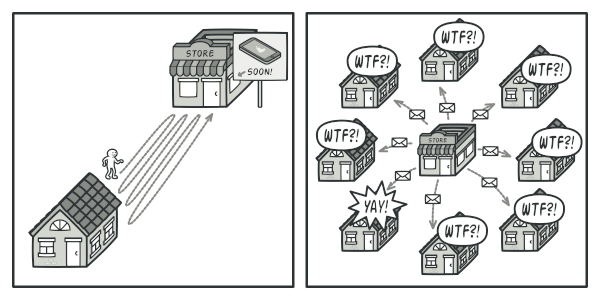
**Observer** is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.



**Problem**

Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it’s a new model of the iPhone) which should become available in the store very soon.

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.



Visiting the store vs. sending spam

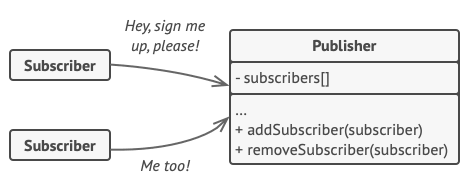
On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it’d upset other customers who aren’t interested in new products.

It looks like we’ve got a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

**Solution**

The object that has some interesting state is often called *subject*, but since it’s also going to notify other objects about the changes to its state, we’ll call it *publisher*. All other objects that want to track changes to the publisher’s state are called *subscribers*.

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. Fear not! Everything isn’t as complicated as it sounds. In reality, this mechanism consists of 1) an array field for storing a list of references to subscriber objects and 2) several public methods which allow adding subscribers to and removing them from that list.

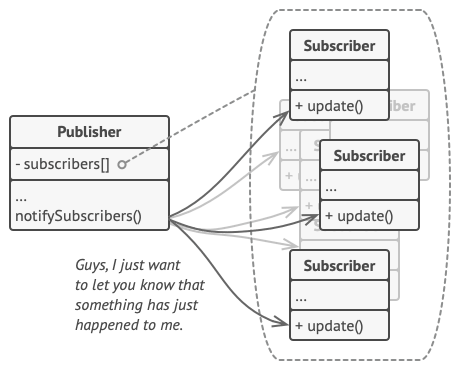


A subscription mechanism lets individual objects subscribe to event notifications.

Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.

Real apps might have dozens of different subscriber classes that are interested in tracking events of the same publisher class. You wouldn’t want to couple the publisher to all of those classes. Besides, you might not even know about some of them beforehand if your publisher class is supposed to be used by other people.

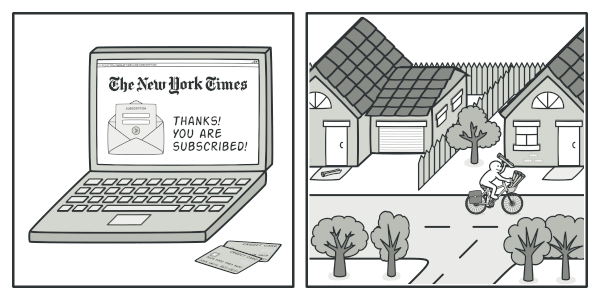
That’s why it’s crucial that all subscribers implement the same interface and that the publisher communicates with them only via that interface. This interface should declare the notification method along with a set of parameters that the publisher can use to pass some contextual data along with the notification.



Publisher notifies subscribers by calling the specific notification method on their objects.

If your app has several different types of publishers and you want to make your subscribers compatible with all of them, you can go even further and make all publishers follow the same interface. This interface would only need to describe a few subscription methods. The interface would allow subscribers to observe publishers’ states without coupling to their concrete classes.

**Real-World Analogy**

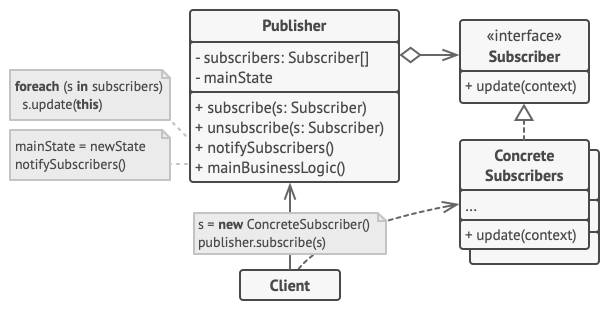


Magazine and newspaper subscriptions.

