☹ Problem

Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.

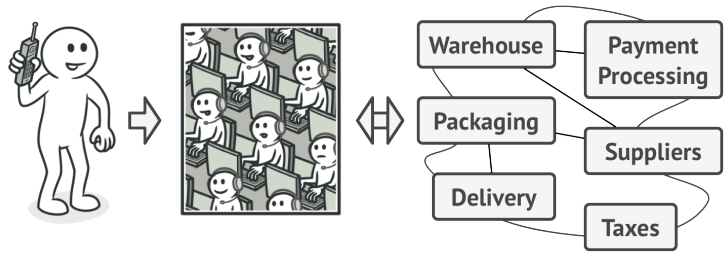
As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

😊 Solution

A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.

Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality. For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method `encode(filename, format)`. After creating such a class and connecting it with the video conversion library, you'll have your first facade.

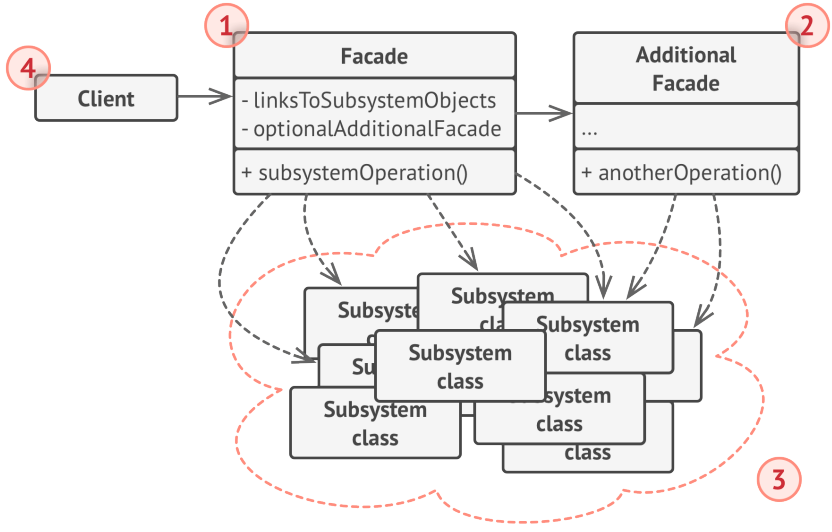
Real-World Analogy



Placing orders by phone.

When you call a shop to place a phone order, an operator is your facade to all services and departments of the shop. The operator provides you with a simple voice interface to the ordering system, payment gateways, and various delivery services.

Structure



1. The **Facade** provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.

2. An **Additional Facade** class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.

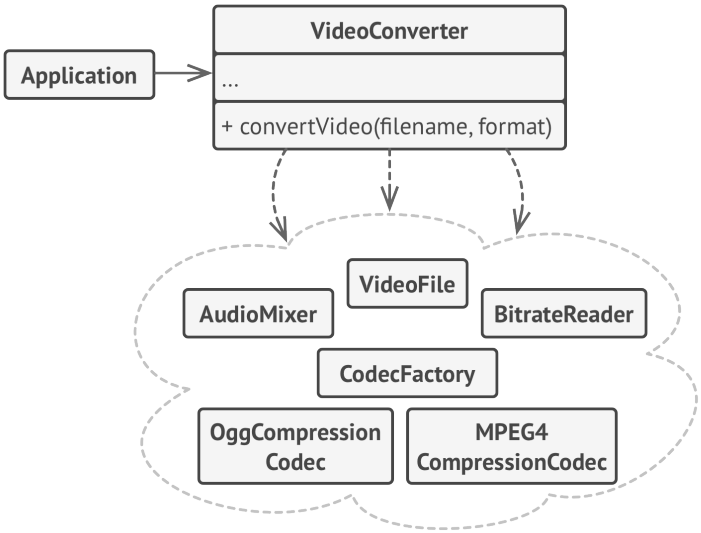
3. The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem's implementation details, such as initializing objects in the correct order and supplying them with data in the proper format.

   Subsystem classes aren't aware of the facade's existence. They operate within the system and work with each other directly.

4. The **Client** uses the facade instead of calling the subsystem objects directly.

Pseudocode

In this example, the **Facade** pattern simplifies interaction with a complex video conversion framework.



An example of isolating multiple dependencies within a single facade class.

Instead of making your code work with dozens of the framework classes directly, you create a facade class which encapsulates that functionality and hides it from the rest of the code. This structure also helps you to minimize the effort of upgrading to future versions of the framework or replacing it with another one. The only thing you'd need to change in your app would be the implementation of the facade's methods.

```
// These are some of the classes of a complex 3rd-party video
// conversion framework. We don't control that code, therefore
// can't simplify it.

class VideoFile
// ...
```

```
class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...


// We create a facade class to hide the framework's complexity
// behind a simple interface. It's a trade-off between
// functionality and simplicity.
class VideoConverter is
  method convert(filename, format):File is
    file = new VideoFile(filename)
    sourceCodec = new CodecFactory.extract(file)
    if (format == "mp4")
      destinationCodec = new MPEG4CompressionCodec()
    else
      destinationCodec = new OggCompressionCodec()
    buffer = BitrateReader.read(filename, sourceCodec)
    result = BitrateReader.convert(buffer, destinationCodec)
    result = (new AudioMixer()).fix(result)
    return new File(result)

```

```
// Application classes don't depend on a billion classes
// provided by the complex framework. Also, if you decide to
// switch frameworks, you only need to rewrite the facade class.
class Application is
  method main() is
    convertor = new VideoConverter()
    mp4 = convertor.convert("youtubevideo.ogg", "mp4")
    mp4.save()
```

Applicability

Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.

Often, subsystems get more complex over time. Even applying design patterns typically leads to creating more classes. A subsystem may become more flexible and easier to reuse in various contexts, but the amount of configuration and boilerplate code it demands from a client grows ever larger. The Facade attempts to fix this problem by providing a shortcut to the most-used features of the subsystem which fit most client requirements.

Use the Facade when you want to structure a subsystem into layers.

Create facades to define entry points to each level of a subsystem. You can reduce coupling between multiple subsystems by requiring them to communicate only through facades.

For example, let's return to our video conversion framework. It can be broken down into two layers: video- and audio-related. For each layer, you can create a facade and then make the classes of each layer communicate with each another via those facades. This approach looks very similar to the **Mediator** pattern.

How to Implement

1. Check whether it's possible to provide a simpler interface than what an existing subsystem already provides. You're on the right track if this interface makes the client code independent from many of the subsystem's classes.

2. Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem. The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.

3. To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade. Now the client code is protected from any changes in the subsystem code. For example, when a subsystem gets upgraded to a new version, you will only need to modify the code in the facade.

4. If the facade becomes **too big**, consider extracting part of its behavior to a new, refined facade class.

Pros and Cons

✓ You can isolate your code from the complexity of a subsystem.

✕ A facade can become **a god object** coupled to all classes of an app.

Relations with Other Patterns

- **Facade** defines a new interface for existing objects, whereas **Adapter** tries to make the existing interface usable. *Adapter* usually wraps just one object, while *Facade* works with an entire subsystem of objects.

- **Abstract Factory** can serve as an alternative to **Facade** when you only want to hide the way the subsystem objects are created from the client code.

- **Flyweight** shows how to make lots of little objects, whereas **Facade** shows how to make a single object that represents an entire subsystem.

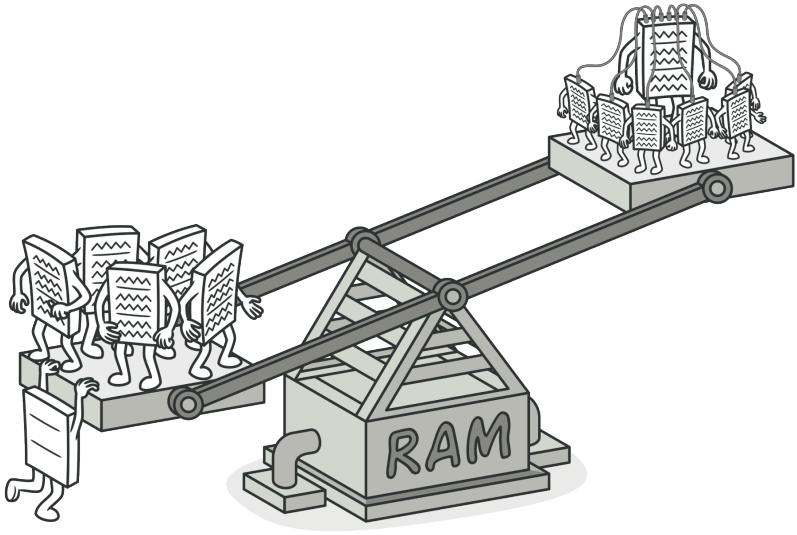
- **Facade** and **Mediator** have similar jobs: they try to organize collaboration between lots of tightly coupled classes.

    - *Facade* defines a simplified interface to a subsystem of objects, but it doesn't introduce any new functionality. The subsystem itself is unaware of the facade. Objects within the subsystem can communicate directly.

    - *Mediator* centralizes communication between components of the system. The components only know about the mediator object and don't communicate directly.

- A **Facade** class can often be transformed into a **Singleton** since a single facade object is sufficient in most cases.

- **Facade** is similar to **Proxy** in that both buffer a complex entity and initialize it on its own. Unlike *Facade*, *Proxy* has the same interface as its service object, which makes them interchangeable.



FLYWEIGHT

Also known as: Cache

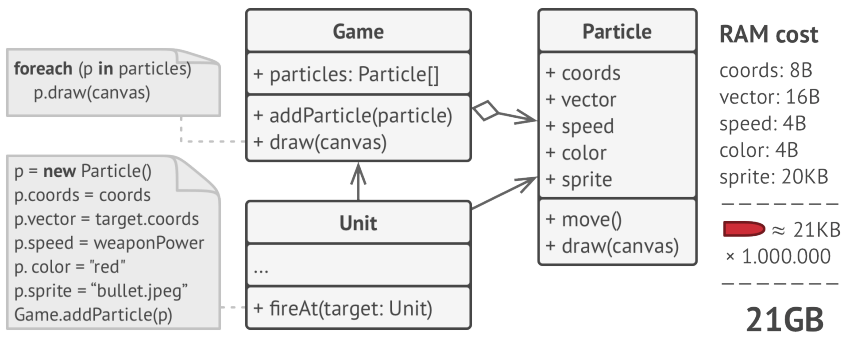
Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

☹ Problem

To have some fun after long working hours, you decided to create a simple video game: players would be moving around a map and shooting each other. You chose to implement a realistic particle system and make it a distinctive feature of the game. Vast quantities of bullets, missiles, and shrapnel from explosions should fly all over the map and deliver a thrilling experience to the player.

Upon its completion, you pushed the last commit, built the game and sent it to your friend for a test drive. Although the game was running flawlessly on your machine, your friend wasn't able to play for long. On his computer, the game kept crashing after a few minutes of gameplay.

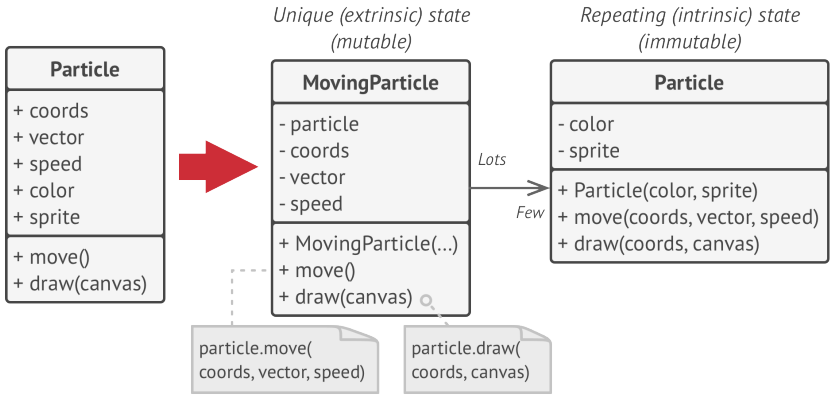
After spending several hours digging through debug logs, you discovered that the game crashed because of an insufficient amount of RAM. It turned out that your friend's rig was much less powerful than your own computer, and that's why the problem emerged so quickly on his machine.



The actual problem was related to your particle system. Each particle, such as a bullet, a missile or a piece of shrapnel was represented by a separate object containing plenty of data. At some point, when the carnage on a player's screen reached its climax, newly created particles no longer fit into the remaining RAM, so the program crashed.

😊 Solution

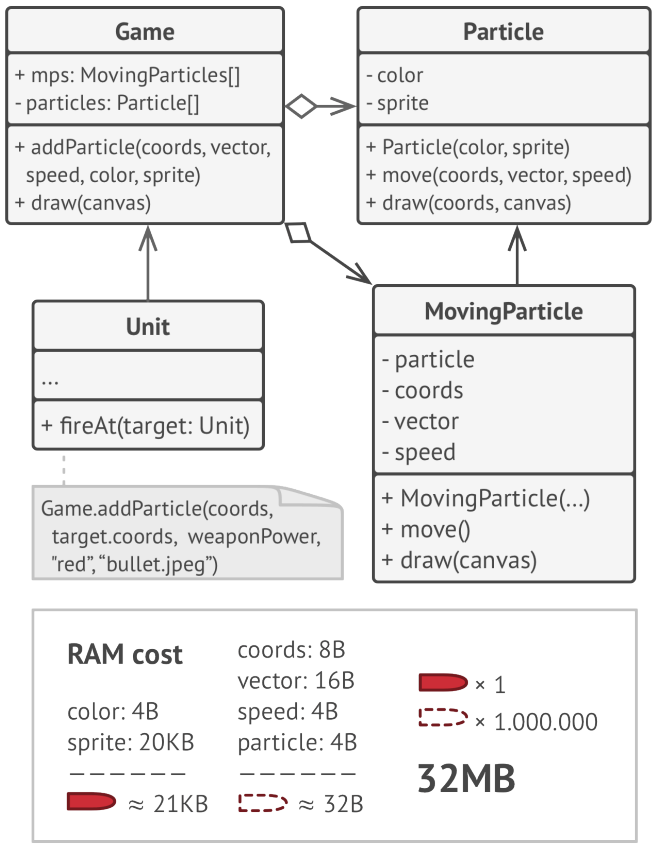
On closer inspection of the `Particle` class, you may notice that the color and sprite fields consume a lot more memory than other fields. What's worse is that these two fields store almost identical data across all particles. For example, all bullets have the same color and sprite.



Other parts of a particle's state, such as coordinates, movement vector and speed, are unique to each particle. After all, the values of these fields change over time. This data represents the always changing context in which the particle exists, while the color and sprite remain constant for each particle.

This constant data of an object is usually called the *intrinsic state*. It lives within the object; other objects can only read it, not change it. The rest of the object's state, often altered "from the outside" by other objects, is called the *extrinsic state*.

The Flyweight pattern suggests that you stop storing the extrinsic state inside the object. Instead, you should pass this state to specific methods which rely on it. Only the intrinsic state stays within the object, letting you reuse it in different contexts.



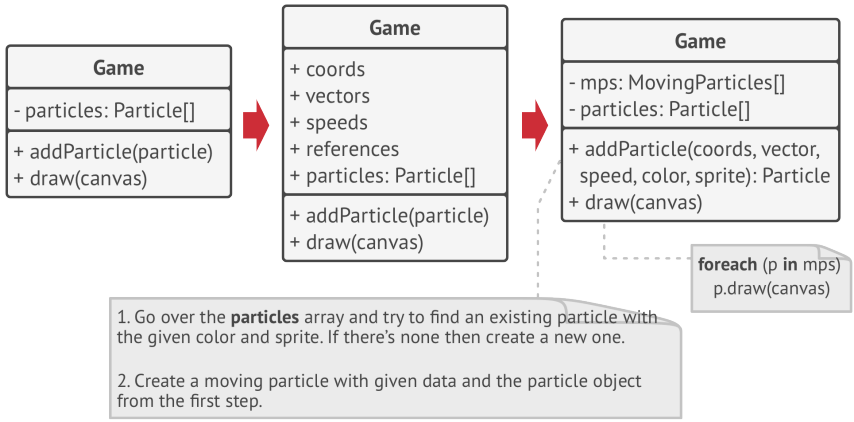
As a result, you'd need fewer of these objects since they only differ in the intrinsic state, which has much fewer variations than the extrinsic.

Let's return to our game. Assuming that we had extracted the extrinsic state from our particle class, only three different objects would suffice to represent all particles in the game: a bullet, a missile, and a piece of shrapnel. As you've probably guessed by now, an object that only stores the intrinsic state is called a flyweight.

Extrinsic state storage

Where does the extrinsic state move to? Some class should still store it, right? In most cases, it gets moved to the container object, which aggregates objects before we apply the pattern.

In our case, that's the main `Game` object that stores all particles in the `particles` field. To move the extrinsic state into this class, you need to create several array fields for storing coordinates, vectors, and speed of each individual particle. But that's not all. You need another array for storing references to a specific flyweight that represents a particle. These arrays must be in sync so that you can access all data of a particle using the same index.



A more elegant solution is to create a separate context class that would store the extrinsic state along with reference to the flyweight object. This approach would require having just a single array in the container class.

Wait a second! Won't we need to have as many of these contextual objects as we had at the very beginning? Technically, yes. But the thing is, these objects are much smaller than before. The most memory-consuming fields have been moved to just a few flyweight objects. Now, a thousand small contextual objects can reuse a single heavy flyweight object instead of storing a thousand copies of its data.

Flyweight and immutability

Since the same flyweight object can be used in different contexts, you have to make sure that its state can't be modified. A flyweight should initialize its state just once, via constructor

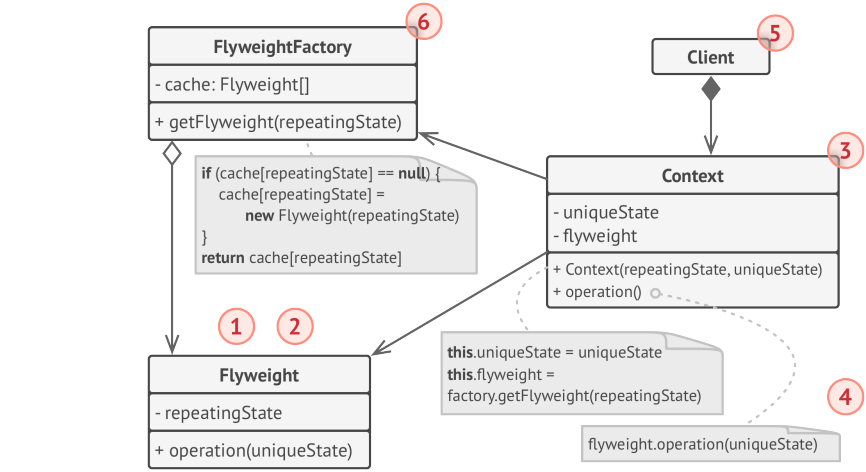
parameters. It shouldn't expose any setters or public fields to other objects.

Flyweight factory

For more convenient access to various flyweights, you can create a factory method that manages a pool of existing flyweight objects. The method accepts the intrinsic state of the desired flyweight from a client, looks for an existing flyweight object matching this state, and returns it if it was found. If not, it creates a new flyweight and adds it to the pool.

There are several options where this method could be placed. The most obvious place is a flyweight container. Alternatively, you could create a new factory class. Or you could make the factory method static and put it inside an actual flyweight class.

Structure



1. The Flyweight pattern is merely an optimization. Before applying it, make sure your program does have the RAM consumption problem related to having a massive number of similar objects in memory at the same time. Make sure that this problem can't be solved in any other meaningful way.

2. The **Flyweight** class contains the portion of the original object's state that can be shared between multiple objects. The same flyweight object can be used in many different contexts. The state stored inside a flyweight is called "intrinsic." The state passed to the flyweight's methods is called "extrinsic."

3. The **Context** class contains the extrinsic state, unique across all original objects. When a context is paired with one of the flyweight objects, it represents the full state of the original object.

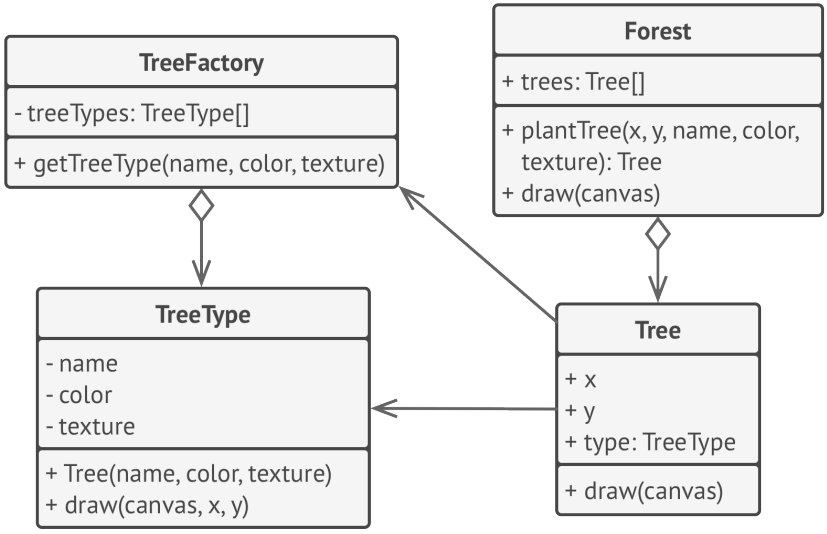
4. Usually, the behavior of the original object remains in the flyweight class. In this case, whoever calls a flyweight's method must also pass appropriate bits of the extrinsic state into the method's parameters. On the other hand, the behavior can be moved to the context class, which would use the linked flyweight merely as a data object.

5. The **Client** calculates or stores the extrinsic state of flyweights. From the client's perspective, a flyweight is a template object which can be configured at runtime by passing some contextual data into parameters of its methods.

6. The **Flyweight Factory** manages a pool of existing flyweights. With the factory, clients don't create flyweights directly. Instead, they call the factory, passing it bits of the intrinsic state of the desired flyweight. The factory looks over previously created flyweights and either returns an existing one that matches search criteria or creates a new one if nothing is found.

Pseudocode

In this example, the **Flyweight** pattern helps to reduce memory usage when rendering millions of tree objects on a canvas.



The pattern extracts the repeating intrinsic state from a main `Tree` class and moves it into the flyweight class `TreeType`.