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.

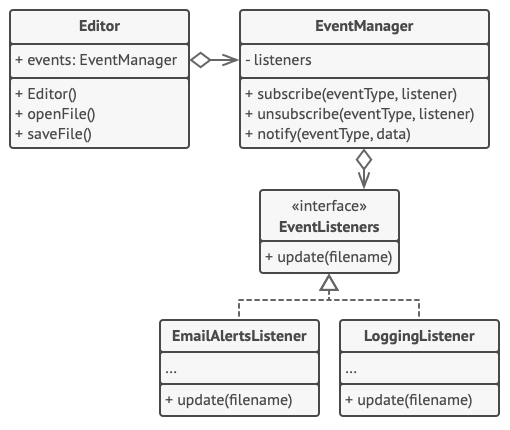
**Structure**



1. The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.
2. When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.
3. The **Subscriber** interface declares the notification interface. In most cases, it consists of a single update method. The method may have several parameters that let the publisher pass some event details along with the update.
4. **Concrete Subscribers** perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn’t coupled to concrete classes.
5. Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method. The publisher can pass itself as an argument, letting subscriber fetch any required data directly.
6. The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

**Pseudocode**

In this example, the **Observer** pattern lets the text editor object notify other service objects about changes in its state.



Notifying objects about events that happen to other objects.

The list of subscribers is compiled dynamically: objects can start or stop listening to notifications at runtime, depending on the desired behavior of your app.

In this implementation, the editor class doesn’t maintain the subscription list by itself. It delegates this job to the special helper object devoted to just that. You could upgrade that object to serve as a centralized event dispatcher, letting any object act as a publisher.

Adding new subscribers to the program doesn’t require changes to existing publisher classes, as long as they work with all subscribers through the same interface.

// The base publisher class includes subscription management

// code and notification methods.

**class** **EventManager** **is**

**private** **field** listeners: hash map of event types **and** listeners

**method** subscribe(eventType, listener) **is**

listeners.add(eventType, listener)

**method** unsubscribe(eventType, listener) **is**

listeners.remove(eventType, listener)

**method** notify(eventType, data) **is**

**foreach** (listener in listeners.of(eventType)) do

listener.update(data)

// The concrete publisher contains real business logic that's

// interesting for some subscribers. We could derive this class

// from the base publisher, but that isn't always possible in

// real life because the concrete publisher might already be a

// subclass. In this case, you can patch the subscription logic

// in with composition, as we did here.

**class** **Editor** **is**

**public** **field** events: EventManager

**private** **field** file: File

**constructor** Editor() **is**

events = **new** EventManager()

// Methods of business logic can notify subscribers about

// changes.

**method** openFile(path) **is**

**this**.file = **new** File(path)

events.notify("open", file.name)

**method** saveFile() **is**

file.write()

events.notify("save", file.name)

// ...

// Here's the subscriber interface. If your programming language

// supports functional types, you can replace the whole

// subscriber hierarchy with a set of functions.

**interface** **EventListener** **is**

**method** update(filename)

// Concrete subscribers react to updates issued by the publisher

// they are attached to.

**class** **LoggingListener** **implements** EventListener **is**

**private** **field** log: File

**private** **field** message: string

**constructor** LoggingListener(log\_filename, message) **is**

**this**.log = **new** File(log\_filename)

**this**.message = message

**method** update(filename) **is**

log.write(replace('%s',filename,message))

**class** **EmailAlertsListener** **implements** EventListener **is**

**private** **field** email: string

**private** **field** message: string

**constructor** EmailAlertsListener(email, message) **is**

**this**.email = email

**this**.message = message

**method** update(filename) **is**

system.email(email, replace('%s',filename,message))

// An application can configure publishers and subscribers at

// runtime.