Now instead of storing the same data in multiple objects, it's kept in just a few flyweight objects and linked to appropriate `Tree` objects which act as contexts. The client code creates new tree objects using the flyweight factory, which encapsulates the complexity of searching for the right object and reusing it if needed.

```
// The flyweight class contains a portion of the state of a
// tree. These fields store values that are unique for each
// particular tree. For instance, you won't find here the tree
// coordinates. But the texture and colors shared between many
// trees are here. Since this data is usually BIG, you'd waste a
// lot of memory by keeping it in each tree object. Instead, we
// can extract texture, color and other repeating data into a
// separate object which lots of individual tree objects can
// reference.
class TreeType is
  field name
  field color
  field texture
  constructor TreeType(name, color, texture) { ... }
  method draw(canvas, x, y) is
    // 1. Create a bitmap of a given type, color & texture.
    // 2. Draw the bitmap on the canvas at X and Y coords.

// Flyweight factory decides whether to re-use existing
// flyweight or to create a new object.
class TreeFactory is
  static field treeTypes: collection of tree types
  static method getTreeType(name, color, texture) is
    type = treeTypes.find(name, color, texture)
```

```
    if (type == null)
      type = new TreeType(name, color, texture)
      treeTypes.add(type)
    return type

// The contextual object contains the extrinsic part of the tree
// state. An application can create billions of these since they
// are pretty small: just two integer coordinates and one
// reference field.
class Tree is
  field x,y
  field type: TreeType
  constructor Tree(x, y, type) { ... }
  method draw(canvas) is
    type.draw(canvas, this.x, this.y)

// The Tree and the Forest classes are the flyweight's clients.
// You can merge them if you don't plan to develop the Tree
// class any further.
class Forest is
  field trees: collection of Trees

  method plantTree(x, y, name, color, texture) is
    type = TreeFactory.getTreeType(name, color, texture)
    tree = new Tree(x, y, type)
    trees.add(tree)

  method draw(canvas) is
    foreach (tree in trees) do
      tree.draw(canvas)
```

Applicability

Use the Flyweight pattern only when your program must support a huge number of objects which barely fit into available RAM.

The benefit of applying the pattern depends heavily on how and where it's used. It's most useful when:

- an application needs to spawn a huge number of similar objects
- this drains all available RAM on a target device
- the objects contain duplicate states which can be extracted and shared between multiple objects

How to Implement

1. Divide fields of a class that will become a flyweight into two parts:
    - the intrinsic state: the fields that contain unchanging data duplicated across many objects
    - the extrinsic state: the fields that contain contextual data unique to each object
2. Leave the fields that represent the intrinsic state in the class, but make sure they're immutable. They should take their initial values only inside the constructor.

3. Go over methods that use fields of the extrinsic state. For each field used in the method, introduce a new parameter and use it instead of the field.

4. Optionally, create a factory class to manage the pool of flyweights. It should check for an existing flyweight before creating a new one. Once the factory is in place, clients must only request flyweights through it. They should describe the desired flyweight by passing its intrinsic state to the factory.

5. The client must store or calculate values of the extrinsic state (context) to be able to call methods of flyweight objects. For the sake of convenience, the extrinsic state along with the flyweight-referencing field may be moved to a separate context class.

 Pros and Cons

✓ You can save lots of RAM, assuming your program has tons of similar objects.

✕ You might be trading RAM over CPU cycles when some of the context data needs to be recalculated each time somebody calls a flyweight method.

✕ The code becomes much more complicated. New team members will always be wondering why the state of an entity was separated in such a way.

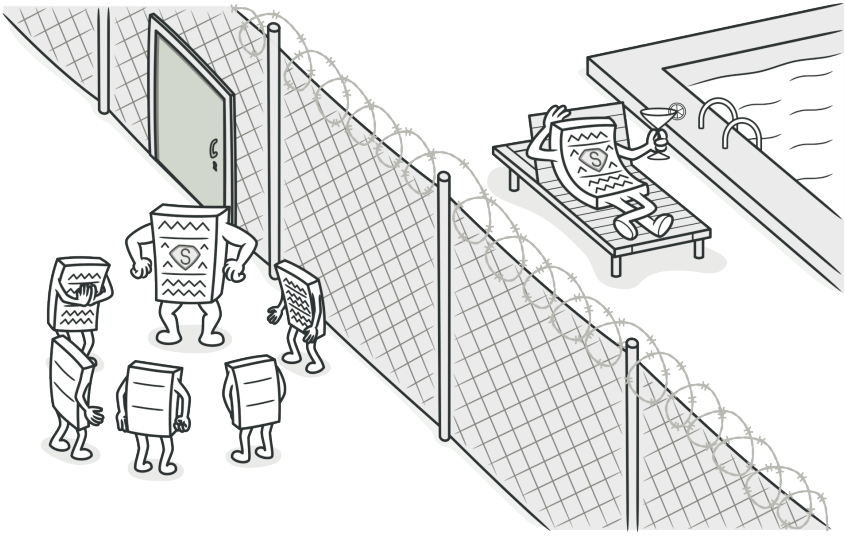
⇄ Relations with Other Patterns

- You can implement shared leaf nodes of the **Composite** tree as **Flyweights** to save some RAM.

- **Flyweight** shows how to make lots of little objects, whereas **Facade** shows how to make a single object that represents an entire subsystem.

- **Flyweight** would resemble **Singleton** if you somehow managed to reduce all shared states of the objects to just one flyweight object. But there are two fundamental differences between these patterns:

    1. There should be only one Singleton instance, whereas a *Flyweight* class can have multiple instances with different intrinsic states.
    2. The *Singleton* object can be mutable. Flyweight objects are immutable.

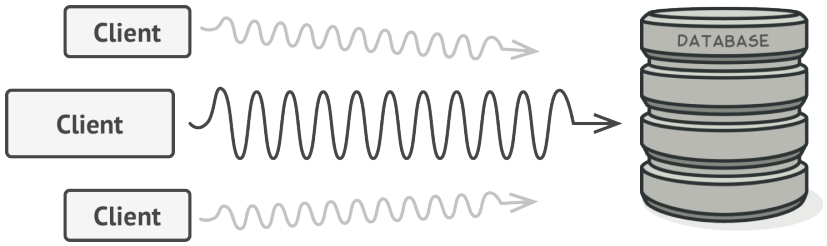


PROXY

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

☹ Problem

Why would you want to control access to an object? Here is an example: you have a massive object that consumes a vast amount of system resources. You need it from time to time, but not always.



Database queries can be really slow.

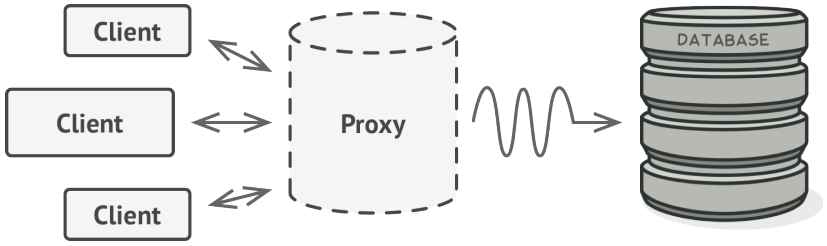
You could implement lazy initialization: create this object only when it's actually needed. All of the object's clients would need to execute some deferred initialization code. Unfortunately, this would probably cause a lot of code duplication.

In an ideal world, we'd want to put this code directly into our object's class, but that isn't always possible. For instance, the class may be part of a closed 3rd-party library.

☺ Solution

The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the

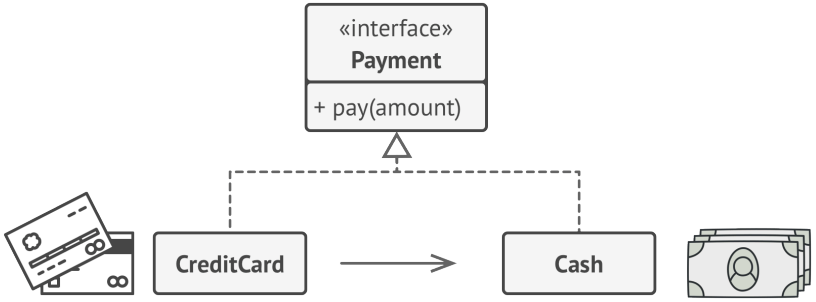
original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.



The proxy disguises itself as a database object. It can handle lazy initialization and result caching without the client or the real database object even knowing.

But what's the benefit? If you need to execute something either before or after the primary logic of the class, the proxy lets you do this without changing that class. Since the proxy implements the same interface as the original class, it can be passed to any client that expects a real service object.

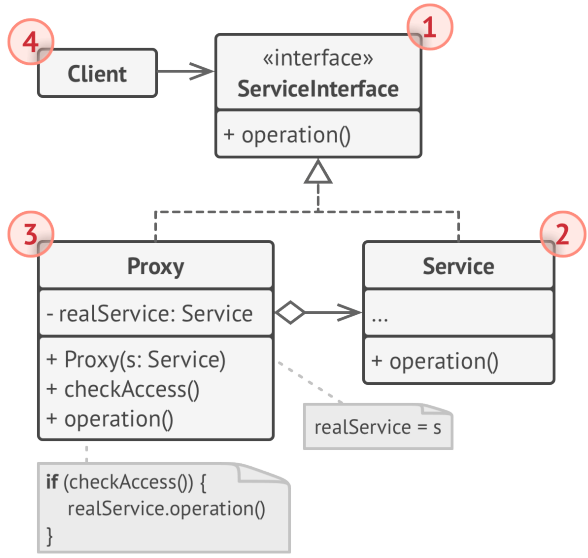
Real-World Analogy



Credit cards can be used for payments just the same as cash.

A credit card is a proxy for a bank account, which is a proxy for a bundle of cash. Both implement the same interface: they can be used for making a payment. A consumer feels great because there's no need to carry loads of cash around. A shop owner is also happy since the income from a transaction gets added electronically to the shop's bank account without the risk of losing the deposit or getting robbed on the way to the bank.

Structure



1. The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

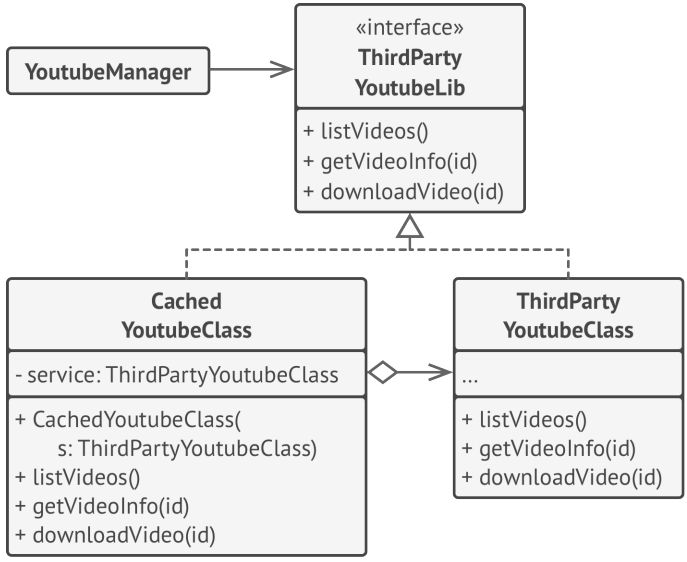
2. The **Service** is a class that provides some useful business logic.

3. The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object. Usually, proxies manage the full lifecycle of their service objects.

4. The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.

Pseudocode

This example illustrates how the **Proxy** pattern can help to introduce lazy initialization and caching to a 3rd-party YouTube integration library.



Caching results of a service with a proxy.

The library provides us with the video downloading class. However, it's very inefficient. If the client application requests the same video multiple times, the library just downloads it over and over, instead of caching and reusing the first downloaded file.

The proxy class implements the same interface as the original downloader and delegates it all the work. However, it keeps track of the downloaded files and returns the cached result when the app requests the same video multiple times.

```
// The interface of a remote service.
interface ThirdPartyYoutubeLib is
  method listVideos()
  method getVideoInfo(id)
  method downloadVideo(id)

// The concrete implementation of a service connector. Methods
// of this class can request information from YouTube. The speed
// of the request depends on a user's internet connection as
// well as YouTube's. The application will slow down if a lot of
// requests are fired at the same time, even if they all request
// the same information.
class ThirdPartyYoutubeClass implements ThirdPartyYoutubeLib is
  method listVideos() is
    // Send an API request to YouTube.

  method getVideoInfo(id) is
    // Get metadata about some video.

```

```
    method downloadVideo(id) is
      // Download a video file from YouTube.

// To save some bandwidth, we can cache request results and keep
// them for some time. But it may be impossible to put such code
// directly into the service class. For example, it could have
// been provided as part of a third party library and/or defined
// as `final`. That's why we put the caching code into a new
// proxy class which implements the same interface as the
// service class. It delegates to the service object only when
// the real requests have to be sent.
class CachedYoutubeClass implements ThirdPartyYouTubeLib is
  private field service: ThirdPartyYouTubeClass
  private field listCache, videoCache
  field needReset

  constructor CachedYoutubeClass(service: ThirdPartyYouTubeLib) is
    this.service = service

  method listVideos() is
    if (listCache == null || needReset)
      listCache = service.listVideos()
    return listCache

  method getVideoInfo(id) is
    if (videoCache == null || needReset)
      videoCache = service.getVideoInfo(id)
    return videoCache

  method downloadVideo(id) is
    if (!downloadExists(id) || needReset)
      service.downloadVideo(id)
```

```
// The GUI class, which used to work directly with a service
// object, stays unchanged as long as it works with the service
// object through an interface. We can safely pass a proxy
// object instead of a real service object since they both
// implement the same interface.
class YoutubeManager is
  protected field service: ThirdPartyYouTubeLib

  constructor YoutubeManager(service: ThirdPartyYouTubeLib) is
    this.service = service

  method renderVideoPage(id) is
    info = service.getVideoInfo(id)
    // Render the video page.

  method renderListPanel() is
    list = service.listVideos()
    // Render the list of video thumbnails.

  method reactOnUserInput() is
    renderVideoPage()
    renderListPanel()

// The application can configure proxies on the fly.
class Application is
  method init() is
    aYouTubeService = new ThirdPartyYouTubeClass()
    aYouTubeProxy = new CachedYouTubeClass(aYouTubeService)
    manager = new YouTubeManager(aYouTubeProxy)
    manager.reactOnUserInput()
```

Applicability

There are dozens of ways to utilize the Proxy pattern. Let's go over the most popular uses.

Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.

Instead of creating the object when the app launches, you can delay the object's initialization to a time when it's really needed.

Access control (protection proxy). This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones).

The proxy can pass the request to the service object only if the client's credentials match some criteria.

Local execution of a remote service (remote proxy). This is when the service object is located on a remote server.

In this case, the proxy passes the client request over the network, handling all of the nasty details of working with the network.

- Logging requests (logging proxy). This is when you want to keep a history of requests to the service object.

  The proxy can log each request before passing it to the service.

- **Caching request results (caching proxy). This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.**

  The proxy can implement caching for recurring requests that always yield the same results. The proxy may use the parameters of requests as the cache keys.

- Smart reference. This is when you need to be able to dismiss a heavyweight object once there are no clients that use it.

  The proxy can keep track of clients that obtained a reference to the service object or its results. From time to time, the proxy may go over the clients and check whether they are still active. If the client list gets empty, the proxy might dismiss the service object and free the underlying system resources.

  The proxy can also track whether the client had modified the service object. Then the unchanged objects may be reused by other clients.

How to Implement

1. If there's no pre-existing service interface, create one to make proxy and service objects interchangeable. Extracting the interface from the service class isn't always possible, because you'd need to change all of the service's clients to use that interface. Plan B is to make the proxy a subclass of the service class, and this way it'll inherit the interface of the service.

2. Create the proxy class. It should have a field for storing a reference to the service. Usually, proxies create and manage the whole life cycle of their servers. In rare occasions, a service is passed to the proxy via a constructor by the client.

3. Implement the proxy methods according to their purposes. In most cases, after doing some work, the proxy should delegate the work to the service object.

4. Consider introducing a creation method that decides whether the client gets a proxy or a real service. This can be a simple static method in the proxy class or a full-blown factory method.

5. Consider implementing lazy initialization for the service object.

Pros and Cons

✓ You can control the service object without clients knowing about it.

✓ You can manage the lifecycle of the service object when clients don't care about it.

✓ The proxy works even if the service object isn't ready or is not available.

✓ *Open/Closed Principle*. You can introduce new proxies without changing the service or clients.

✕ The code may become more complicated since you need to introduce a lot of new classes.

✕ The response from the service might get delayed.

Relations with Other Patterns

- **Adapter** provides a different interface to the wrapped object, **Proxy** provides it with the same interface, and **Decorator** provides it with an enhanced interface.

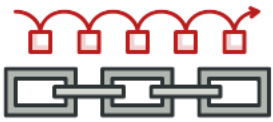
- **Facade** is similar to **Proxy** in that both buffer a complex entity and initialize it on its own. Unlike *Facade*, *Proxy* has the same interface as its service object, which makes them interchangeable.

- **Decorator** and **Proxy** have similar structures, but very different intents. Both patterns are built on the composition principle,

where one object is supposed to delegate some of the work to another. The difference is that a *Proxy* usually manages the life cycle of its service object on its own, whereas the composition of *Decorators* is always controlled by the client.

Behavioral Design Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.



Chain of Responsibility

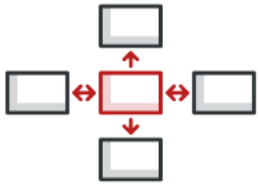
Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.

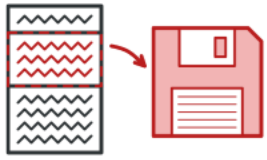
Iterator

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



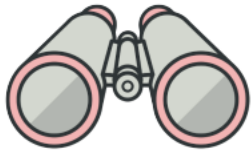
Mediator

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.



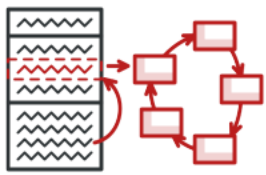
Memento

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



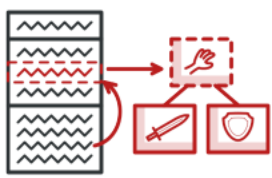
Observer

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

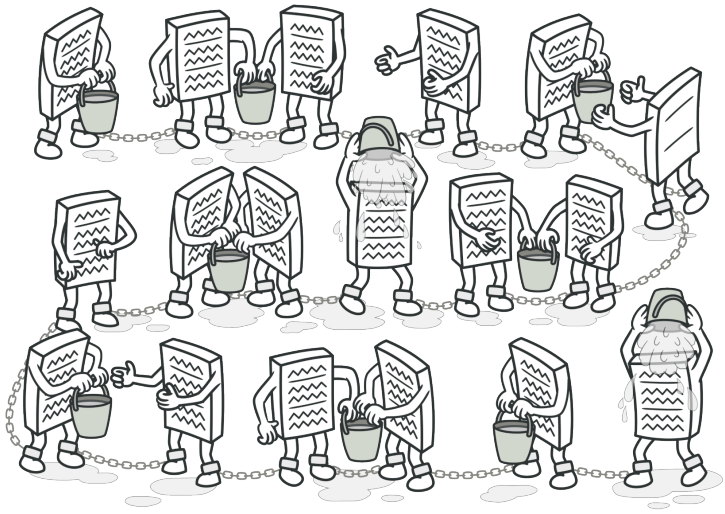
Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

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

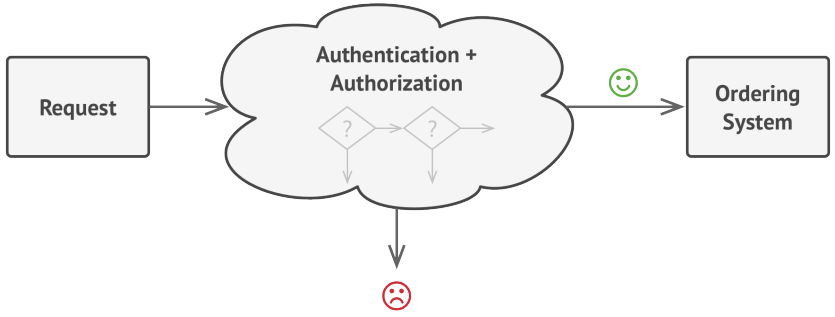
Also known as: CoR, Chain of Command

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 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.

After a bit of planning, you realized that these checks must be performed sequentially. The application can attempt to authenticate a user to the system whenever it receives a request that contains the user's credentials. However, if those credentials aren't correct and authentication fails, there's no reason to proceed with any other checks.



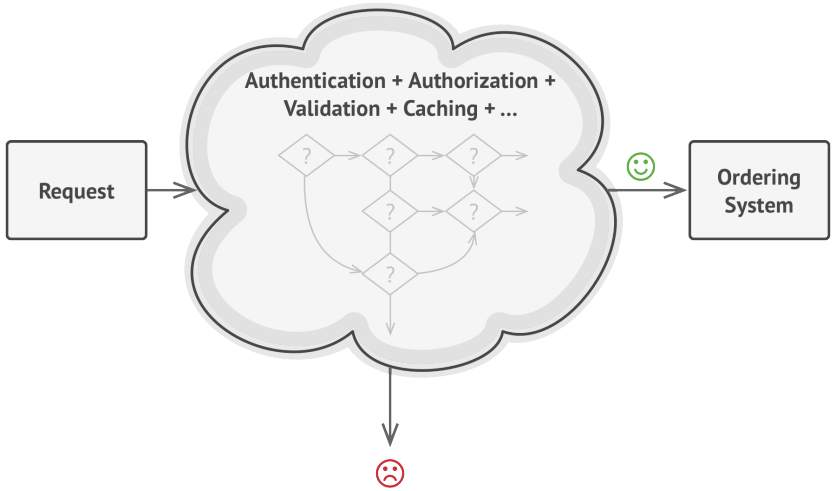
The request must pass a series of checks before the ordering system itself can handle it.

During the next few months, you implemented several more of those sequential checks.

- One of your colleagues suggested that it's unsafe to pass raw data straight to the ordering system. So you added an extra validation step to sanitize the data in a request.

- Later, somebody noticed that the system is vulnerable to brute force password cracking. To negate this, you promptly added a check that filters repeated failed requests coming from the same IP address.

- Someone else suggested that you could speed up the system by returning cached results on repeated requests containing the same data. Hence, you added another check which lets the request pass through to the system only if there's no suitable cached response.



The bigger the code grew, the messier it became.

The code of the checks, which had already looked like a mess, became more and more 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.

The system became very hard to comprehend and expensive to maintain. You struggled with the code for a while, until one day you decided to refactor the whole thing.

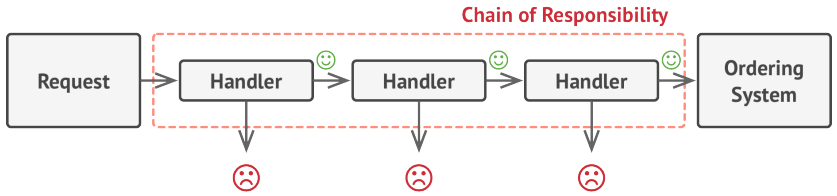
😊 Solution

Like many other behavioral design patterns, 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 own class with a single method that performs the check. The request, along with its data, is passed to this method as an argument.

The pattern suggests that you link these handlers into a chain. Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain. The request travels along the chain until all handlers have had a chance to process it.

Here's the best part: a handler can decide not to pass the request further down the chain and effectively stop any further processing.

In our example with ordering systems, a handler performs the processing and then decides whether to pass the request further down the chain. Assuming the request contains the right data, all the handlers can execute their primary behavior, whether it's authentication checks or caching.

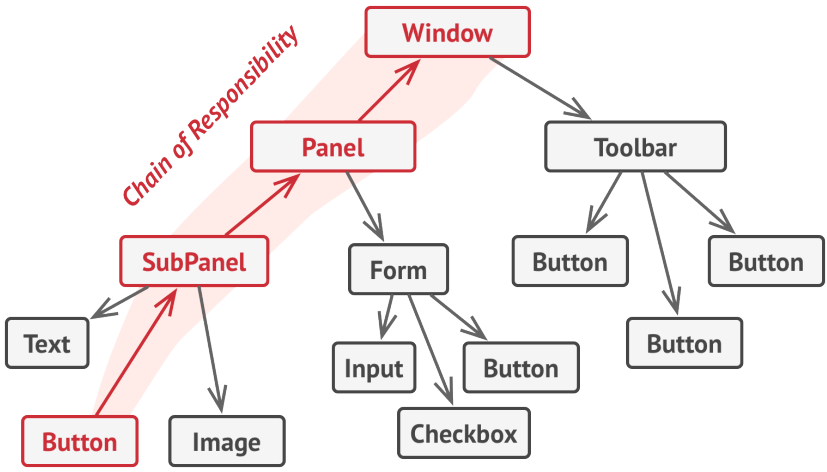


Handlers are lined up one by one, forming a chain.

However, there's a slightly different approach (and it's a bit more canonical) in which, upon receiving a request, a handler decides whether it can process it. If it can, it doesn't pass the request any further. So it's either only one handler that processes the request or none at all. This approach is very common when dealing with events in stacks of elements within a graphical user interface.

For instance, when a user clicks a button, the event propagates through the chain of GUI elements that starts with the button, goes along its containers (like forms or panels), and ends up with the main application window. The event is processed by the first element in the chain that's capable of handling it. This

example is also noteworthy because it shows that a chain can always be extracted from an object tree.



A chain can be formed from a branch of an object tree.

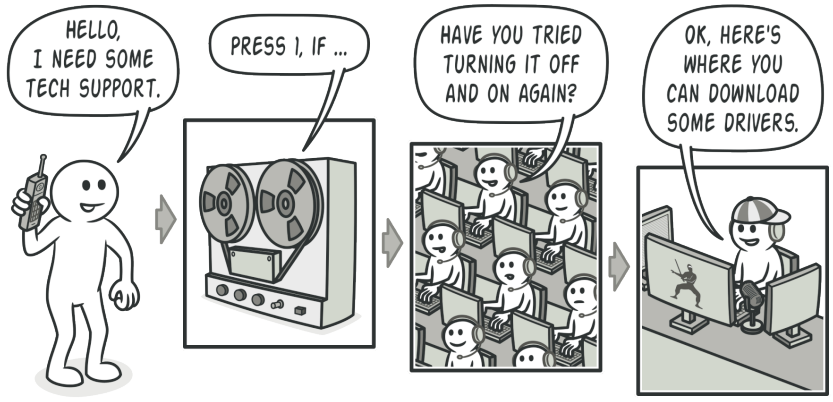
It's crucial that all handler classes implement the same interface. Each concrete handler should only care about the following one having the `execute` method. This way you can compose chains at runtime, using various handlers without coupling your code to their concrete classes.

Real-World Analogy

You've just bought and installed a new piece of hardware on your computer. Since you're a geek, the computer has several operating systems installed. You try to boot all of them to see whether the hardware is supported. Windows detects and enables the hardware automatically. However, your beloved Linux refuses to work with the new hardware. With a small

flicker of hope, you decide to call the tech-support phone number written on the box.

The first thing you hear is the robotic voice of the autoresponder. It suggests nine popular solutions to various problems, none of which are relevant to your case. After a while, the robot connects you to a live operator.



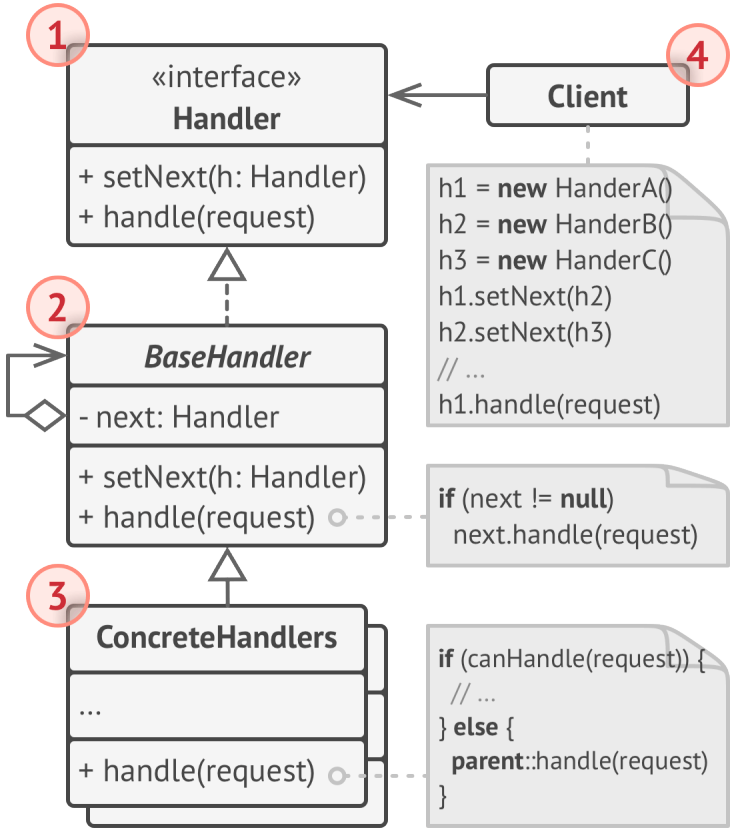
A call to tech support can go through multiple operators.

Alas, the operator isn't able to suggest anything specific either. He keeps quoting lengthy excerpts from the manual, refusing to listen to your comments. After hearing the phrase "have you tried turning the computer off and on again?" for the 10th time, you demand to be connected to a proper engineer.

Eventually, the operator passes your call to one of the engineers, who had probably longed for a live human chat for hours as he sat in his lonely server room in the dark basement of some office building. The engineer tells you where to down-

load proper drivers for your new hardware and how to install them on Linux. Finally, the solution! You end the call, bursting with joy.

Structure



1. 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.

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

   Usually, this class defines a field for storing a reference to the next handler. The clients can build a chain by passing a handler to the constructor or setter of the previous handler. The class may also implement the default handling behavior: it can pass execution to the next handler after checking for its existence.

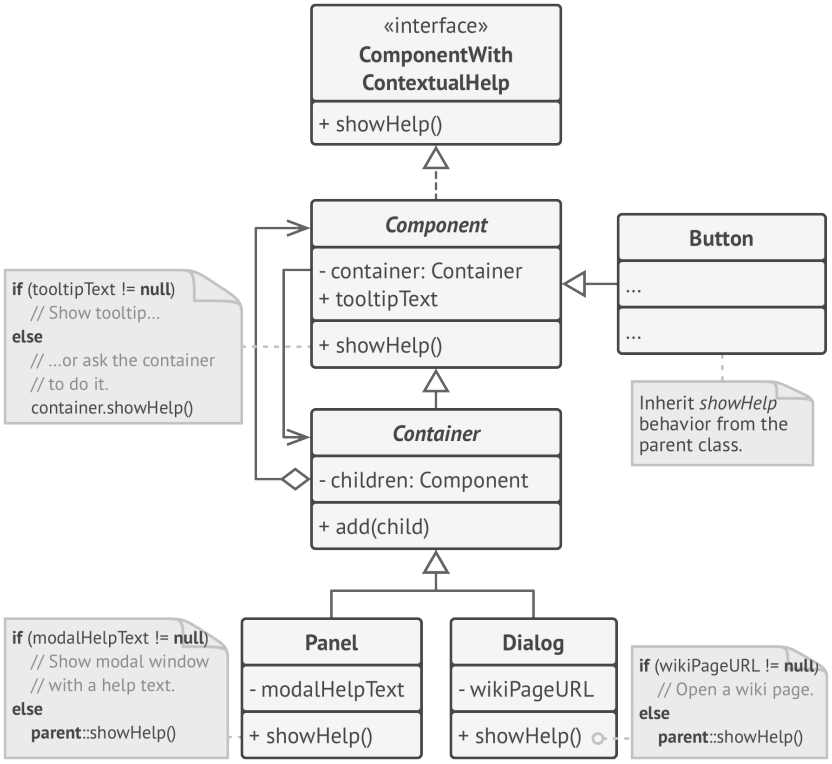
3. **Concrete Handlers** contain the actual code for processing requests. Upon receiving a request, each handler must decide whether to process it and, additionally, whether to pass it along the chain.

   Handlers are usually self-contained and immutable, accepting all necessary data just once via the constructor.

4. 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.

Pseudocode

In this example, the **Chain of Responsibility** pattern is responsible for displaying contextual help information for active GUI elements.

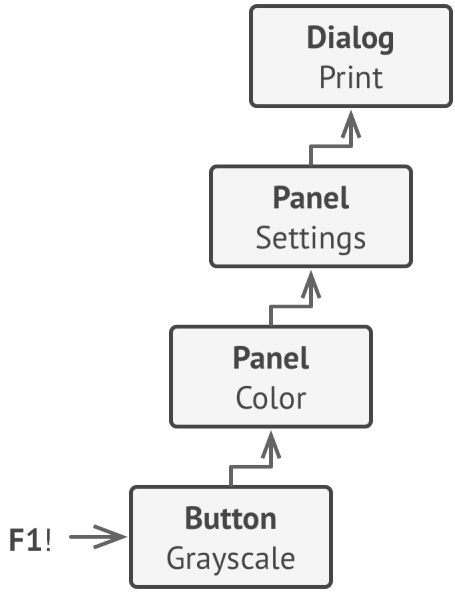


The GUI classes are built with the Composite pattern. Each element is linked to its container element. At any point, you can build a chain of elements that starts with the element itself and goes through all of its container elements.

The application's GUI is usually structured as an object tree. For example, the `Dialog` class, which renders the main window of the app, would be the root of the object tree. The dialog contains `Panels`, which might contain other panels or simple low-level elements like `Buttons` and `TextFields`.

A simple component can show brief contextual tooltips, as long as the component has some help text assigned. But more

complex components define their own way of showing contextual help, such as showing an excerpt from the manual or opening a page in a browser.



That's how a help request traverses GUI objects.

When a user points the mouse cursor at an element and presses the `F1` key, the application detects the component under the pointer and sends it a help request. The request bubbles up through all the element's containers until it reaches the element that's capable of displaying the help information.

```
// The handler interface declares a method for building a chain
// of handlers. It also declares a method for executing a
// request.
interface ComponentWithContextualHelp is
  method showHelp()


// The base class for simple components.
abstract class Component implements ComponentWithContextualHelp is
  field tooltipText: string

  // The component's container acts as the next link in the
  // chain of handlers.
  protected field container: Container

  // The component shows a tooltip if there's help text
  // assigned to it. Otherwise it forwards the call to the
  // container, if it exists.
  method showHelp() is
    if (tooltipText != null)
      // Show tooltip.
    else
      container.showHelp()


// Containers can contain both simple components and other
// containers as children. The chain relationships are
// established here. The class inherits showHelp behavior from
// its parent.
abstract class Container extends Component is
  protected field children: array of Component

```

```
  method add(child) is
    children.add(child)
    child.container = this


// Primitive components may be fine with default help
// implementation...
class Button extends Component is
  // ...

// But complex components may override the default
// implementation. If the help text can't be provided in a new
// way, the component can always call the base implementation
// (see Component class).
class Panel extends Container is
  field modalHelpText: string

  method showHelp() is
    if (modalHelpText != null)
      // Show a modal window with the help text.
    else
      super.showHelp()

// ...same as above...
class Dialog extends Container is
  field wikiPageURL: string

  method showHelp() is
    if (wikiPageURL != null)
      // Open the wiki help page.
    else
      super.showHelp()
```

```
// Client code.
class Application is
  // Every application configures the chain differently.
  method createUI() is
    dialog = new Dialog("Budget Reports")
    dialog.wikiPageURL = "http://..."
    panel = new Panel(0, 0, 400, 800)
    panel.modalHelpText = "This panel does..."
    ok = new Button(250, 760, 50, 20, "OK")
    ok.tooltipText = "This is an OK button that..."
    cancel = new Button(320, 760, 50, 20, "Cancel")
    // ...
    panel.add(ok)
    panel.add(cancel)
    dialog.add(panel)

  // Imagine what happens here.
  method onF1KeyPress() is
    component = this.getComponentAtMouseCoords()
    component.showHelp()
```

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.

The pattern lets you link several handlers into one chain and, upon receiving a request, “ask” each handler whether it can

process it. This way all handlers get a chance to process the request.

Use the pattern when it's essential to execute several handlers in a particular order.

Since you can link the handlers in the chain in any order, all requests will get through the chain exactly as you planned.

Use the CoR pattern when the set of handlers and their order are supposed to change at runtime.

If you provide setters for a reference field inside the handler classes, you'll be able to insert, remove or reorder handlers dynamically.

How to Implement

1. Declare the handler interface and describe the signature of a method for handling requests.

   Decide how the client will pass the request data into the method. The most flexible way is to convert the request into an object and pass it to the handling method as an argument.

2. To eliminate duplicate boilerplate code in concrete handlers, it might be worth creating an abstract base handler class, derived from the handler interface.

This class should have a field for storing a reference to the next handler in the chain. Consider making the class immutable. However, if you plan to modify chains at runtime, you need to define a setter for altering the value of the reference field.

You can also implement the convenient default behavior for the handling method, which is to forward the request to the next object unless there's none left. Concrete handlers will be able to use this behavior by calling the parent method.

3. One by one create concrete handler subclasses and implement their handling methods. Each handler should make two decisions when receiving a request:

    - Whether it'll process the request.
    - Whether it'll pass the request along the chain.

4. The client may either assemble chains on its own or receive pre-built chains from other objects. In the latter case, you must implement some factory classes to build chains according to the configuration or environment settings.

5. The client may trigger any handler in the chain, not just the first one. The request will be passed along the chain until some handler refuses to pass it further or until it reaches the end of the chain.

6. Due to the dynamic nature of the chain, the client should be ready to handle the following scenarios:

    - The chain may consist of a single link.
    - Some requests may not reach the end of the chain.
    - Others may reach the end of the chain unhandled.

⚖ Pros and Cons

✓ 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.

✕ Some requests may end up unhandled.

⇄ Relations with Other Patterns

- **Chain of Responsibility**, **Command**, **Mediator** and **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.

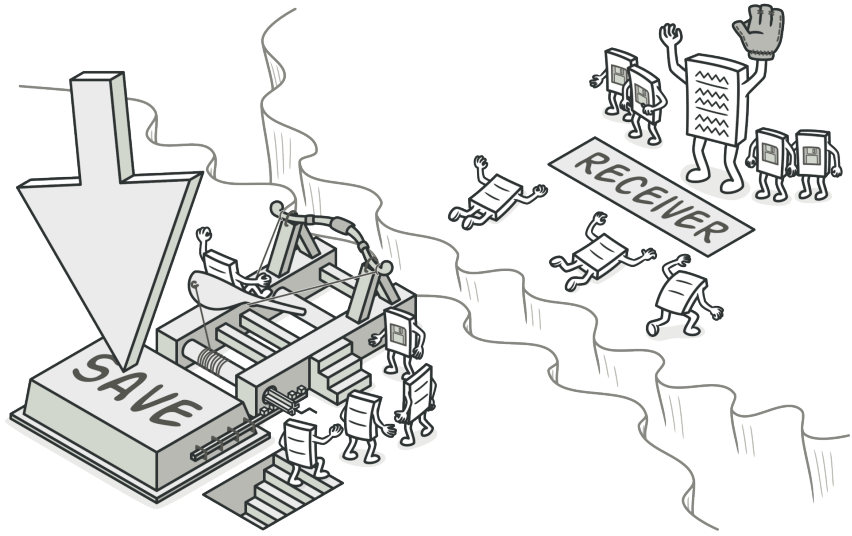
- **Chain of Responsibility** is often used in conjunction with **Composite**. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree.

- 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.

  However, there's another approach, where the request itself is a *Command* object. In this case, you can execute the same operation in a series of different contexts linked into a chain.

- **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.

  The *CoR* handlers can execute arbitrary operations independently of each other. They can also stop passing the request further at any point. On the other hand, various *Decorators* can extend the object's behavior while keeping it consistent with the base interface. In addition, decorators aren't allowed to break the flow of the request.



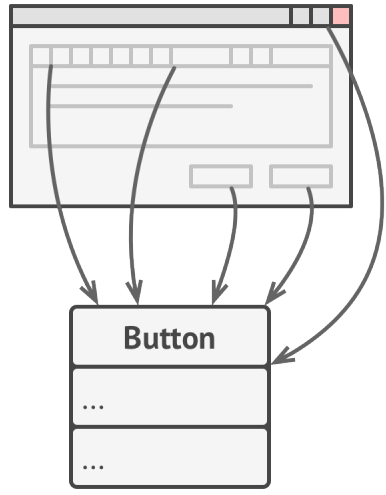
COMMAND

Also known as: Action, Transaction

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.

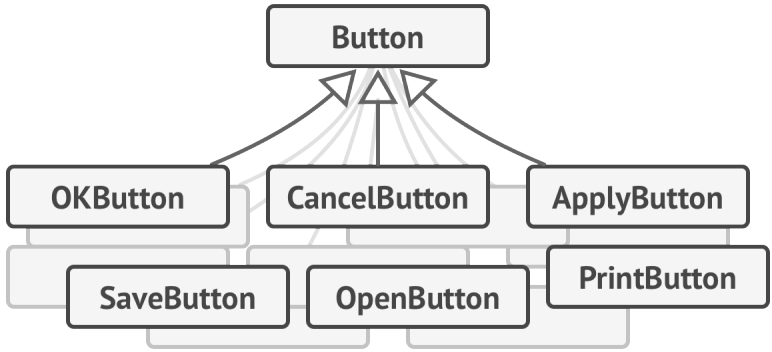
☹ Problem

Imagine that you're working on a new text-editor app. Your current task is to create a toolbar with a bunch of buttons for various operations of the editor. You created a very neat `Button` class that can be used for buttons on the toolbar, as well as for generic buttons in various dialogs.



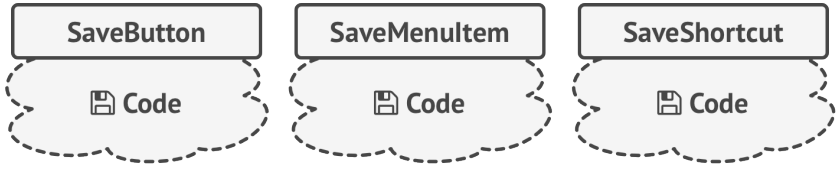
All buttons of the app are derived from the same class.

While all of these buttons look similar, they're all supposed to do different things. Where would you put the code for the various click handlers of these buttons? The simplest solution is to create tons of subclasses for each place where the button is used. These subclasses would contain the code that would have to be executed on a button click.



Lots of button subclasses. What can go wrong?

Before long, you realize that this approach is deeply flawed. First, you have an enormous number of subclasses, and that would be okay if you weren't risking breaking the code in these subclasses each time you modify the base `Button` class. Put simply, your GUI code has become awkwardly dependent on the volatile code of the business logic.



Several classes implement the same functionality.

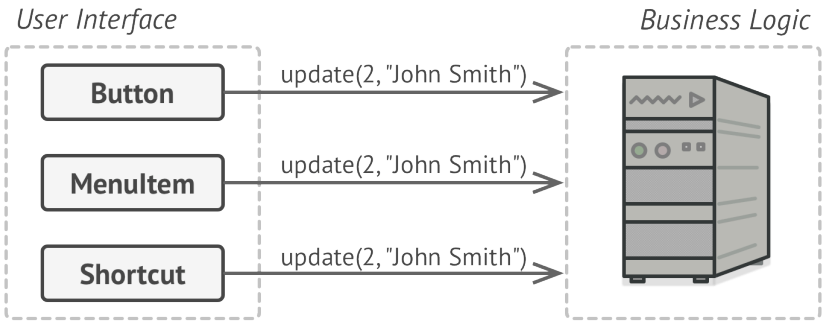
And here's the ugliest part. Some operations, such as copying/pasting text, would need to be invoked from multiple places. For example, a user could click a small "Copy" button on the toolbar, or copy something via the context menu, or just hit `Ctrl+C` on the keyboard.

Initially, when our app only had the toolbar, it was okay to place the implementation of various operations into the button subclasses. In other words, having the code for copying text inside the `CopyButton` subclass was fine. But then, when you implement context menus, shortcuts, and other stuff, you have to either duplicate the operation's code in many classes or make menus dependent on buttons, which is an even worse option.

😊 Solution

Good software design is often based on the principle of separation of concerns, which usually results in breaking an app into layers. The most common example: a layer for the graphical user interface and another layer for the business logic. The GUI layer is responsible for rendering a beautiful picture on the screen, capturing any input and showing results of what the user and the app are doing. However, when it comes to doing something important, like calculating the trajectory of the moon or composing an annual report, the GUI layer delegates the work to the underlying layer of business logic.

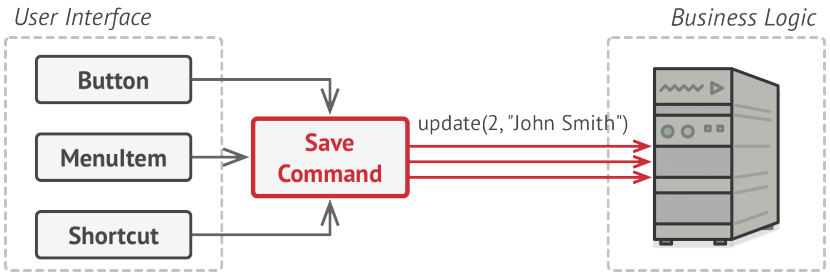
In the code it might look like this: a GUI object calls a method of a business logic object, passing it some arguments. This process is usually described as one object sending another a *request*.



The GUI objects may access the business logic objects directly.

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.