**class** **Application** **is**

**method** config() **is**

editor = **new** Editor()

logger = **new** LoggingListener(

"/path/to/log.txt",

"Someone has opened the file: %s")

editor.events.subscribe("open", logger)

emailAlerts = **new** EmailAlertsListener(

"admin@example.com",

"Someone has changed the file: %s")

editor.events.subscribe("save", emailAlerts)

**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.**

 You can often experience this problem when working with classes of the graphical user interface. For example, you created custom button classes, and you want to let the clients hook some custom code to your buttons so that it fires whenever a user presses a button.

The Observer pattern lets any object that implements the subscriber interface subscribe for event notifications in publisher objects. You can add the subscription mechanism to your buttons, letting the clients hook up their custom code via custom subscriber classes.

**Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.**

 The subscription list is dynamic, so subscribers can join or leave the list whenever they need to.

**How to Implement**

1. Look over your business logic and try to break it down into two parts: the core functionality, independent from other code, will act as the publisher; the rest will turn into a set of subscriber classes.
2. Declare the subscriber interface. At a bare minimum, it should declare a single update method.
3. Declare the publisher interface and describe a pair of methods for adding a subscriber object to and removing it from the list. Remember that publishers must work with subscribers only via the subscriber interface.
4. Decide where to put the actual subscription list and the implementation of subscription methods. Usually, this code looks the same for all types of publishers, so the obvious place to put it is in an abstract class derived directly from the publisher interface. Concrete publishers extend that class, inheriting the subscription behavior.

However, if you’re applying the pattern to an existing class hierarchy, consider an approach based on composition: put the subscription logic into a separate object, and make all real publishers use it.

1. Create concrete publisher classes. Each time something important happens inside a publisher, it must notify all its subscribers.
2. Implement the update notification methods in concrete subscriber classes. Most subscribers would need some context data about the event. It can be passed as an argument of the notification method.

But there’s another option. Upon receiving a notification, the subscriber can fetch any data directly from the notification. In this case, the publisher must pass itself via the update method. The less flexible option is to link a publisher to the subscriber permanently via the constructor.

1. The client must create all necessary subscribers and register them with proper publishers.

**Pros and Cons**

* *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.
* Subscribers are notified in random order.

**Relations with Other Patterns**

* [**Chain of Responsibility**](https://refactoring.guru/design-patterns/chain-of-responsibility), **[Command](https://refactoring.guru/design-patterns/command)**, [**Mediator**](https://refactoring.guru/design-patterns/mediator) and **[Observer](https://refactoring.guru/design-patterns/observer)** address various ways of connecting senders and receivers of requests:
  + *Chain of Responsibility* passes a request sequentially along a dynamic chain of potential receivers until one of them handles it.
  + *Command* establishes unidirectional connections between senders and receivers.
  + *Mediator* eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object.
  + *Observer* lets receivers dynamically subscribe to and unsubscribe from receiving requests.
* The difference between [**Mediator**](https://refactoring.guru/design-patterns/mediator) and **[Observer](https://refactoring.guru/design-patterns/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.

The primary goal of *Mediator* is to eliminate mutual dependencies among a set of system components. Instead, these components become dependent on a single mediator object. The goal of *Observer* is to establish dynamic one-way connections between objects, where some objects act as subordinates of others.

There’s a popular implementation of the *Mediator* pattern that relies on *Observer*. The mediator object plays the role of publisher, and the components act as subscribers which subscribe to and unsubscribe from the mediator’s events. When *Mediator* is implemented this way, it may look very similar to *Observer*.

When you’re confused, remember that you can implement the Mediator pattern in other ways. For example, you can permanently link all the components to the same mediator object. This implementation won’t resemble *Observer* but will still be an instance of the Mediator pattern.

Now imagine a program where all components have become publishers, allowing dynamic connections between each other. There won’t be a centralized mediator object, only a distributed set of observers.