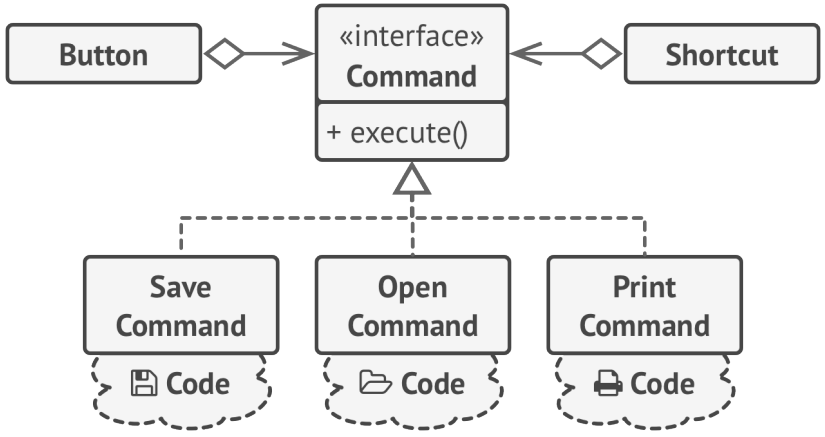
Command objects serve as links between various GUI and business logic objects. From now on, the GUI object doesn't need to know what business logic object will receive the request and how it'll be processed. The GUI object just triggers the command, which handles all the details.



Accessing the business logic layer via a command.

The next step is to make your commands implement the same interface. Usually it has just a single execution method that takes no parameters. This interface lets you use various commands with the same request sender, without coupling it to concrete classes of commands. As a bonus, now you can switch command objects linked to the sender, effectively changing the sender's behavior at runtime.

You might have noticed one missing piece of the puzzle, which is the request parameters. A GUI object might have supplied the business-layer object with some parameters. Since the command execution method doesn't have any parameters, how would we pass the request details to the receiver? It turns out the command should be either pre-configured with this data, or capable of getting it on its own.



The GUI objects delegate the work to commands.

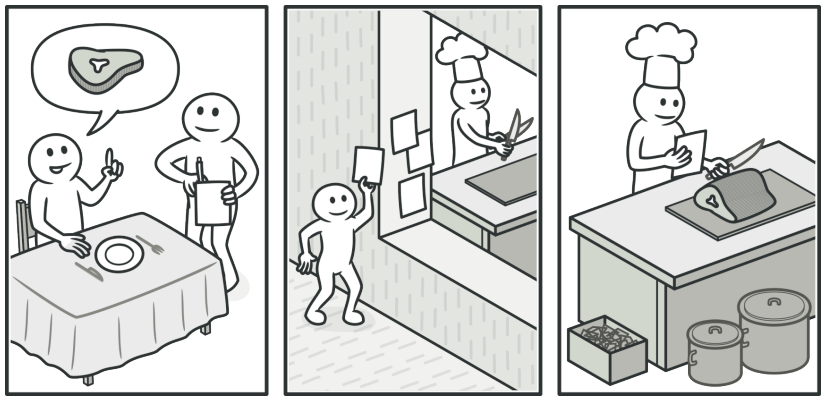
Let's get back to our text editor. After we apply the Command pattern, we no longer need all those button subclasses to implement various click behaviors. It's enough to put a single field into the base `Button` class that stores a reference to a command object and make the button execute that command on a click.

You'll implement a bunch of command classes for every possible operation and link them with particular buttons, depending on the buttons' intended behavior.

Other GUI elements, such as menus, shortcuts or entire dialogs, can be implemented in the same way. They'll be linked to a command which gets executed when a user interacts with the GUI element. As you've probably guessed by now, the elements related to the same operations will be linked to the same commands, preventing any code duplication.

As a result, commands become a convenient middle layer that reduces coupling between the GUI and business logic layers. And that's only a fraction of the benefits that the Command pattern can offer!

Real-World Analogy

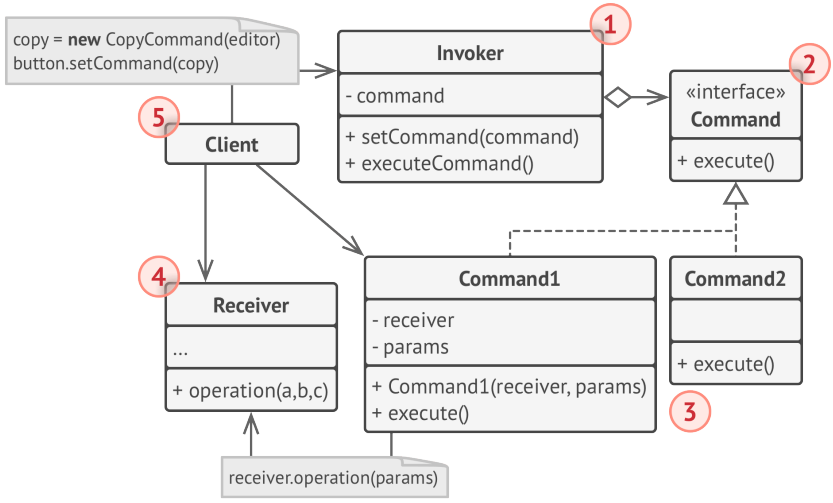


Making an order in a restaurant.

After a long walk through the city, you get to a nice restaurant and sit at the table by the window. A friendly waiter approaches you and quickly takes your order, writing it down on a piece of paper. The waiter goes to the kitchen and sticks the order on the wall. After a while, the order gets to the chef, who reads it and cooks the meal accordingly. The cook places the meal on a tray along with the order. The waiter discovers the tray, checks the order to make sure everything is as you wanted it, and brings everything to your table.

The paper order serves as a command. It remains in a queue until the chef is ready to serve it. The order contains all the relevant information required to cook the meal. It allows the chef to start cooking right away instead of running around clarifying the order details from you directly.

Structure



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. Note that the sender isn't responsible for creating the command object. Usually, it gets a pre-created command from the client via the constructor.

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

3. **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. However, for the sake of simplifying the code, these classes can be merged.

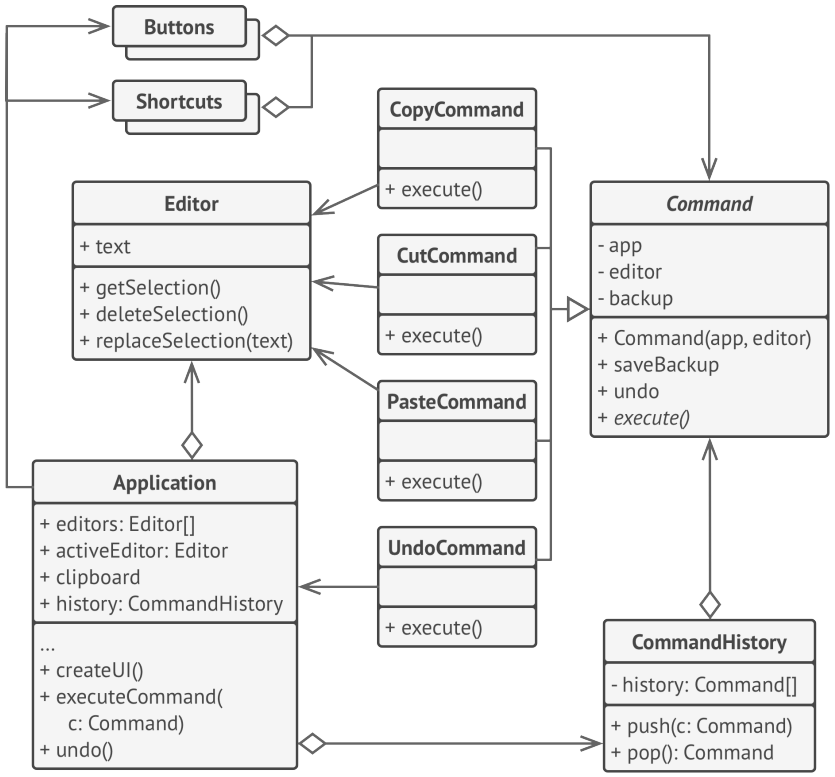
Parameters required to execute a method on a receiving object can be declared as fields in the concrete command. You can make command objects immutable by only allowing the initialization of these fields via the constructor.

4. 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.

5. The **Client** creates and configures concrete command objects. The client must pass all of the request parameters, including a receiver instance, into the command's constructor. After that, the resulting command may be associated with one or multiple senders.

Pseudocode

In this example, the **Command** pattern helps to track the history of executed operations and makes it possible to revert an operation if needed.



Undoable operations in a text editor.

Commands which result in changing the state of the editor (e.g., cutting and pasting) make a backup copy of the editor's state before executing an operation associated with the command. After a command is executed, it's placed into the command history (a stack of command objects) along with the backup copy of the editor's state at that point. Later, if the user needs to revert an operation, the app can take the most recent command from the history, read the associated backup of the editor's state, and restore it.

The client code (GUI elements, command history, etc.) isn't coupled to concrete command classes because it works with commands via the command interface. This approach lets you introduce new commands into the app without breaking any existing code.

```
// The base command class defines the common interface for all
// concrete commands.
abstract class Command is
  protected field app: Application
  protected field editor: Editor
  protected field backup: text

  constructor Command(app: Application, editor: Editor) is
    this.app = app
    this.editor = editor

  // Make a backup of the editor's state.
  method saveBackup() is
    backup = editor.text

  // Restore the editor's state.
  method undo() is
    editor.text = backup

  // The execution method is declared abstract to force all
  // concrete commands to provide their own implementations.
  // The method must return true or false depending on whether
  // the command changes the editor's state.
  abstract method execute()

```

```
// The concrete commands go here.
class CopyCommand extends Command is
  // The copy command isn't saved to the history since it
  // doesn't change the editor's state.
  method execute() is
    app.clipboard = editor.getSelection()
    return false

class CutCommand extends Command is
  // The cut command does change the editor's state, therefore
  // it must be saved to the history. And it'll be saved as
  // long as the method returns true.
  method execute() is
    saveBackup()
    app.clipboard = editor.getSelection()
    editor.deleteSelection()
    return true

class PasteCommand extends Command is
  method execute() is
    saveBackup()
    editor.replaceSelection(app.clipboard)
    return true

// The undo operation is also a command.
class UndoCommand extends Command is
  method execute() is
    app.undo()
    return false


// The global command history is just a stack.
```

```
class CommandHistory is
  private field history: array of Command

  // Last in...
  method push(c: Command) is
    // Push the command to the end of the history array.

  // ...first out
  method pop():Command is
    // Get the most recent command from the history.


// The editor class has actual text editing operations. It plays
// the role of a receiver: all commands end up delegating
// execution to the editor's methods.
class Editor is
  field text: string

  method getSelection() is
    // Return selected text.

  method deleteSelection() is
    // Delete selected text.

  method replaceSelection(text) is
    // Insert the clipboard's contents at the current
    // position.


// The application class sets up object relations. It acts as a
// sender: when something needs to be done, it creates a command
// object and executes it.
```

```
class Application is
  field clipboard: string
  field editors: array of Editors
  field activeEditor: Editor
  field history: CommandHistory

  // The code which assigns commands to UI objects may look
  // like this.
  method createUI() is
    // ...
    copy = function() { executeCommand(
      new CopyCommand(this, activeEditor)) }
    copyButton.setCommand(copy)
    shortcuts.onKeyPress("Ctrl+C", copy)

    cut = function() { executeCommand(
      new CutCommand(this, activeEditor)) }
    cutButton.setCommand(cut)
    shortcuts.onKeyPress("Ctrl+X", cut)

    paste = function() { executeCommand(
      new PasteCommand(this, activeEditor)) }
    pasteButton.setCommand(paste)
    shortcuts.onKeyPress("Ctrl+V", paste)

    undo = function() { executeCommand(
      new UndoCommand(this, activeEditor)) }
    undoButton.setCommand(undo)
    shortcuts.onKeyPress("Ctrl+Z", undo)

  // Execute a command and check whether it has to be added to
  // the history.
```

```
  method executeCommand(command) is
    if (command.execute)
      history.push(command)

  // Take the most recent command from the history and run its
  // undo method. Note that we don't know the class of that
  // command. But we don't have to, since the command knows
  // how to undo its own action.
  method undo() is
    command = history.pop()
    if (command != null)
      command.undo()
```

Applicability

Use the Command pattern when you want to parametrize objects with operations.

The Command pattern can turn a specific method call into a stand-alone object. This change opens up a lot of interesting uses: you can pass commands as method arguments, store them inside other objects, switch linked commands at runtime, etc.

Here's an example: you're developing a GUI component such as a context menu, and you want your users to be able to configure menu items that trigger operations when an end user clicks an item.

Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely.

As with any other object, a command can be serialized, which means converting it to a string that can be easily written to a file or a database. Later, the string can be restored as the initial command object. Thus, you can delay and schedule command execution. But there's even more! In the same way, you can queue, log or send commands over the network.

Use the Command pattern when you want to implement reversible operations.

Although there are many ways to implement undo/redo, the Command pattern is perhaps the most popular of all.

To be able to revert operations, you need to implement the history of performed operations. The command history is a stack that contains all executed command objects along with related backups of the application's state.

This method has two drawbacks. First, it isn't that easy to save an application's state because some of it can be private. This problem can be mitigated with the **Memento** pattern.

Second, the state backups may consume quite a lot of RAM. Therefore, sometimes you can resort to an alternative implementation: instead of restoring the past state, the command performs the inverse operation. The reverse operation also

has a price: it may turn out to be hard or even impossible to implement.

How to Implement

1. Declare the command interface with a single execution method.

2. Start extracting requests into concrete command classes that implement the command interface. Each class must have a set of fields for storing the request arguments along with a reference to the actual receiver object. All these values must be initialized via the command's constructor.

3. Identify classes that will act as *senders*. Add the fields for storing commands into these classes. Senders should communicate with their commands only via the command interface. Senders usually don't create command objects on their own, but rather get them from the client code.

4. Change the senders so they execute the command instead of sending a request to the receiver directly.

5. The client should initialize objects in the following order:

    - Create receivers.
    - Create commands, and associate them with receivers if needed.

- Create senders, and associate them with specific commands.

Pros and 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.

✕ The code may become more complicated since you're introducing a whole new layer between senders and receivers.

Relations with Other Patterns

- **Chain of Responsibility**, **Command**, **Mediator** and **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.

- 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.

  However, there's another approach, where the request itself is a *Command* object. In this case, you can execute the same operation in a series of different contexts linked into a chain.

- 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.

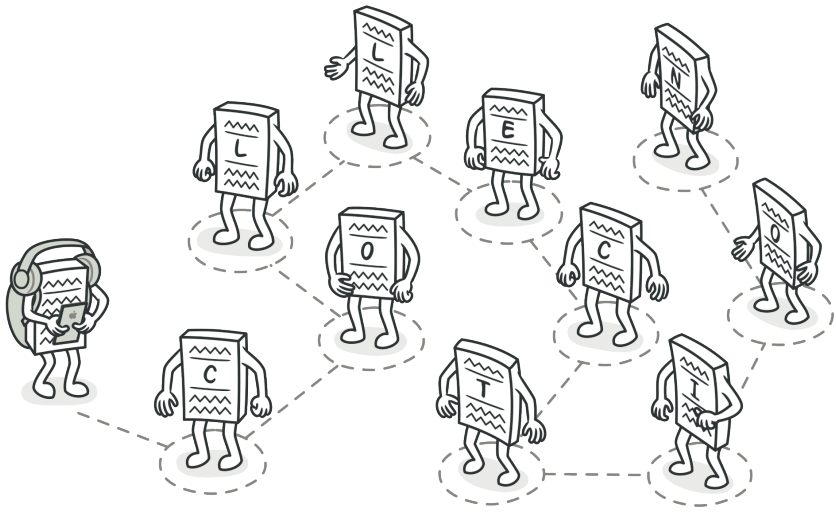
  - You can use *Command* to convert any operation into an object. The operation's parameters become fields of that object. The conversion lets you defer execution of the oper-

ation, 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.

- **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.

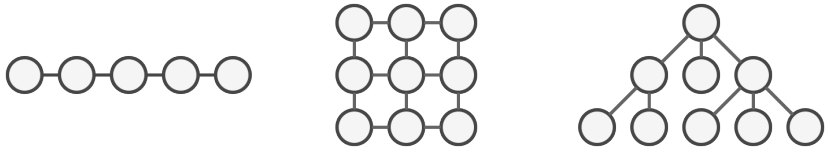


ITERATOR

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

☹ Problem

Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.

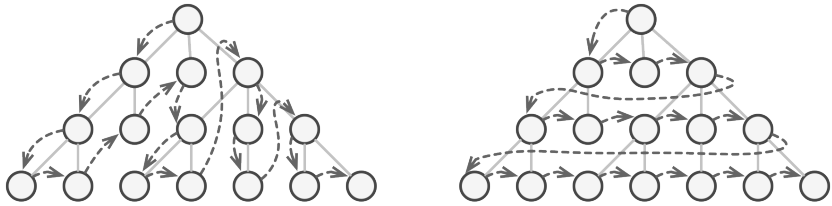


Various types of collections.

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

But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

This may sound like an easy job if you have a collection based on a list. You just loop over all of the elements. But how do you sequentially traverse elements of a complex data structure, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.



The same collection can be traversed in several different ways.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

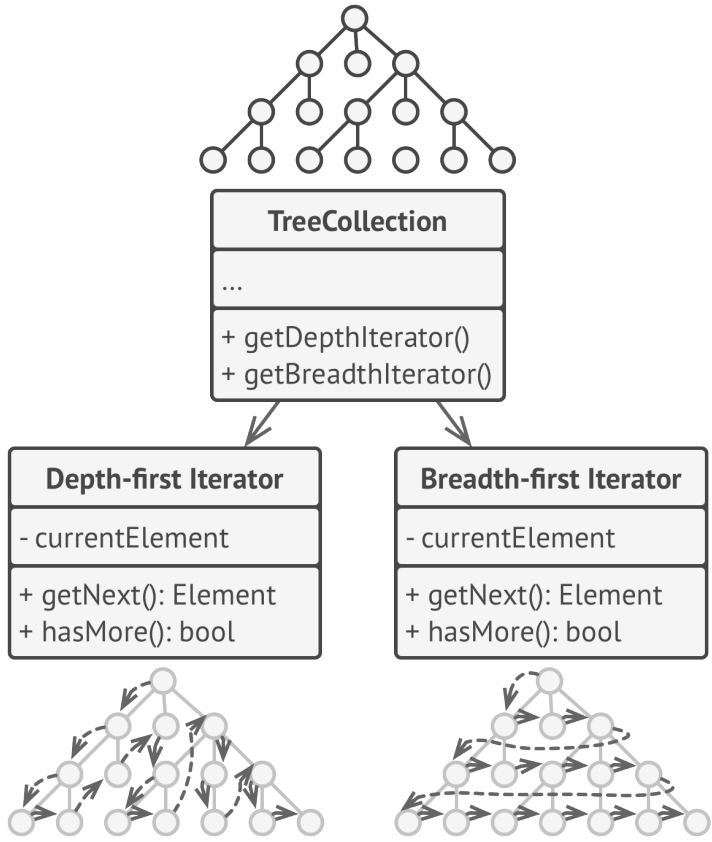
On the other hand, the client code that's supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, you have no option other than to couple your code to the specific collection classes.

😊 Solution

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

In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end.

Because of this, several iterators can go through the same collection at the same time, independently of each other.

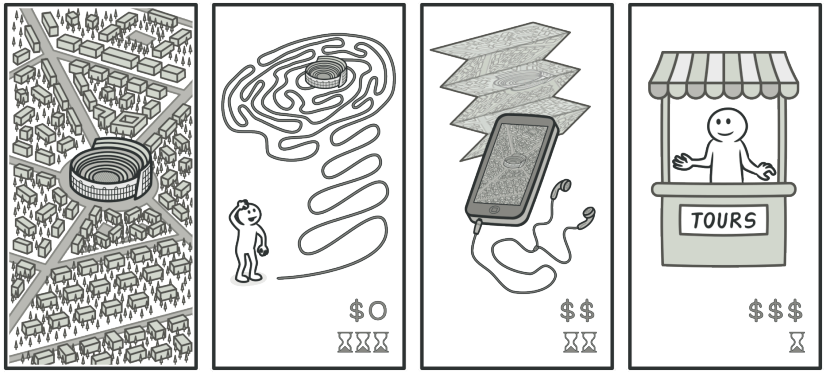


Iterators implement various traversal algorithms. Several iterator objects can traverse the same collection at the same time.

Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.

All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.

Real-World Analogy



Various ways to walk around Rome.

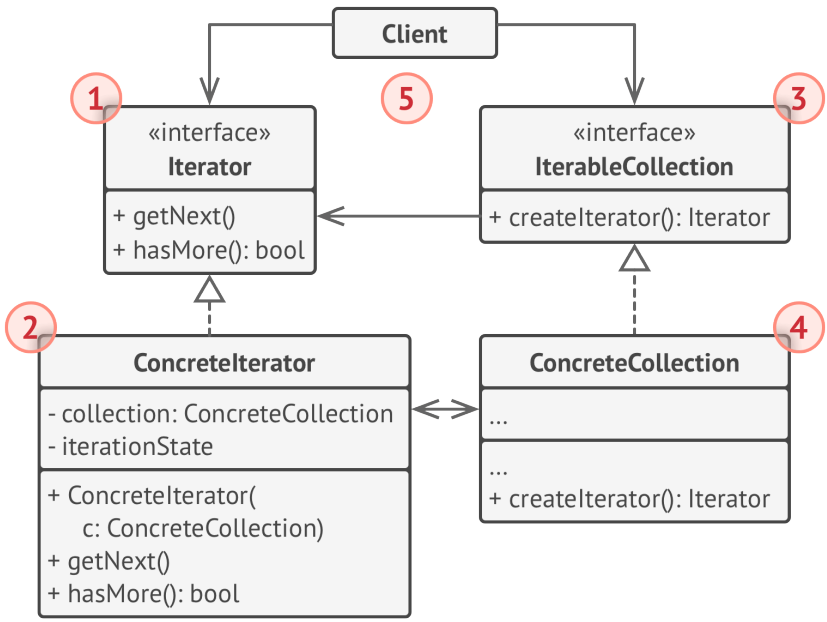
You plan to visit Rome for a few days and visit all of its main sights and attractions. But once there, you could waste a lot of time walking in circles, unable to find even the Colosseum.

On the other hand, you could buy a virtual guide app for your smartphone and use it for navigation. It's smart and inexpensive, and you could be staying at some interesting places for as long as you want. Another alternative is that you could spend some of the trip's budget and hire a local guide who knows the city like the back of his hand. The guide would be able to tailor

the tour to your likings, show you every attraction and tell a lot of exciting stories. That'll be even more fun; but, alas, more expensive, too.

All of these options—the random directions born in your head, the smartphone navigator or the human guide—act as iterators over the vast collection of sights and attractions located in Rome.

Structure



1. The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.

2. **Concrete Iterators** implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other.

3. The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.

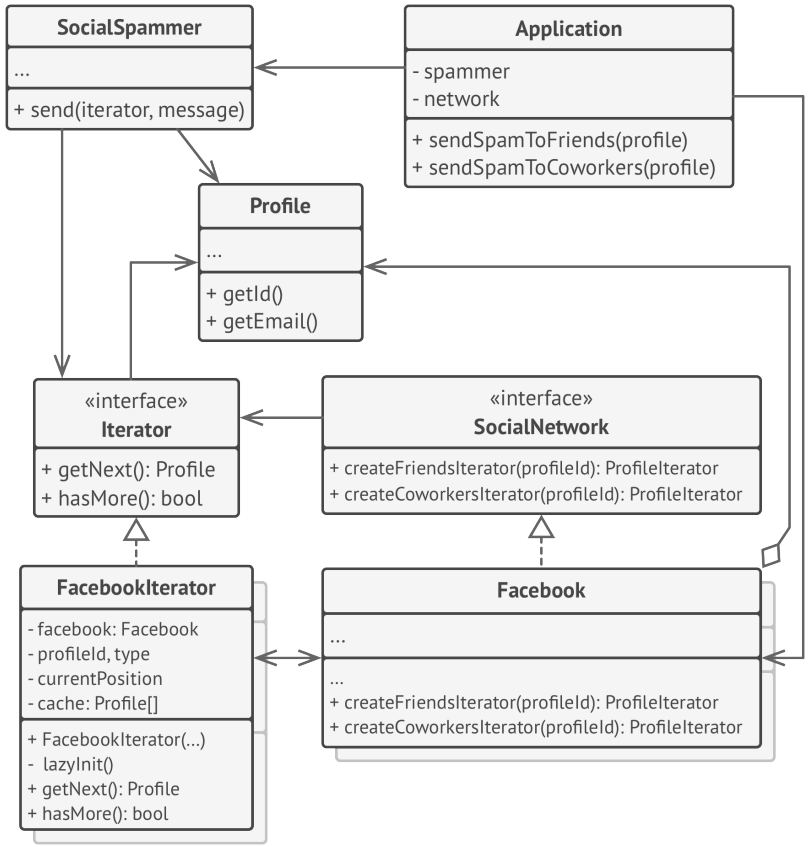
4. **Concrete Collections** return new instances of a particular concrete iterator class each time the client requests one. You might be wondering, where's the rest of the collection's code? Don't worry, it should be in the same class. It's just that these details aren't crucial to the actual pattern, so we're omitting them.

5. The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code.

   Typically, clients don't create iterators on their own, but instead get them from collections. Yet, in certain cases, the client can create one directly; for example, when the client defines its own special iterator.

Pseudocode

In this example, the **Iterator** pattern is used to walk through a special kind of collection which encapsulates access to Facebook's social graph. The collection provides several iterators that can traverse profiles in various ways.



Example of iterating over social profiles.

The 'friends' iterator can be used to go over the friends of a given profile. The 'colleague' iterator does the same, except it

omits friends who don't work at the same company as a target person. Both iterators implement a common interface which allows clients to fetch profiles without diving into implementation details such as authentication and sending REST requests.

The client code isn't coupled to concrete classes because it works with collections and iterators only through interfaces. If you decide to connect your app to a new social network, you simply need to provide new collection and iterator classes without changing the existing code.

```
// The collection interface must declare a factory method for
// producing iterators. You can declare several methods if there
// are different kinds of iteration available in your program.
interface SocialNetwork is
  method createFriendsIterator(profileId):ProfileIterator
  method createCoworkersIterator(profileId):ProfileIterator

// Each concrete collection is coupled to a set of concrete
// iterator classes it returns. But the client isn't, since the
// signature of these methods returns iterator interfaces.
class Facebook implements SocialNetwork is
  // ... The bulk of the collection's code should go here ...

  // Iterator creation code.
  method createFriendsIterator(profileId) is
    return new FacebookIterator(this, profileId, "friends")
  method createCoworkersIterator(profileId) is
    return new FacebookIterator(this, profileId, "coworkers")
```

```
// The common interface for all iterators.
interface ProfileIterator is
  method getNext():Profile
  method hasMore():bool


// The concrete iterator class.
class FacebookIterator implements ProfileIterator is
  // The iterator needs a reference to the collection that it
  // traverses.
  private field facebook: Facebook
  private field profileId, type: string

  // An iterator object traverses the collection independently
  // from other iterators. Therefore it has to store the
  // iteration state.
  private field currentPosition
  private field cache: array of Profile

  constructor FacebookIterator(facebook, profileId, type) is
    this.facebook = facebook
    this.profileId = profileId
    this.type = type

  private method lazyInit() is
    if (cache == null)
      cache = facebook.socialGraphRequest(profileId, type)

  // Each concrete iterator class has its own implementation
  // of the common iterator interface.
  method getNext() is
    if (hasMore())
```

```
        currentPosition++
        return cache[currentPosition]

    method hasMore() is
      lazyInit()
      return cache.length < currentPosition


// Here is another useful trick: you can pass an iterator to a
// client class instead of giving it access to a whole
// collection. This way, you don't expose the collection to the
// client.
//
// And there's another benefit: you can change the way the
// client works with the collection at runtime by passing it a
// different iterator. This is possible because the client code
// isn't coupled to concrete iterator classes.
class SocialSpammer is
  method send(iterator: ProfileIterator, message: string) is
    while (iterator.hasNext())
      profile = iterator.getNext()
      System.sendEmail(profile.getEmail(), message)

// The application class configures collections and iterators
// and then passes them to the client code.
class Application is
  field network: SocialNetwork
  field spammer: SocialSpammer

  method config() is
    if working with Facebook
      this.network = new Facebook()
```

```
    if working with LinkedIn
      this.network = new LinkedIn()
    this.spammer = new SocialSpammer()

  method sendSpamToFriends(profile) is
    iterator = network.createFriendsIterator(profile.getId())
    spammer.send(iterator, "Very important message")

  method sendSpamToCoworkers(profile) is
    iterator = network.createCoworkersIterator(profile.getId())
    spammer.send(iterator, "Very important message")
```

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).

The iterator encapsulates the details of working with a complex data structure, providing the client with several simple methods of accessing the collection elements. While this approach is very convenient for the client, it also protects the collection from careless or malicious actions which the client would be able to perform if working with the collection directly.

Use the pattern to reduce duplication of the traversal code across your app.

The code of non-trivial iteration algorithms tends to be very bulky. When placed within the business logic of an app, it may blur the responsibility of the original code and make it less maintainable. Moving the traversal code to designated iterators can help you make the code of the application more lean and clean.

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.

The pattern provides a couple of generic interfaces for both collections and iterators. Given that your code now uses these interfaces, it'll still work if you pass it various kinds of collections and iterators that implement these interfaces.

How to Implement

1. Declare the iterator interface. At the very least, it must have a method for fetching the next element from a collection. But for the sake of convenience you can add a couple of other methods, such as fetching the previous element, tracking the current position, and checking the end of the iteration.

2. Declare the collection interface and describe a method for fetching iterators. The return type should be equal to that of the iterator interface. You may declare similar methods if you plan to have several distinct groups of iterators.

3. Implement concrete iterator classes for the collections that you want to be traversable with iterators. An iterator object must be linked with a single collection instance. Usually, this link is established via the iterator's constructor.

4. Implement the collection interface in your collection classes. The main idea is to provide the client with a shortcut for creating iterators, tailored for a particular collection class. The collection object must pass itself to the iterator's constructor to establish a link between them.

5. Go over the client code to replace all of the collection traversal code with the use of iterators. The client fetches a new iterator object each time it needs to iterate over the collection elements.

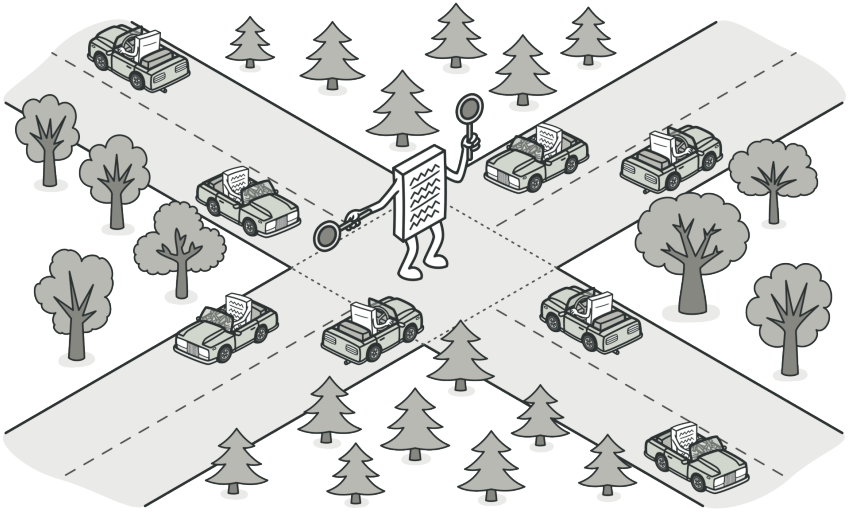
Pros and Cons

- ✓ *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.

- ✕ 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.

⇄ Relations 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.



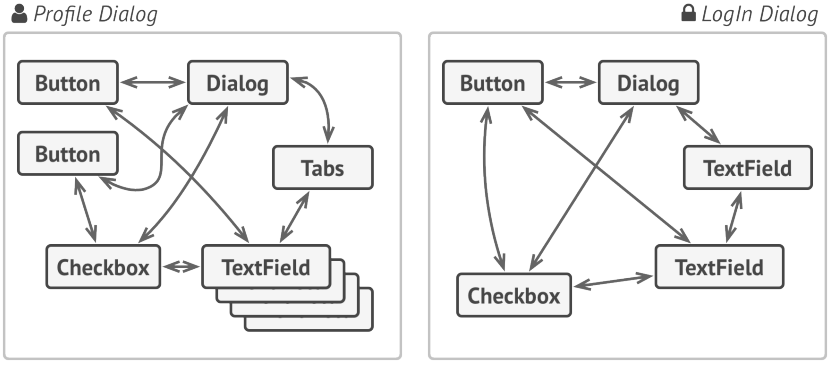
MEDIATOR

Also known as: Intermediary, Controller

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.



Relations between elements of the user interface can become chaotic as the application evolves.

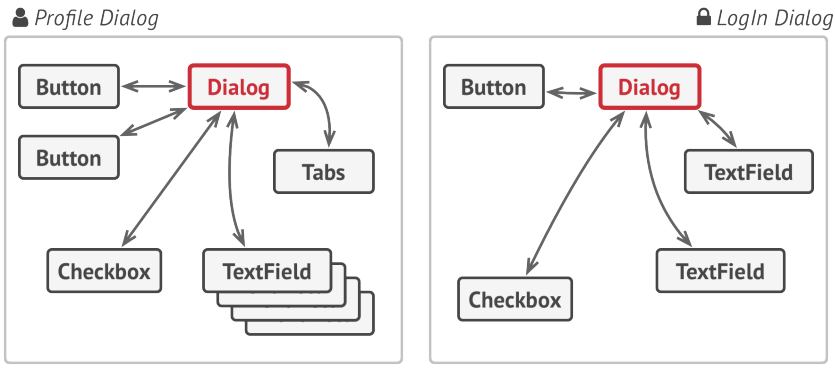
Some of the form elements may interact with others. For instance, selecting the “I have a dog” checkbox may reveal a hidden text field for entering the dog’s name. Another example is the submit button that has to validate values of all fields before saving the data.

Elements can have lots of relations with other elements. Hence, changes to some elements may affect the 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. For example, you won't be able to use that checkbox class inside another form, because it's coupled to the dog's text field. You can use either all the classes involved in rendering the profile form, or none at all.

😊 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.



UI elements should communicate indirectly, via the mediator object.

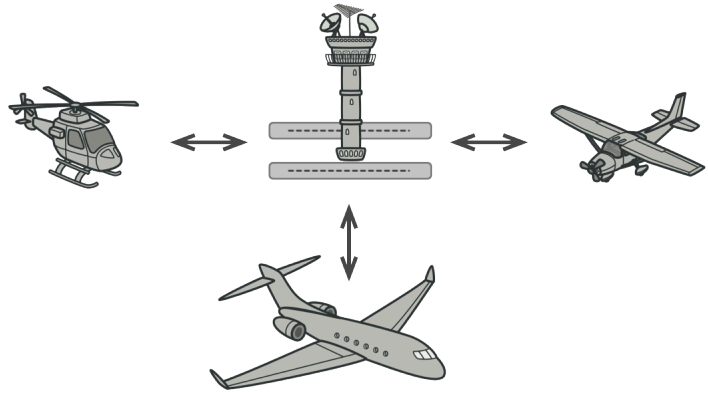
In our example with the profile editing form, the dialog class itself may act as the mediator. Most likely, the dialog class is already aware of all of its sub-elements, so you won't even need to introduce new dependencies into this class.

The most significant change happens to the actual form elements. Let's consider the submit button. Previously, each time a user clicked the button, it had to validate the values of all individual form elements. Now its single job is to notify the dialog about the click. Upon receiving this notification, the dialog itself performs the validations or passes the task to the individual elements. Thus, instead of being tied to a dozen form elements, the button is only dependent on the dialog class.

You can go further and make the dependency even looser by extracting the common interface for all types of dialogs. The interface would declare the notification method which all form elements can use to notify the dialog about events happening to those elements. Thus, our submit button should now be able to work with any dialog that implements that interface.

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.

Real-World Analogy

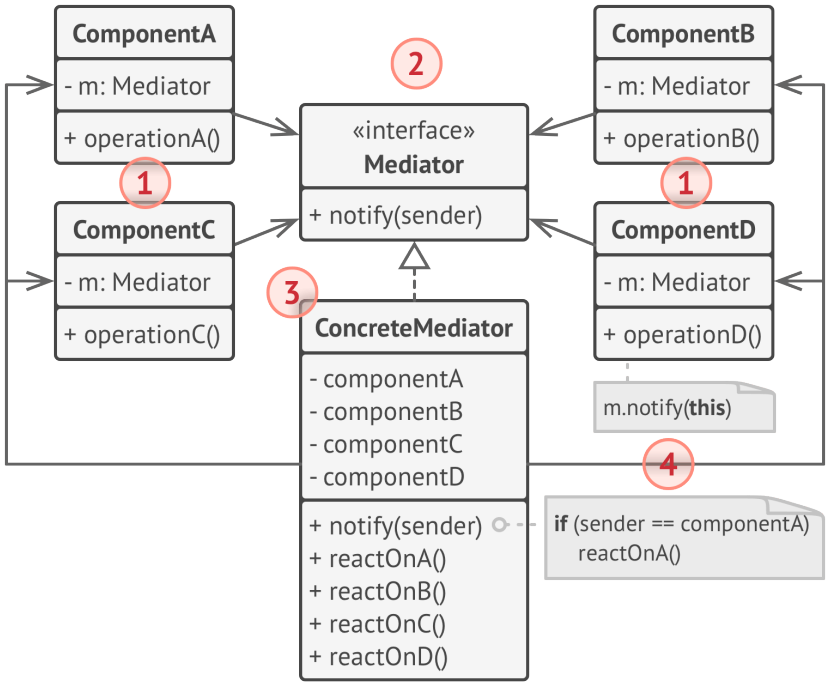


Aircraft pilots don't talk to each other directly when deciding who gets to land their plane next. All communication goes through the control tower.

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. Without the air traffic controller, pilots would need to be aware of every plane in the vicinity of the airport, discussing landing priorities with a committee of dozens of other pilots. That would probably skyrocket the airplane crash statistics.

The tower doesn't need to control the whole flight. It exists only to enforce constraints in the terminal area because the number of involved actors there might be overwhelming to a pilot.

Structure



1. **Components** are various classes that contain some business logic. Each component has a reference to a mediator, declared with the type of the mediator interface. The component isn't aware of the actual class of the mediator, so you can reuse the component in other programs by linking it to a different mediator.

2. The **Mediator** interface declares methods of communication with components, which usually include just a single notification method. Components may pass any context as arguments of this method, including their own objects, but only in such

a way that no coupling occurs between a receiving component and the sender's class.

3. **Concrete Mediators** encapsulate relations between various components. Concrete mediators often keep references to all components they manage and sometimes even manage their lifecycle.

4. Components must not be aware of other components. If something important happens within or to a component, it must only notify the mediator. When the mediator receives the notification, it can easily identify the sender, which might be just enough to decide what component should be triggered in return.

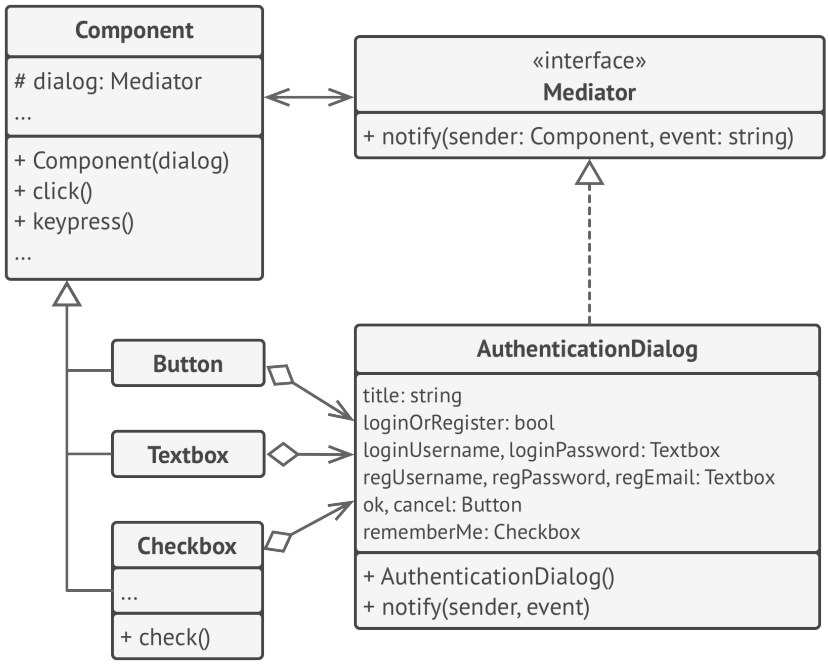
   From a component's perspective, it all looks like a total black box. The sender doesn't know who'll end up handling its request, and the receiver doesn't know who sent the request in the first place.

Pseudocode

In this example, the **Mediator** pattern helps you eliminate mutual dependencies between various UI classes: buttons, checkboxes and text labels.

An element, triggered by a user, doesn't communicate with other elements directly, even if it looks like it's supposed to. Instead, the element only needs to let its mediator know about

the event, passing any contextual info along with that notification.



Structure of the UI dialog classes.

In this example, the whole authentication dialog acts as the mediator. It knows how concrete elements are supposed to collaborate and facilitates their indirect communication. Upon receiving a notification about an event, the dialog decides what element should address the event and redirects the call accordingly.

```
// The mediator interface declares a method used by components
// to notify the mediator about various events. The mediator may
// react to these events and pass the execution to other
// components.
interface Mediator is
  method notify(sender: Component, event: string)


// The concrete mediator class. The intertwined web of
// connections between individual components has been untangled
// and moved into the mediator.
class AuthenticationDialog implements Mediator is
  private field title: string
  private field loginOrRegisterChkBx: Checkbox
  private field loginUsername, loginPassword: Textbox
  private field registrationUsername, registrationPassword
  private field registrationEmail: Textbox
  private field okBtn, cancelBtn: Button

  constructor AuthenticationDialog() is
    // Create all component objects and pass the current
    // mediator into their constructors to establish links.

  // When something happens with a component, it notifies the
  // mediator. Upon receiving a notification, the mediator may
  // do something on its own or pass the request to another
  // component.
  method notify(sender, event) is
    if (sender == loginOrRegisterChkBx and event == "check")
      if (loginOrRegisterChkBx.checked)
        title = "Log in"
        // 1. Show login form components.
```

```
        // 2. Hide registration form components.
      else
        title = "Register"
        // 1. Show registration form components.
        // 2. Hide login form components

    if (sender == okBtn && event == "click")
      if (loginOrRegister.checked)
        // Try to find a user using login credentials.
        if (!found)
          // Show an error message above the login
          // field.
      else
        // 1. Create a user account using data from the
        // registration fields.
        // 2. Log that user in.
        // ...


// Components communicate with a mediator using the mediator
// interface. Thanks to that, you can use the same components in
// other contexts by linking them with different mediator
// objects.
class Component is
  field dialog: Mediator

  constructor Component(dialog) is
    this.dialog = dialog

  method click() is
    dialog.notify(this, "click")

```

```
    method keypress() is
      dialog.notify(this, "keypress")

// Concrete components don't talk to each other. They have only
// one communication channel, which is sending notifications to
// the mediator.
class Button extends Component is
  // ...

class Textbox extends Component is
  // ...

class Checkbox extends Component is
  method check() is
    dialog.notify(this, "check")
  // ...
```

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.

The pattern lets you extract all the relationships between classes into a separate class, isolating any changes to a specific component from the rest of the components.

Use the pattern when you can't reuse a component in a different program because it's too dependent on other components.

After you apply the Mediator, individual components become unaware of the other components. They could still communicate with each other, albeit indirectly, through a mediator object. To reuse a component in a different app, you need to provide it with a new mediator class.

Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.

Since all relations between components are contained within the mediator, it's easy to define entirely new ways for these components to collaborate by introducing new mediator classes, without having to change the components themselves.

How to Implement

1. Identify a group of tightly coupled classes which would benefit from being more independent (e.g., for easier maintenance or simpler reuse of these classes).

2. Declare the mediator interface and describe the desired communication protocol between mediators and various components. In most cases, a single method for receiving notifications from components is sufficient.

   This interface is crucial when you want to reuse component classes in different contexts. As long as the component works

with its mediator via the generic interface, you can link the component with a different implementation of the mediator.

3. Implement the concrete mediator class. This class would benefit from storing references to all of the components it manages.

4. You can go even further and make the mediator responsible for the creation and destruction of component objects. After this, the mediator may resemble a **factory** or a **facade**.

5. Components should store a reference to the mediator object. The connection is usually established in the component's constructor, where a mediator object is passed as an argument.

6. Change the components' code so that they call the mediator's notification method instead of methods on other components. Extract the code that involves calling other components into the mediator class. Execute this code whenever the mediator receives notifications from that component.

⚖ Pros and Cons

✓ *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.

✕ Over time a mediator can evolve into a **God Object**.

⇄ Relations with Other Patterns

- **Chain of Responsibility**, **Command**, **Mediator** and **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.
- **Facade** and **Mediator** have similar jobs: they try to organize collaboration between lots of tightly coupled classes.
    - *Facade* defines a simplified interface to a subsystem of objects, but it doesn't introduce any new functionality. The subsystem itself is unaware of the facade. Objects within the subsystem can communicate directly.

  - *Mediator* centralizes communication between components of the system. The components only know about the mediator object and don't communicate directly.

- 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.

  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.

MEMENTO

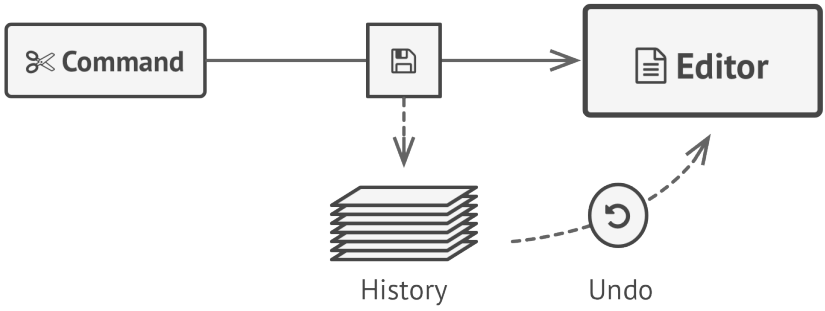
Also known as: Snapshot

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. In addition to simple text editing, your editor can format text, insert inline images, etc.

At some point, you decided to let users undo any operations carried out on the text. This feature has become so common over the years that nowadays people expect every app to have it. For the implementation, you chose to take the direct approach. Before performing any operation, the app records the state of all objects and saves it in some storage. Later, when a user decides to revert an action, the app fetches the latest snapshot from the history and uses it to restore the state of all objects.

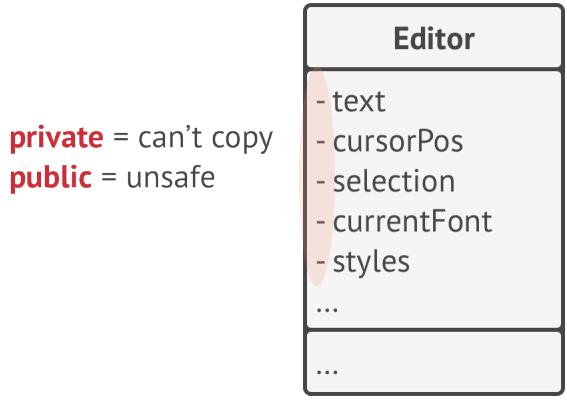


Before executing an operation, the app saves a snapshot of the objects' state, which can later be used to restore objects to their previous state.

Let's think about those state snapshots. How exactly would you produce one? You'd probably need to go over all the fields in an object and copy their values into storage. 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 significant data in private fields.

Ignore that problem for now and let's assume that our objects behave like hippies: preferring open relations and keeping their state public. While this approach would solve the immediate problem and let you produce snapshots of objects' states at will, it still has some serious issues. In the future, you might decide to refactor some of the editor classes, or add or remove some of the fields. Sounds easy, but this would also require chaining the classes responsible for copying the state of the affected objects.



How to make a copy of the object's private state?

But there's more. Let's consider the actual "snapshots" of the editor's state. What data does it contain? At a bare minimum, it must contain the actual text, cursor coordinates, current scroll

position, etc. To make a snapshot, you'd need to collect these values and put them into some kind of container.

Most likely, you're going to store lots of these container objects inside some list that would represent the history. Therefore the containers would probably end up being objects of one class. The class would have almost no methods, but lots of fields that mirror the editor's state. To allow other objects to write and read data to and from a snapshot, you'd probably need to make its fields public. That would expose all the editor's states, private or not. Other classes would become dependent on every little change to the snapshot class, which would otherwise happen within private fields and methods without affecting outer classes.

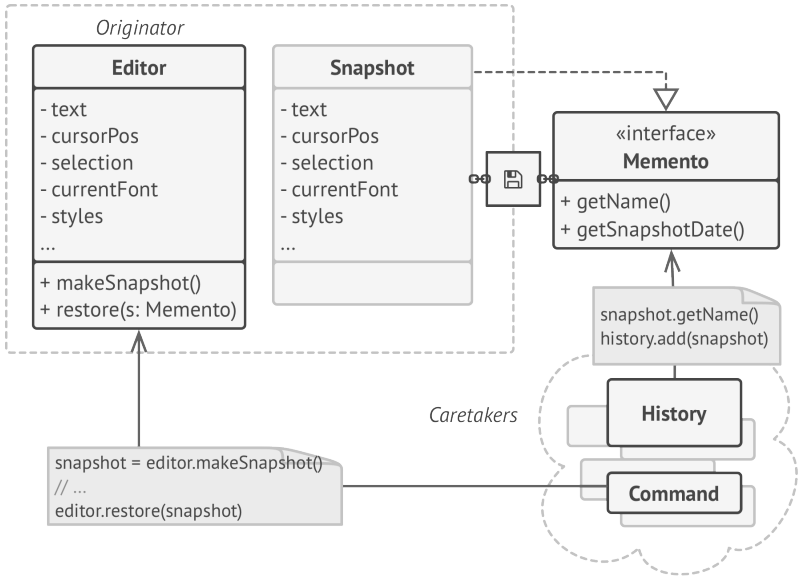
It looks like we've reached a dead end: you either expose all internal details of classes, making them too fragile, or restrict access to their state, making it impossible to produce snapshots. Is there any other way to implement the "undo"?

😊 Solution

All problems that we've just experienced are caused by broken encapsulation. Some objects try to do more than they are supposed to. To collect the data required to perform some action, they invade the private space of other objects instead of letting these objects perform the actual action.

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 itself can make the snapshot since it has full access to its own state.

The pattern suggests storing the copy of the object's state in a special object called *memento*. The contents of the memento aren't accessible to any other object except the one that produced it. Other objects must communicate with mementos using a limited interface which may allow fetching the snapshot's metadata (creation time, the name of the performed operation, etc.), but not the original object's state contained in the snapshot.



The originator has full access to the memento, whereas the caretaker can only access the metadata.

Such a restrictive policy lets you store mementos inside other objects, usually called *caretakers*. Since the caretaker works with the memento only via the limited interface, it's not able to tamper with the state stored inside the memento. At the same time, the originator has access to all fields inside the memento, allowing it to restore its previous state at will.

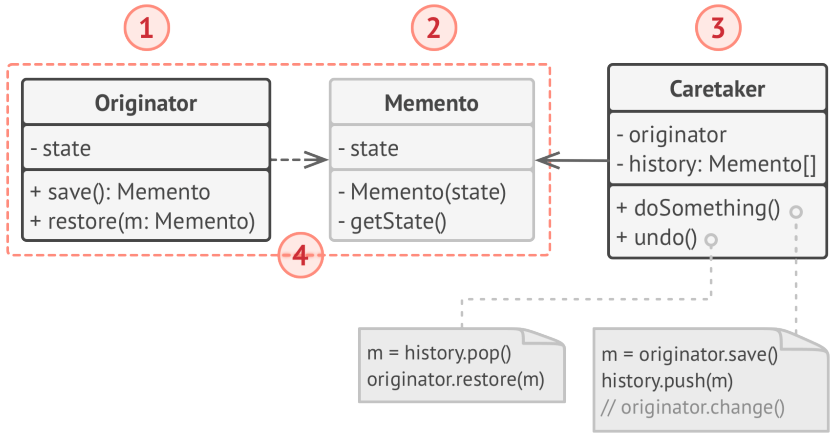
In our text editor example, we can create a separate history class to act as the caretaker. A stack of mementos stored inside the caretaker will grow each time the editor is about to execute an operation. You could even render this stack within the app's UI, displaying the history of previously performed operations to a user.

When a user triggers the undo, the history grabs the most recent memento from the stack and passes it back to the editor, requesting a roll-back. Since the editor has full access to the memento, it changes its own state with the values taken from the memento.

Structure

Implementation based on nested classes

The classic implementation of the pattern relies on support for nested classes, available in many popular programming languages (such as C++, C#, and Java).



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

2. 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.

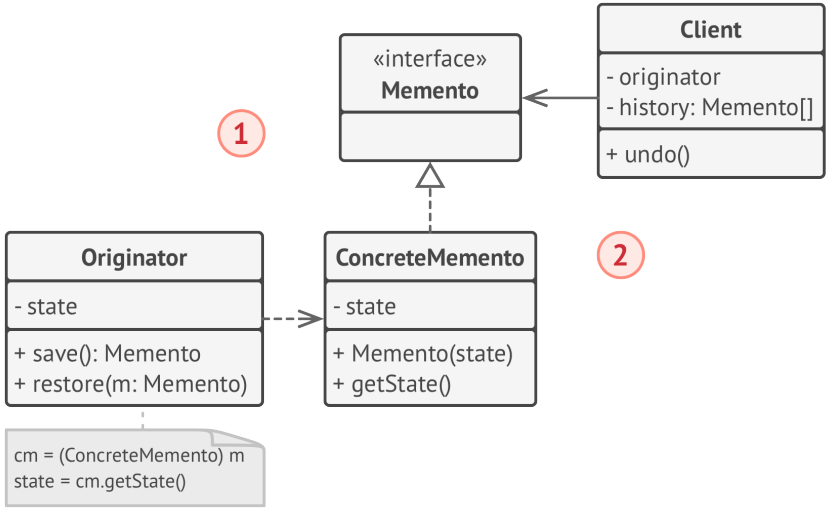
3. The **Caretaker** knows not only "when" and "why" to capture the originator's state, but also when the state should be restored.

A caretaker can keep track of the originator's history by storing a stack of mementos. When the originator has to travel back in history, the caretaker fetches the topmost memento from the stack and passes it to the originator's restoration method.

4. 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.

Implementation based on an intermediate interface

There's an alternative implementation, suitable for programming languages that don't support nested classes (yeah, PHP, I'm talking about you).

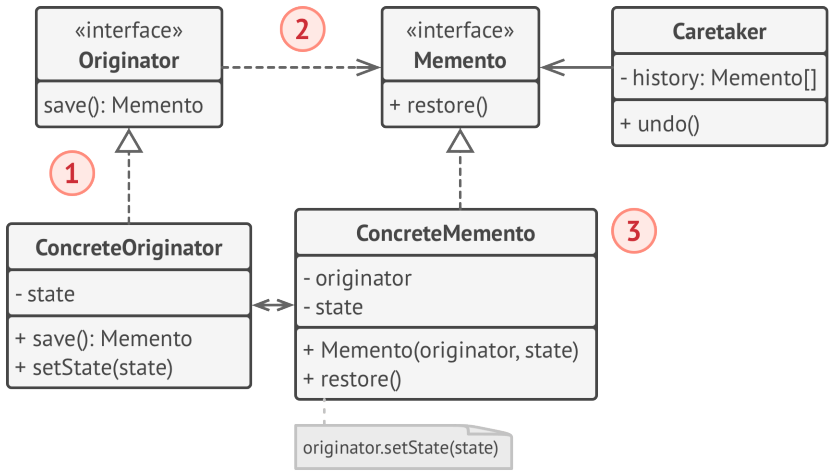


1. In the absence of nested classes, you can restrict access to the memento's fields by establishing a convention that caretakers can work with a memento only through an explicitly declared intermediary interface, which would only declare methods related to the memento's metadata.

2. On the other hand, originators can work with a memento object directly, accessing fields and methods declared in the memento class. The downside of this approach is that you need to declare all members of the memento public.

Implementation with even stricter encapsulation

There's another implementation which is useful when you don't want to leave even the slightest chance of other classes accessing the state of the originator through the memento.



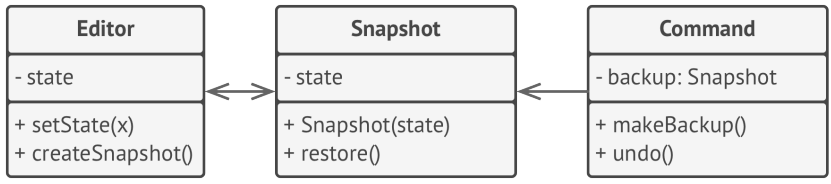
1. This implementation allows having multiple types of originators and mementos. Each originator works with a corresponding memento class. Neither originators nor mementos expose their state to anyone.

2. Caretakers are now explicitly restricted from changing the state stored in mementos. Moreover, the caretaker class becomes independent from the originator because the restoration method is now defined in the memento class.

3. Each memento becomes linked to the originator that produced it. The originator passes itself to the memento's constructor, along with the values of its state. Thanks to the close relationship between these classes, a memento can restore the state of its originator, given that the latter has defined the appropriate setters.

Pseudocode

In this example uses the Memento pattern alongside the **Command** pattern for storing snapshots of the complex text editor's state and restoring an earlier state from these snapshots when needed.



Saving snapshots of the text editor's state.

The command objects act as caretakers. They fetch the editor's memento before executing operations related to commands. When a user attempts to undo the most recent command, the editor can use the memento stored in that command to revert itself to the previous state.

The memento class doesn't declare any public fields, getters or setters. Therefore no object can alter its contents. Mementos are linked to the editor object that created them. This lets a memento restore the linked editor's state by passing the data via setters on the editor object. Since mementos are linked to specific editor objects, you can make your app support several independent editor windows with a centralized undo stack.

```
// The originator holds some important data that may change over
// time. It also defines a method for saving its state inside a
// memento and another method for restoring the state from it.
class Editor is
  private field text, curX, curY, selectionWidth

  method setText(text) is
    this.text = text

  method setCursor(x, y) is
    this.curX = curX
    this.curY = curY

  method setSelectionWidth(width) is
    this.selectionWidth = width

```

```
  // Saves the current state inside a memento.
  method createSnapshot():Snapshot is
    // Memento is an immutable object; that's why the
    // originator passes its state to the memento's
    // constructor parameters.
    return new Snapshot(this, text, curX, curY, selectionWidth)

// The memento class stores the past state of the editor.
class Snapshot is
  private field editor: Editor
  private field text, curX, curY, selectionWidth

  constructor Snapshot(editor, text, curX, curY, selectionWidth) is
    this.editor = editor
    this.text = text
    this.curX = curX
    this.curY = curY
    this.selectionWidth = selectionWidth

  // At some point, a previous state of the editor can be
  // restored using a memento object.
  method restore() is
    editor.setText(text)
    editor.setCursor(curX, curY)
    editor.setSelectionWidth(selectionWidth)

// A command object can act as a caretaker. In that case, the
// command gets a memento just before it changes the
// originator's state. When undo is requested, it restores the
// originator's state from a memento.
class Command is
  private field backup: Snapshot
```

```
49    method makeBackup() is
50      backup = editor.createSnapshot()
51
52    method undo() is
53      if (backup != null)
54        backup.restore()
55    // ...
```

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.

The Memento pattern lets you make full copies of an object's state, including private fields, and store them separately from the object. While most people remember this pattern thanks to the "undo" use case, it's also indispensable when dealing with transactions (i.e., if you need to roll back an operation on error).

Use the pattern when direct access to the object's fields/getters/setters violates its encapsulation.

The Memento makes the object itself responsible for creating a snapshot of its state. No other object can read the snapshot, making the original object's state data safe and secure.

How to Implement

1. Determine what class will play the role of the originator. It's important to know whether the program uses one central object of this type or multiple smaller ones.

2. Create the memento class. One by one, declare a set of fields that mirror the fields declared inside the originator class.

3. Make the memento class immutable. A memento should accept the data just once, via the constructor. The class should have no setters.

4. If your programming language supports nested classes, nest the memento inside the originator. If not, extract a blank interface from the memento class and make all other objects use it to refer to the memento. You may add some metadata operations to the interface, but nothing that exposes the originator's state.

5. Add a method for producing mementos to the originator class. The originator should pass its state to the memento via one or multiple arguments of the memento's constructor.

   The return type of the method should be of the interface you extracted in the previous step (assuming that you extracted it at all). Under the hood, the memento-producing method should work directly with the memento class.

6. Add a method for restoring the originator's state to its class. It should accept a memento object as an argument. If you extracted an interface in the previous step, make it the type of the parameter. In this case, you need to typecast the incoming object to the mediator class, since the originator needs full access to that object.

7. The caretaker, whether it represents a command object, a history, or something entirely different, should know when to request new mementos from the originator, how to store them and when to restore the originator with a particular memento.

8. The link between caretakers and originators may be moved into the memento class. In this case, each memento must be connected to the originator that had created it. The restoration method would also move to the memento class. However, this would all make sense only if the memento class is nested into originator or the originator class provides sufficient setters for overriding its state.

⚖ Pros and Cons

✓ 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.

✕ The app might consume lots of RAM if clients create mementos too often.

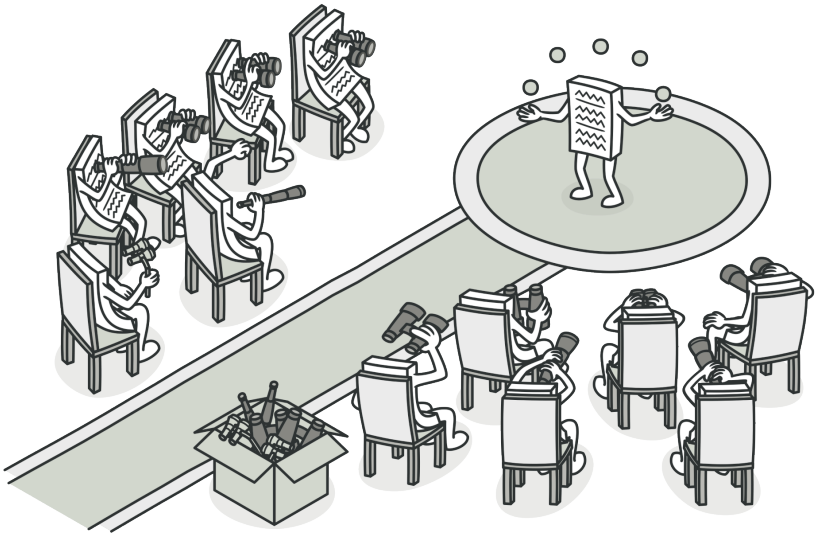
- ✕ Caretakers should track the originator's lifecycle to be able to destroy obsolete mementos.
- ✕ Most dynamic programming languages, such as PHP, Python and JavaScript, can't guarantee that the state within the memento stays untouched.

⇄ Relations 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, while 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

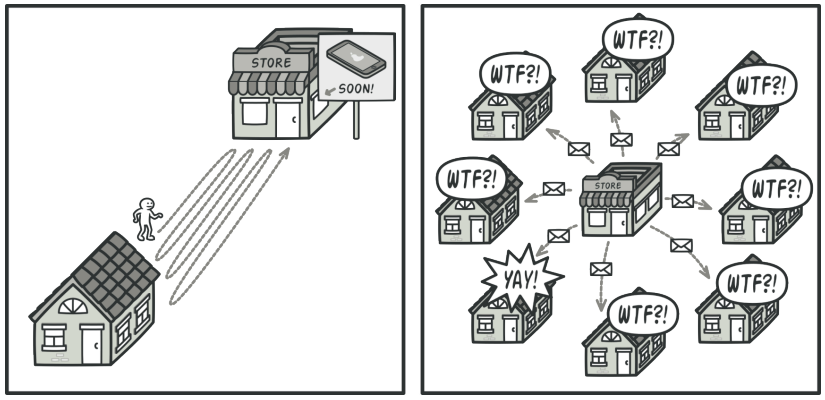
Also known as: Event-Subscriber, Listener

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

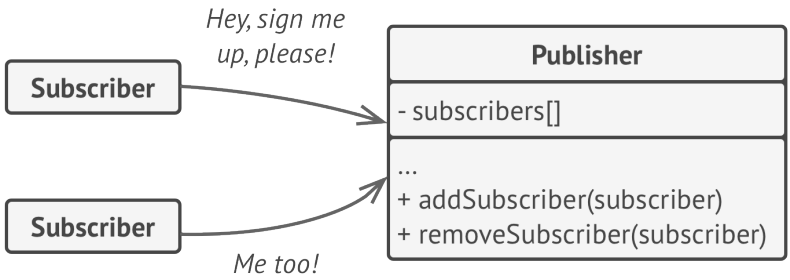
On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it'd upset other customers who aren't interested in new products.

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.

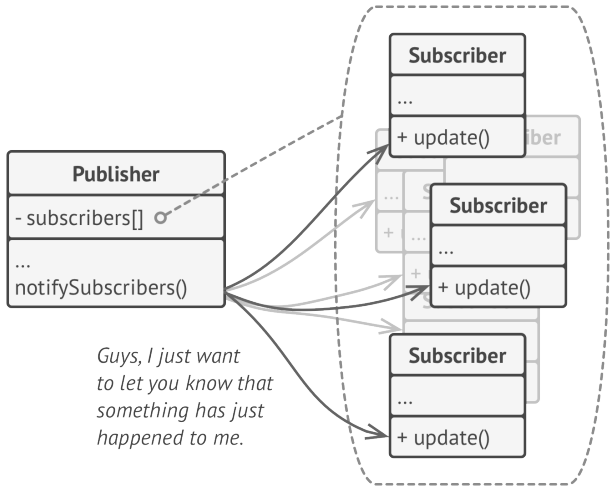


A subscription mechanism lets individual objects subscribe to event notifications.

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.

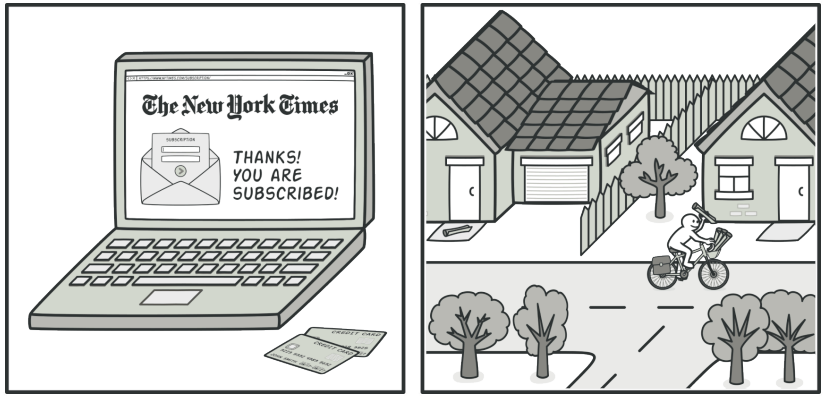


Publisher notifies subscribers by calling the specific notification method on their objects.

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

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.

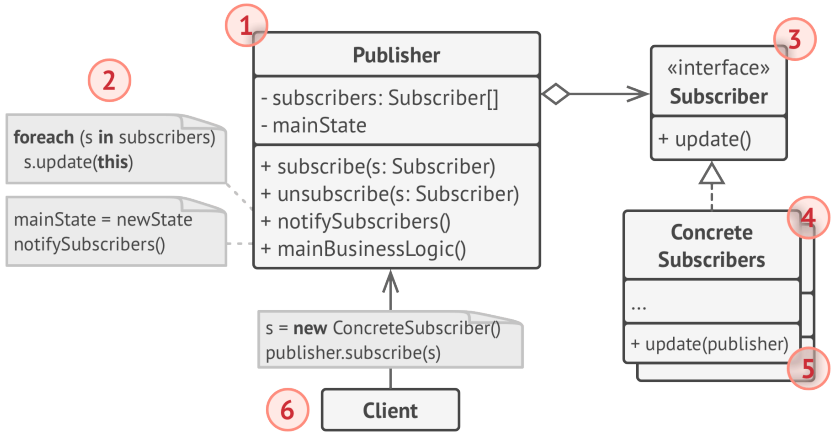


Magazine and newspaper subscriptions.

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.

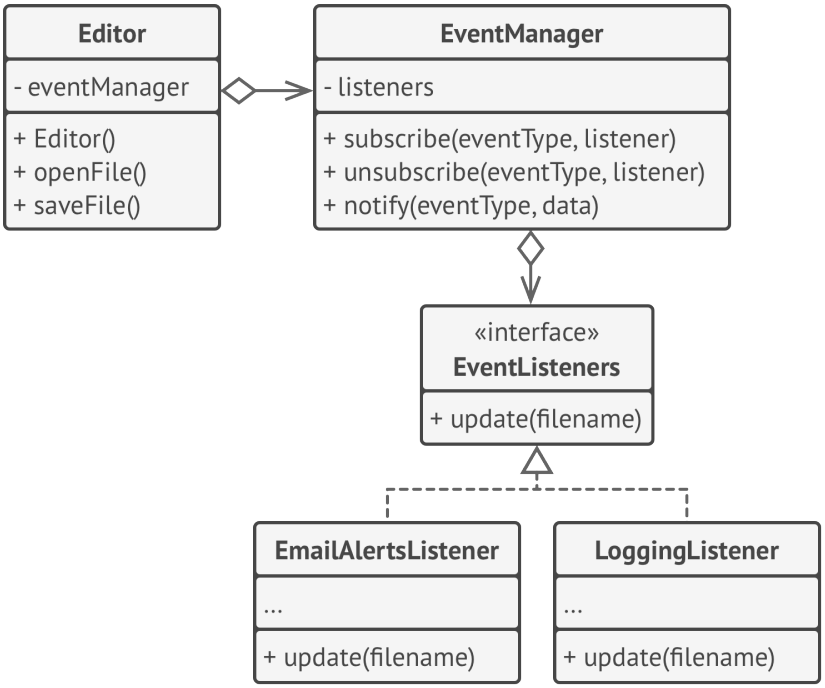
4. **Concrete Subscribers** perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.

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
  private 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

    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

  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 TextEditor()

    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.

5. Create concrete publisher classes. Each time something important happens inside a publisher, it must notify all its subscribers.

6. 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.

7. 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**, **Command**, **Mediator** and **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.

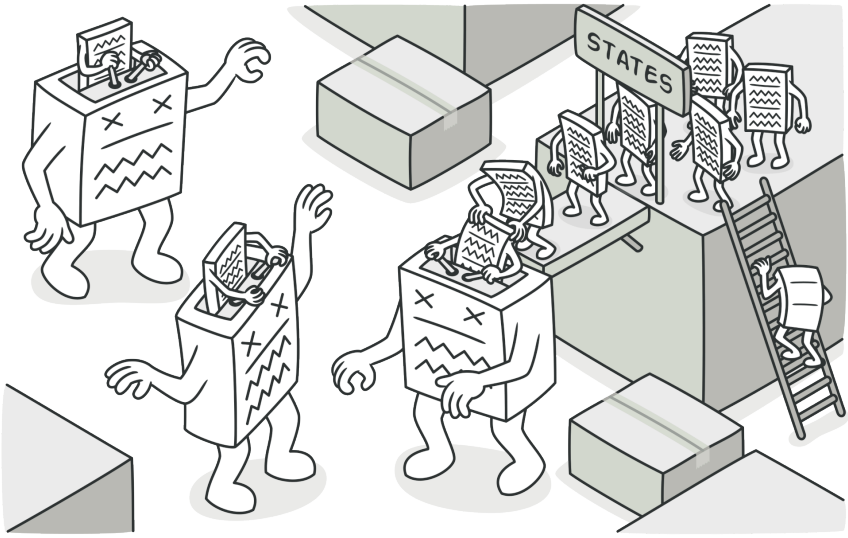
- The difference between **Mediator** and **Observer** is often elusive. In most cases, you can implement either of these patterns; but sometimes you can apply both simultaneously. Let's see how we can do that.

  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.

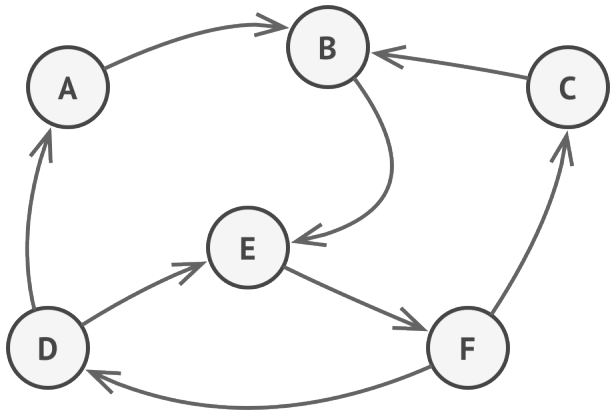


STATE

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

The State pattern is closely related to the concept of a **Finite-State Machine**.

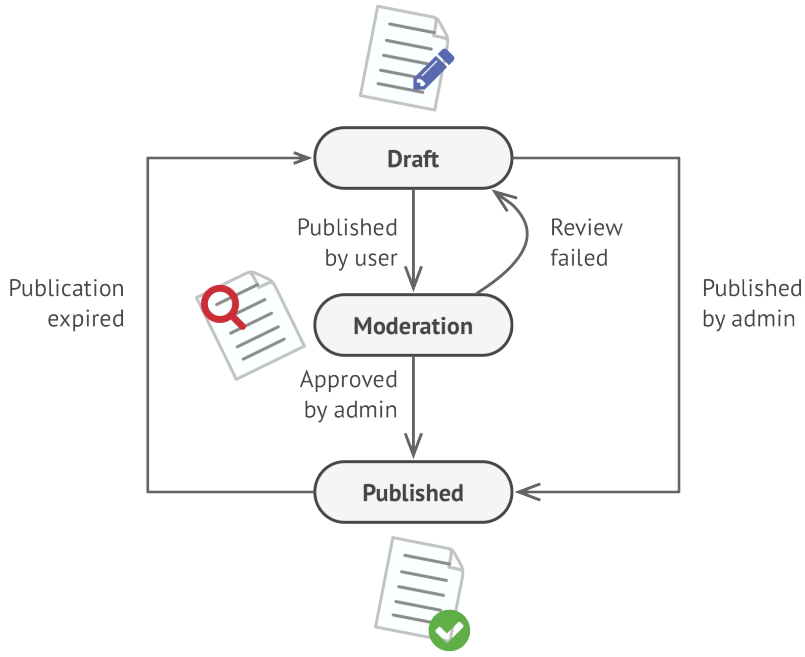


Finite-State Machine.

The main idea is that, at any given moment, there's a *finite* number of *states* 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.

You can also apply this approach to objects. Imagine that we have a `Document` class. A document can be in one of three states: `Draft`, `Moderation` and `Published`. The `publish` method of the document works a little bit differently in each state:

- In `Draft`, it moves the document to moderation.
- In `Moderation`, it makes the document public, but only if the current user is an administrator.
- In `Published`, it doesn't do anything at all.



Possible states and transitions of a document object.

State machines are usually implemented with lots of conditional operators (`if` or `switch`) that select the appropriate behavior depending on the current state of the object. Usually, this "state" is just a set of values of the object's fields. Even if you've never heard about finite-state machines before, you've probably implemented a state at least once. Does the following code structure ring a bell?

```
class Document is
  field state: string
  // ...
  method publish() is
    switch (state)
      "draft":
        state = "moderation"
        break
      "moderation":
        if (currentUser.role == 'admin')
          state = "published"
        break
      "published":
        // Do nothing.
        break
  // ...
```

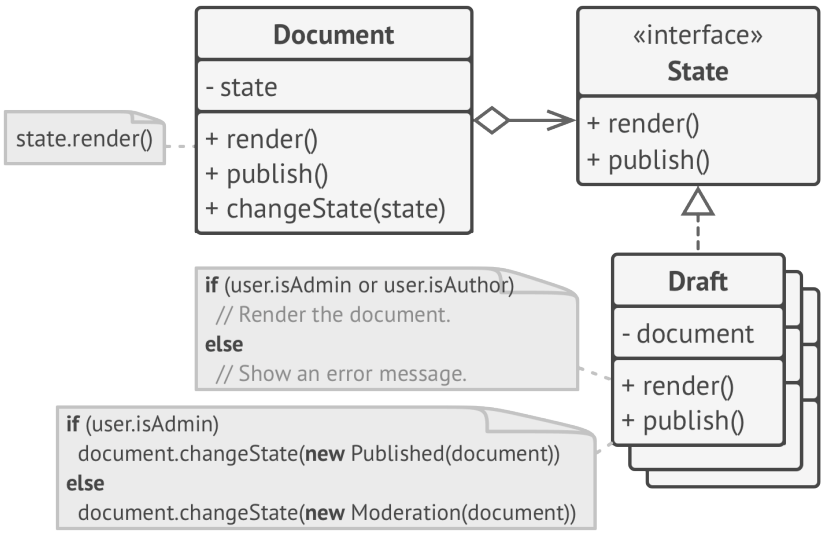
The biggest weakness of a state machine based on conditionals reveals itself once we start adding more and more states and state-dependent behaviors to the `Document` class. Most methods will contain monstrous conditionals that pick the proper behavior of a method according to the current state. Code like this is very difficult to maintain because any change to the transition logic may require changing state conditionals in every method.

The problem tends to get bigger as a project evolves. It's quite difficult to predict all possible states and transitions at the design stage. Hence, a lean state machine built with a limited set of conditionals can grow into a bloated mess over time.

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.

Instead of implementing all behaviors on its own, the original object, called *context*, stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object.



Document delegates the work to a state object.

To transition the context into another state, replace the active state object with another object that represents that new state. This is possible only if all state classes follow the same inter-

face and the context itself works with these objects through that interface.

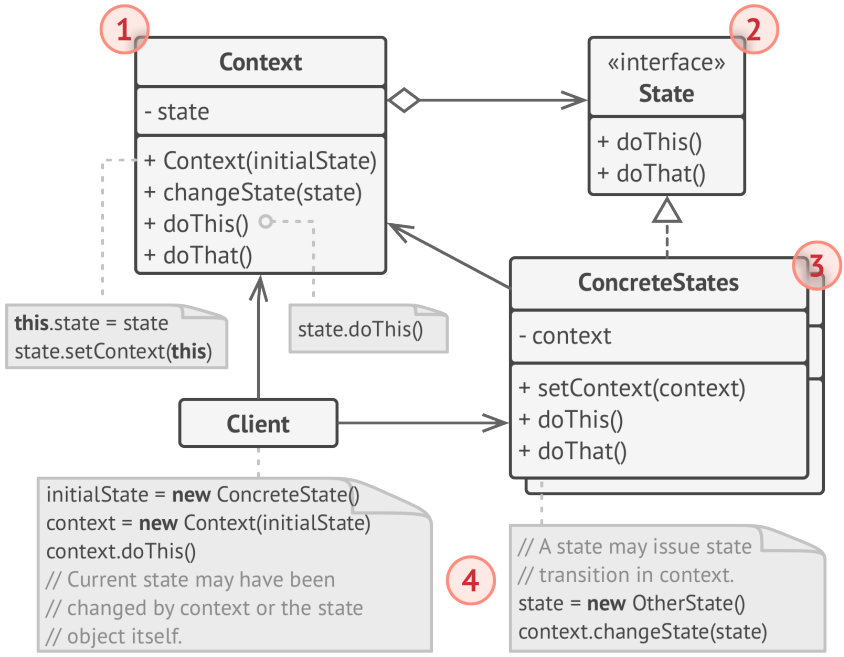
This structure may look similar to the **Strategy** pattern, but there's one key difference. In the State pattern, the particular states may be aware of each other and initiate transitions from one state to another, whereas strategies almost never know about each other.

Real-World Analogy

The buttons and switches in your smartphone behave differently depending on the current state of the device:

- When the phone is unlocked, pressing buttons leads to executing various functions.
- When the phone is locked, pressing any button leads to the unlock screen.
- When the phone's charge is low, pressing any button shows the charging screen.

Structure



1. **Context** stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

2. The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will never be called.

3. **Concrete States** provide their own implementations for the state-specific methods. To avoid duplication of similar code

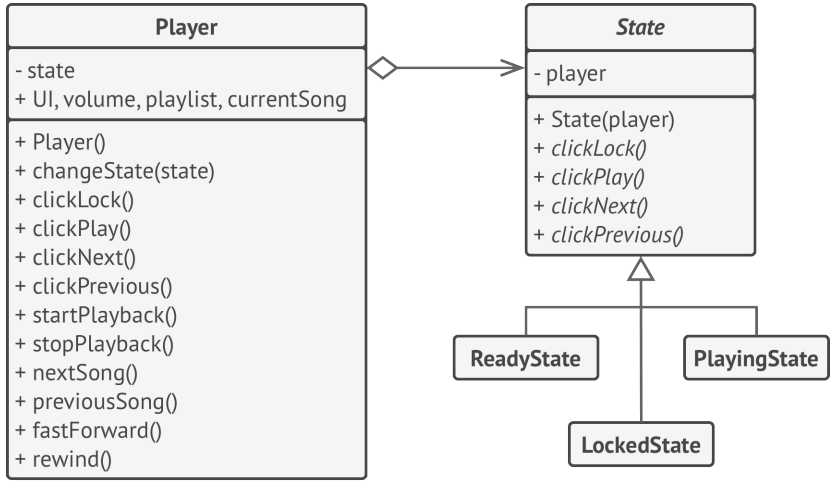
across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

State objects may store a backreference to the context object. Through this reference, the state can fetch any required info from the context object, as well as initiate state transitions.

4. Both context and concrete states can set the next state of the context and perform the actual state transition by replacing the state object linked to the context.

Pseudocode

In this example, the **State** pattern lets the same controls of the media player behave differently, depending on the current playback state.



Example of changing object behavior with state objects.

The main object of the player is always linked to a state object that performs most of the work for the player. Some actions replace the current state object of the player with another, which changes the way the player reacts to user interactions.

```
// The AudioPlayer class acts as a context. It also maintains a
// reference to an instance of one of the state classes that
// represents the current state of the audio player.
class AudioPlayer is
  field state: State
  field UI, volume, playlist, currentSong

  constructor AudioPlayer() is
    this.state = new ReadyState(this)

    // Context delegates handling user input to a state
    // object. Naturally, the outcome depends on what state
    // is currently active, since each state can handle the
    // input differently.
    UI = new UserInterface()
    UI.lockButton.onClick(this.clickLock)
    UI.playButton.onClick(this.clickPlay)
    UI.nextButton.onClick(this.clickNext)
    UI.prevButton.onClick(this.clickPrevious)

  // Other objects must be able to switch the audio player's
  // active state.
  method changeState(state: State) is
    this.state = state


```

```
    // UI methods delegate execution to the active state.
    method clickLock() is
      state.clickLock()
    method clickPlay() is
      state.clickPlay()
    method clickNext() is
      state.clickNext()
    method clickPrevious() is
      state.clickPrevious()

    // A state may call some service methods on the context.
    method startPlayback() is
      // ...
    method stopPlayback() is
      // ...
    method nextSong() is
      // ...
    method previousSong() is
      // ...
    method fastForward(time) is
      // ...
    method rewind(time) is
      // ...


// The base state class declares methods that all concrete
// states should implement and also provides a backreference to
// the context object associated with the state. States can use
// the backreference to transition the context to another state.
abstract class State is
    protected field player: AudioPlayer

```

```
    // Context passes itself through the state constructor. This
    // may help a state fetch some useful context data if it's
    // needed.
    constructor State(player) is
      this.player = player

    abstract method clickLock()
    abstract method clickPlay()
    abstract method clickNext()
    abstract method clickPrevious()


// Concrete states implement various behaviors associated with a
// state of the context.
class LockedState extends State is

    // When you unlock a locked player, it may assume one of two
    // states.
    method clickLock() is
      if (player.playing)
        player.changeState(new PlayingState(player))
      else
        player.changeState(new ReadyState(player))

    method clickPlay() is
      // Locked, so do nothing.

    method clickNext() is
      // Locked, so do nothing.

    method clickPrevious() is
      // Locked, so do nothing.
```

```
// They can also trigger state transitions in the context.
class ReadyState extends State is
  method clickLock() is
    player.changeState(new LockedState(player))

  method clickPlay() is
    player.startPlayback()
    player.changeState(new PlayingState(player))

  method clickNext() is
    player.nextSong()

  method clickPrevious() is
    player.previousSong()


class PlayingState extends State is
  method clickLock() is
    player.changeState(new LockedState(player))

  method clickPlay() is
    player.stopPlayback()
    player.changeState(new ReadyState(player))

  method clickNext() is
    if (event.doubleclick)
      player.nextSong()
    else
      player.fastForward(5)

  method clickPrevious() is
    if (event.doubleclick)
```

```
123         player.previous()
124       else
125         player.rewind(5)
```

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.

The pattern suggests that you extract all state-specific code into a set of distinct classes. As a result, you can add new states or change existing ones independently of each other, reducing the maintenance cost.

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.

The State pattern lets you extract branches of these conditionals into methods of corresponding state classes. While doing so, you can also clean temporary fields and helper methods involved in state-specific code out of your main class.

Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.

The State pattern lets you compose hierarchies of state classes and reduce duplication by extracting common code into abstract base classes.

How to Implement

1. Decide what class will act as the context. It could be an existing class which already has the state-dependent code; or a new class, if the state-specific code is distributed across multiple classes.

2. Declare the state interface. Although it may mirror all the methods declared in the context, aim only for those that may contain state-specific behavior.

3. For every actual state, create a class that derives from the state interface. Then go over the methods of the context and extract all code related to that state into your newly created class.

   While moving the code to the state class, you might discover that it depends on private members of the context. There are several workarounds:

   - Make these fields or methods public.
   - Turn the behavior you're extracting into a public method in the context and call it from the state class. This way is ugly but quick, and you can always fix it later.

- Nest the state classes into the context class, but only if your programming language supports nesting classes.

4. In the context class, add a reference field of the state interface type and a public setter that allows overriding the value of that field.

5. Go over the method of the context again and replace empty state conditionals with calls to corresponding methods of the state object.

6. To switch the state of the context, create an instance of one of the state classes and pass it to the context. You can do this within the context itself, or in various states, or in the client. Wherever this is done, the class becomes dependent on the concrete state class that it instantiates.

⚖ Pros and Cons

✓ *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.

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

⇄ Relations with Other Patterns

- **Bridge**, **State**, **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. 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.

- **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 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.

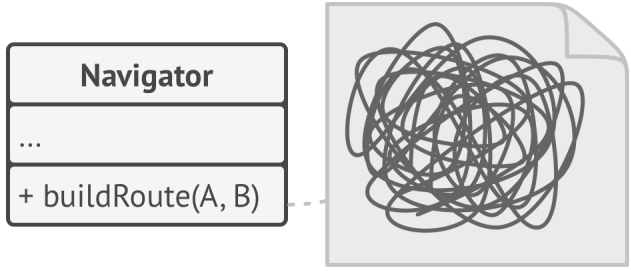
☹ Problem

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.

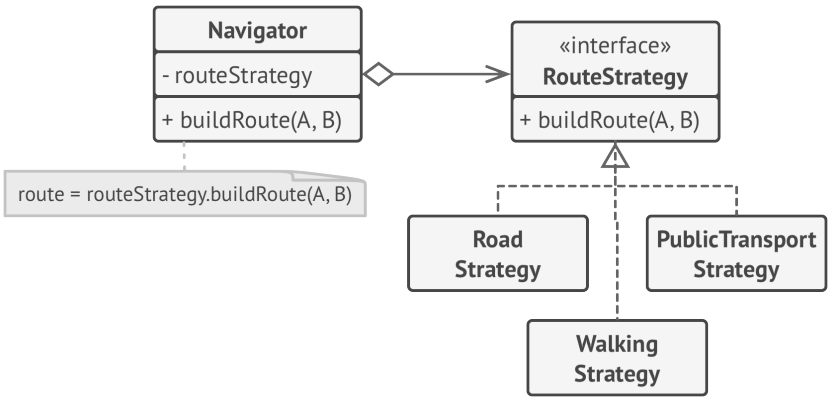
😊 Solution

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.



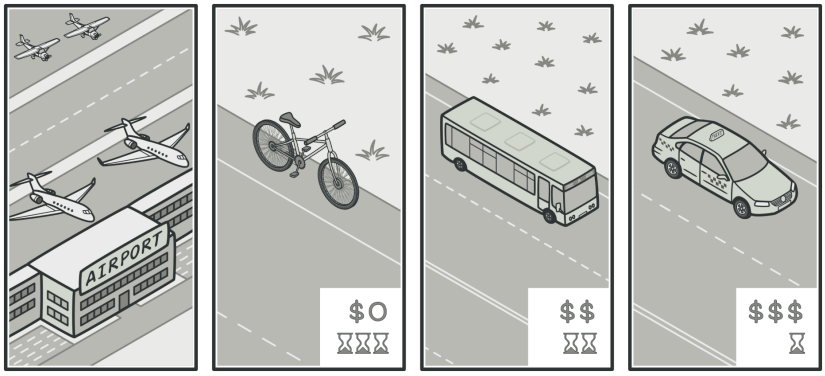
Route planning strategies.

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.

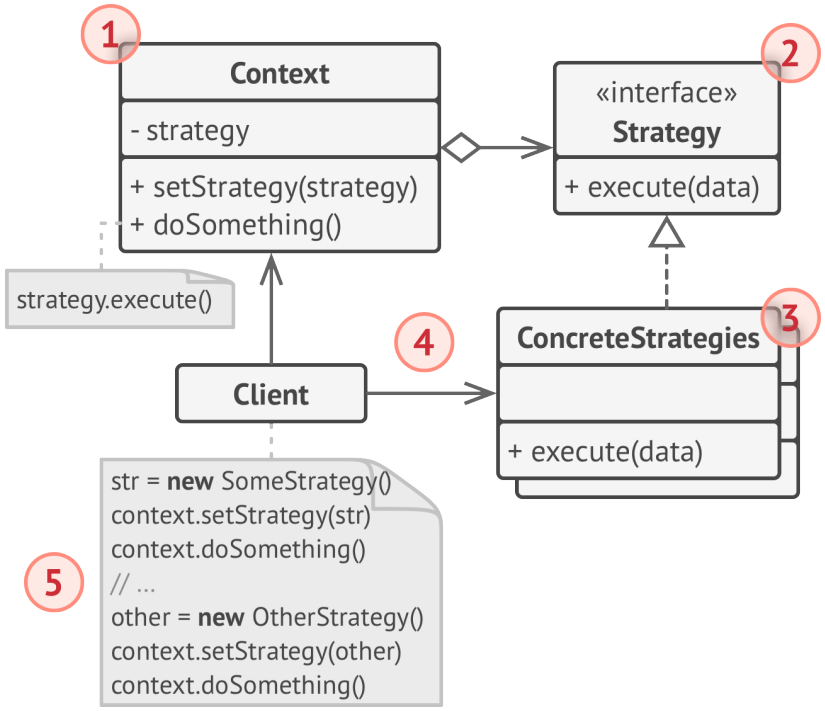
Real-World Analogy



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 operator 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**, **State**, **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. 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** and **Strategy** may look similar because you can use both to parameterize an object with some action. However, they have very different intents.

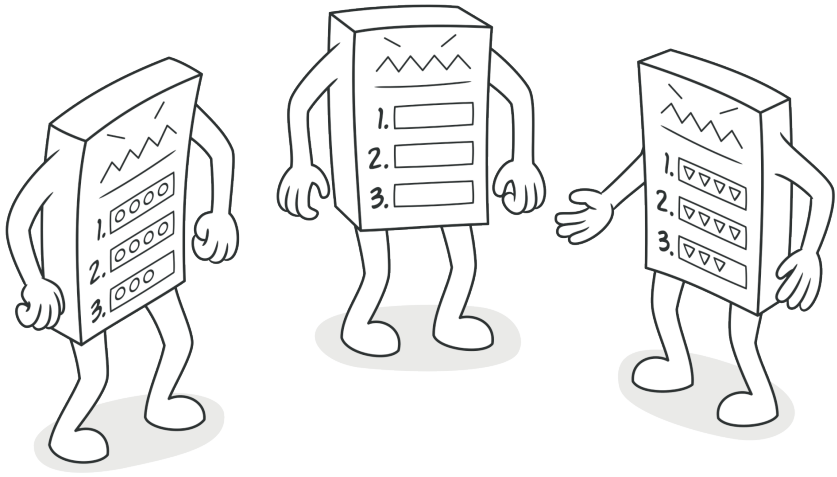
  - You can use *Command* to convert any operation into an object. The operation's parameters become fields of that object. The conversion lets you defer execution of the operation, queue it, store the history of commands, send commands to remote services, etc.

  - 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** 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** 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.



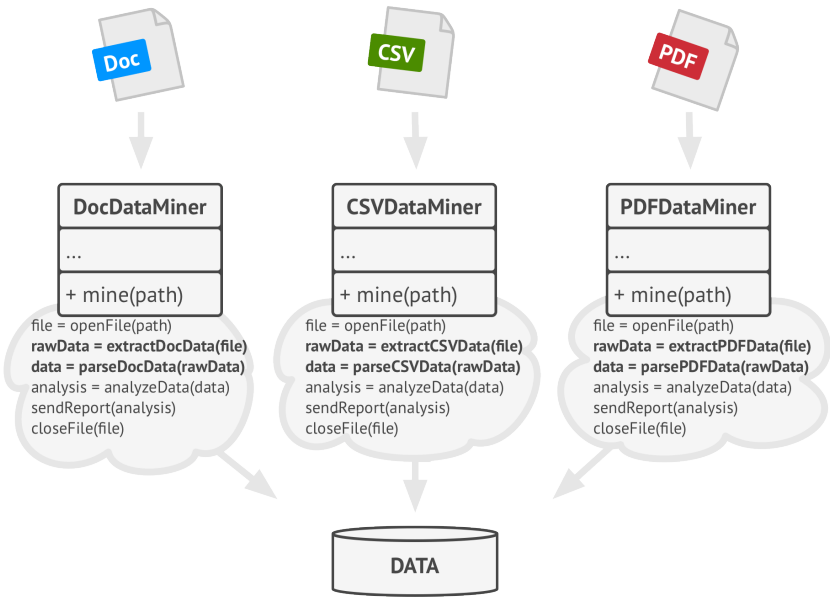
TEMPLATE METHOD

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.

The first version of the app could work only with DOC files. In the following version, it was able to support CSV files. A month later, you "taught" it to extract data from PDF files.



Data mining classes contained a lot of duplicate code.

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 pro-

cessing and analysis is almost identical. Wouldn't it be great to get rid of the code duplication, leaving the algorithm structure intact?

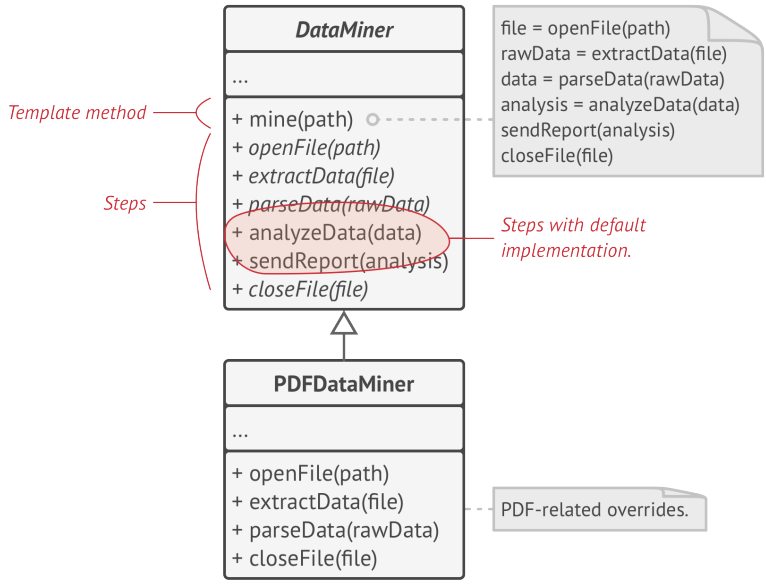
There was another problem related to client code that used these classes. It had lots of conditionals that picked a proper course of action depending on the class of the processing object. If all three processing classes had a common interface or a base class, you'd be able to eliminate the conditionals in client code and use polymorphism when calling methods on a processing object.

😊 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).

Let's see how this will play out in our data mining app. We can create a base class for all three parsing algorithms. This class defines a template method consisting of a series of calls to various document-processing steps.

At first, we can declare all steps `abstract`, forcing the subclasses to provide their own implementations for these methods. In our case, subclasses already have all necessary implementations, so the only thing we might need to do is adjust signatures of the methods to match the methods of the superclass.



Template method breaks the algorithm into steps, allowing subclasses to override these steps but not the actual method.

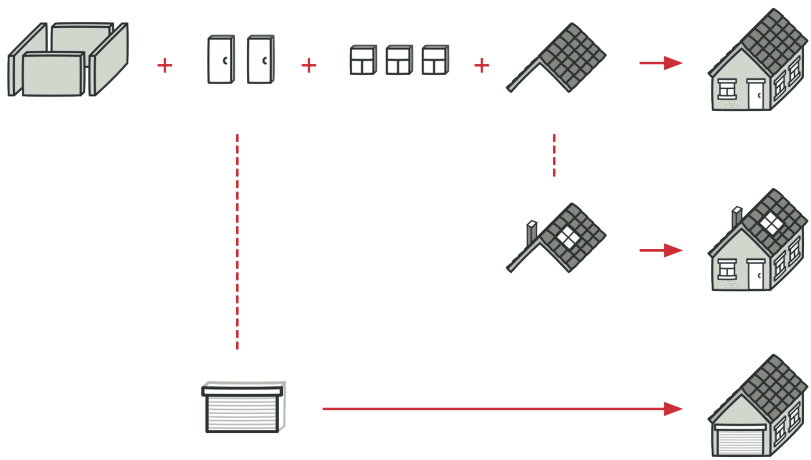
Now, let's see what we can do to get rid of the duplicate code. It looks like the code for opening/closing files and extracting/parsing data is different for various data formats, so there's no point in touching those methods. However, implementation of other steps, such as analyzing the raw data and composing reports, is very similar, so it can be pulled up into the base class, where subclasses can share that code.

As you can see, we've got two types of steps:

- *abstract steps* must be implemented by every subclass
- *optional steps* already have some default implementation, but still can be overridden if needed

There's another type of step, called *hooks*. A hook is an optional step with an empty body. A template method would work even if a hook isn't overridden. Usually, hooks are placed before and after crucial steps of algorithms, providing subclasses with additional extension points for an algorithm.

Real-World Analogy



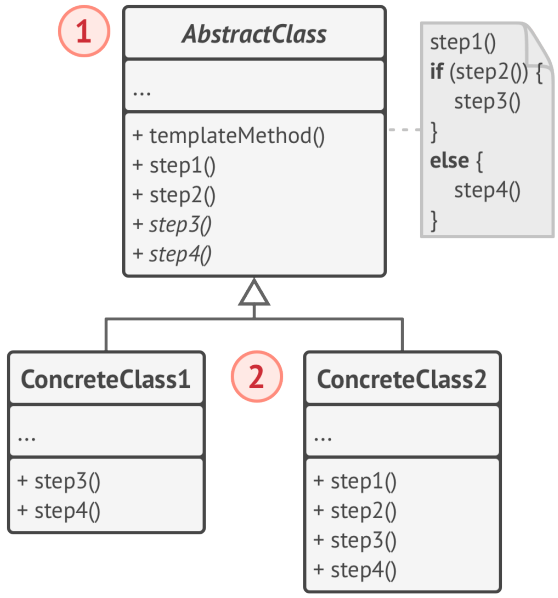
A typical architectural plan can be slightly altered to better fit the client's needs.

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.

Each building step, such as laying the foundation, framing, building walls, installing plumbing and wiring for water and electricity, etc., can be slightly changed to make the resulting house a little bit different from others.

Structure

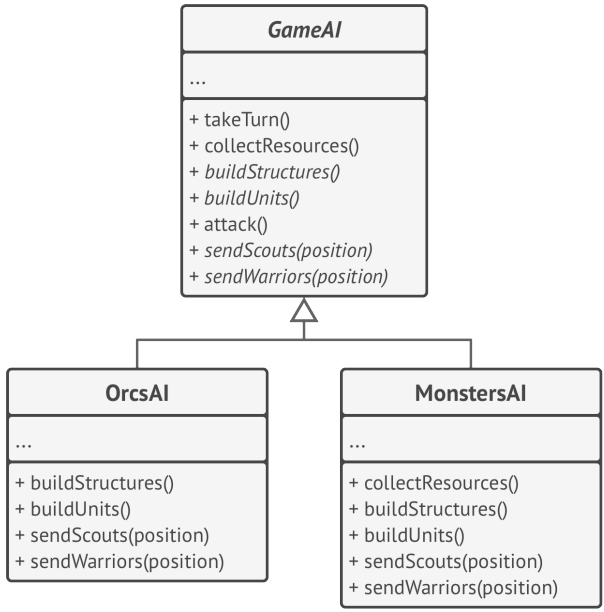


1. The **Abstract Class** declares methods that act as steps of an algorithm, as well as the actual template method which calls these methods in a specific order. The steps may either be declared `abstract` or have some default implementation.

2. **Concrete Classes** can override all of the steps, but not the template method itself.

Pseudocode

In this example, the **Template Method** pattern provides a "skeleton" for various branches of artificial intelligence in a simple strategy video game.



AI classes of a simple video game.

All races in the game have almost the same types of units and buildings. Therefore you can reuse the same AI structure for various races, while being able to override some of the details. With this approach, you can override the orcs' AI to make it more aggressive, make humans more defense-oriented, and

make monsters unable to build anything. Adding a new race to the game would require creating a new AI subclass and overriding the default methods declared in the base AI class.

```
// The abstract class defines a template method that contains a
// skeleton of some algorithm composed of calls, usually to
// abstract primitive operations. Concrete subclasses implement
// these operations, but leave the template method itself
// intact.
class GameAI is
  // The template method defines the skeleton of an algorithm.
  method turn() is
    collectResources()
    buildStructures()
    buildUnits()
    attack()

  // Some of the steps may be implemented right in a base
  // class.
  method collectResources() is
    foreach (s in this.builtStructures) do
      s.collect()

  // And some of them may be defined as abstract.
  abstract method buildStructures()
  abstract method buildUnits()

  // A class can have several template methods.
  method attack() is
    enemy = closestEnemy()
    if (enemy == null)
```

```
      sendScouts(map.center)
    else
      sendWarriors(enemy.position)

  abstract method sendScouts(position)
  abstract method sendWarriors(position)

// Concrete classes have to implement all abstract operations of
// the base class but they must not override the template method
// itself.
class OrcsAI extends GameAI is
  method buildStructures() is
    if (there are some resources) then
      // Build farms, then barracks, then stronghold.

  method buildUnits() is
    if (there are plenty of resources) then
      if (there are no scouts)
        // Build peon, add it to scouts group.
      else
        // Build grunt, add it to warriors group.

  // ...

  method sendScouts(position) is
    if (scouts.length > 0) then
      // Send scouts to position.

  method sendWarriors(position) is
    if (warriors.length > 5) then
      // Send warriors to position.

```

```
// Subclasses can also override some operations with a default
// implementation.
class MonstersAI extends GameAI is
  method collectResources() is
    // Monsters don't collect resources.

  method buildStructures() is
    // Monsters don't build structures.

  method buildUnits() is
    // Monsters don't build units.
```

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.

The Template Method lets you turn a monolithic algorithm into a series of individual steps which can be easily extended by subclasses while keeping intact the structure defined in a superclass.

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 both classes when the algorithm changes.

When you turn such an algorithm into a template method, you can also pull up the steps with similar implementations into a superclass, eliminating code duplication. Code that varies between subclasses can remain in subclasses.

How to Implement

1. Analyze the target algorithm to see whether you can break it into steps. Consider which steps are common to all subclasses and which ones will always be unique.

2. Create the abstract base class and declare the template method and a set of abstract methods representing the algorithm's steps. Outline the algorithm's structure in the template method by executing corresponding steps. Consider making the template method `final` to prevent subclasses from overriding it.

3. It's okay if all the steps end up being abstract. However, some steps might benefit from having a default implementation. Subclasses don't have to implement those methods.

4. Think of adding hooks between the crucial steps of the algorithm.

5. For each variation of the algorithm, create a new concrete subclass. It *must* implement all of the abstract steps, but *may* also override some of the optional ones.

Pros and Cons

✓ 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.

✕ Some clients may be limited by the provided skeleton of an algorithm.

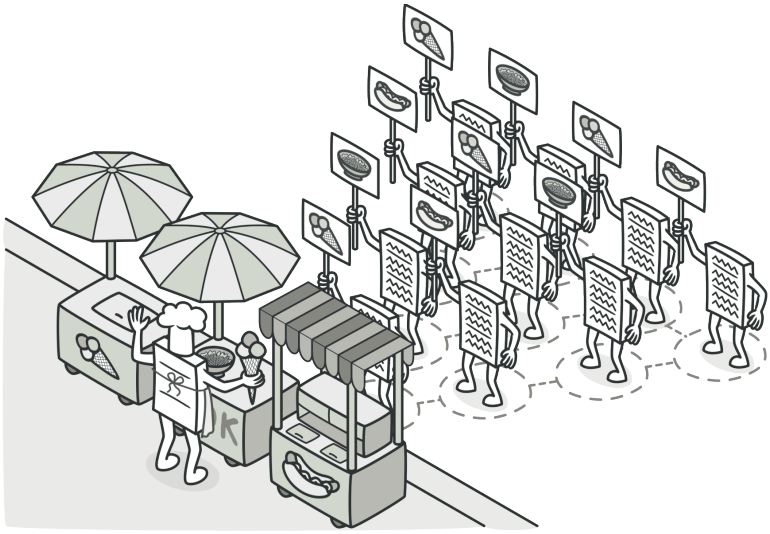
✕ You might violate the *Liskov Substitution Principle* by suppressing a default step implementation via a subclass.

✕ Template methods tend to be harder to maintain the more steps they have.

Relations 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. *Strategy* works on the object level, letting you switch behaviors at runtime.

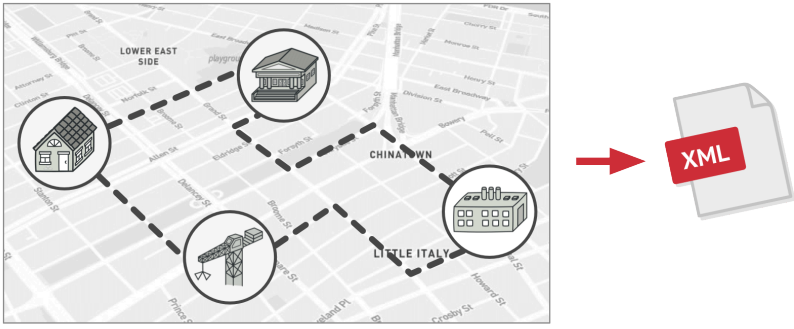


VISITOR

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

Problem

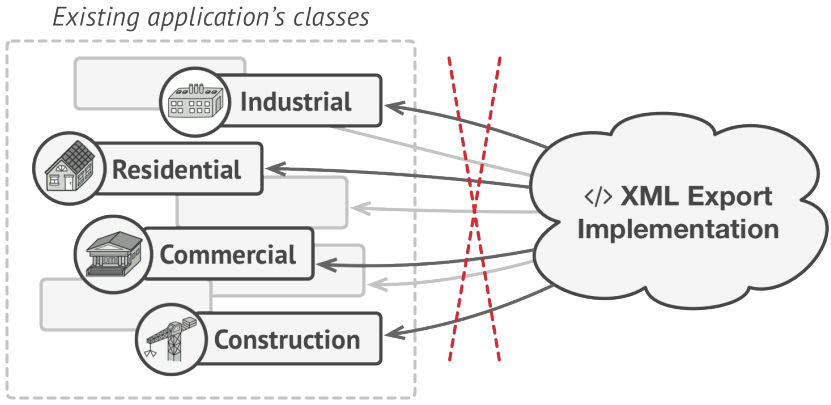
Imagine that your team develops an app which works with geographic information structured as one colossal graph. Each node of the graph may represent a complex entity such as a city, but also more granular things like industries, sightseeing areas, etc. The nodes are connected with others if there's a road between the real objects that they represent. Under the hood, each node type is represented by its own class, while each specific node is an object.



Exporting the graph into XML.

At some point, you got a task to implement exporting the graph into XML format. At first, the job seemed pretty straightforward. You planned to add an export method to each node class and then leverage recursion to go over each node of the graph, executing the export method. The solution was simple and elegant: thanks to polymorphism, you weren't coupling the code which called the export method to concrete classes of nodes.

Unfortunately, the system architect refused to allow you to alter existing node classes. He said that the code was already in production and he didn't want to risk breaking it because of a potential bug in your changes.



The XML export method had to be added into all node classes, which bore the risk of breaking the whole application if any bugs slipped through along with the change.

Besides, he questioned whether it makes sense to have the XML export code within the node classes. The primary job of these classes was to work with geodata. The XML export behavior would look alien there.

There was another reason for the refusal. It was highly likely that after this feature was implemented, someone from the marketing department would ask you to provide the ability to export into a different format, or request some other weird stuff. This would force you to change those precious and fragile classes again.

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 visitor's methods as an argument, providing the method access to all necessary data contained within the object.

Now, what if that behavior can be executed over objects of different classes? For example, in our case with XML export, the actual implementation will probably be a little bit different across various node classes. Thus, the visitor class may define not one, but a set of methods, each of which could take arguments of different types, like this:

```
class ExportVisitor implements Visitor is
  method doForCity(City c) { ... }
  method doForIndustry(Industry f) { ... }
  method doForSightSeeing(SightSeeing ss) { ... }
  // ...
```

But how exactly would we call these methods, especially when dealing with the whole graph? These methods have different signatures, so we can't use polymorphism. To pick a proper visitor method that's able to process a given object, we'd need to check its class. Doesn't this sound like a nightmare?

```
foreach (Node node in graph)
  if (node instanceof City)
    exportVisitor.doForCity((City) node)
  if (node instanceof Industry)
    exportVisitor.doForIndustry((Industry) node)
  // ...
}
```

You might ask, why don't we use method overloading? That's when you give all methods the same name, even if they support different sets of parameters. Unfortunately, even assuming that our programming language supports it at all (as Java and C# do), it won't help us. Since the exact class of a node object is unknown in advance, the overloading mechanism won't be able to determine the correct method to execute. It'll default to the method that takes an object of the base `Node` class.

However, the Visitor pattern addresses this problem. It uses a technique called **Double Dispatch**, which helps to execute the proper method on an object without cumbersome conditionals. Instead of letting the client select a proper version of the method to call, how about we delegate this choice to objects we're passing to the visitor as an argument?

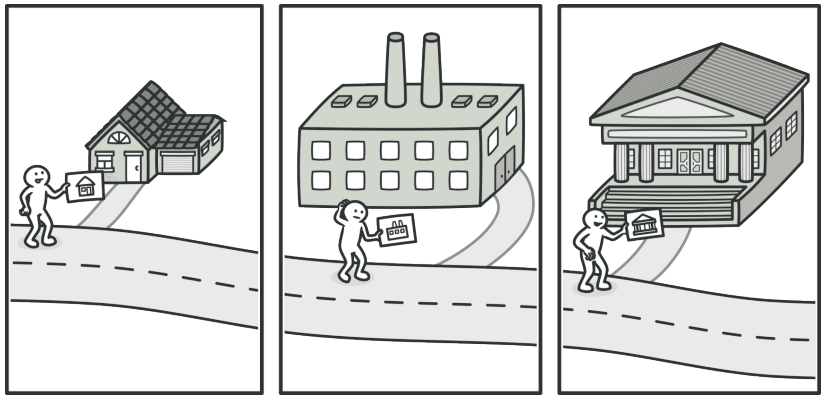
Since the objects know their own classes, they'll be able to pick a proper method on the visitor less awkwardly. They "accept" a visitor and tell it what visiting method should be executed.

```
// Client code
foreach (Node node in graph)
  node.accept(exportVisitor)

// City
class City is
  method accept(Visitor v) is
    v.doForCity(this)
  // ...

// Industry
class Industry is
  method accept(Visitor v) is
    v.doForIndustry(this)
  // ...
```

I confess. We had to change the node classes after all. But at least the change is trivial and it lets us add further behaviors without altering the code once again.

Now, if we extract a common interface for all visitors, all existing nodes can work with any visitor you introduce into the app. If you find yourself introducing a new behavior related to nodes, all you have to do is implement a new visitor class.

Real-World Analogy

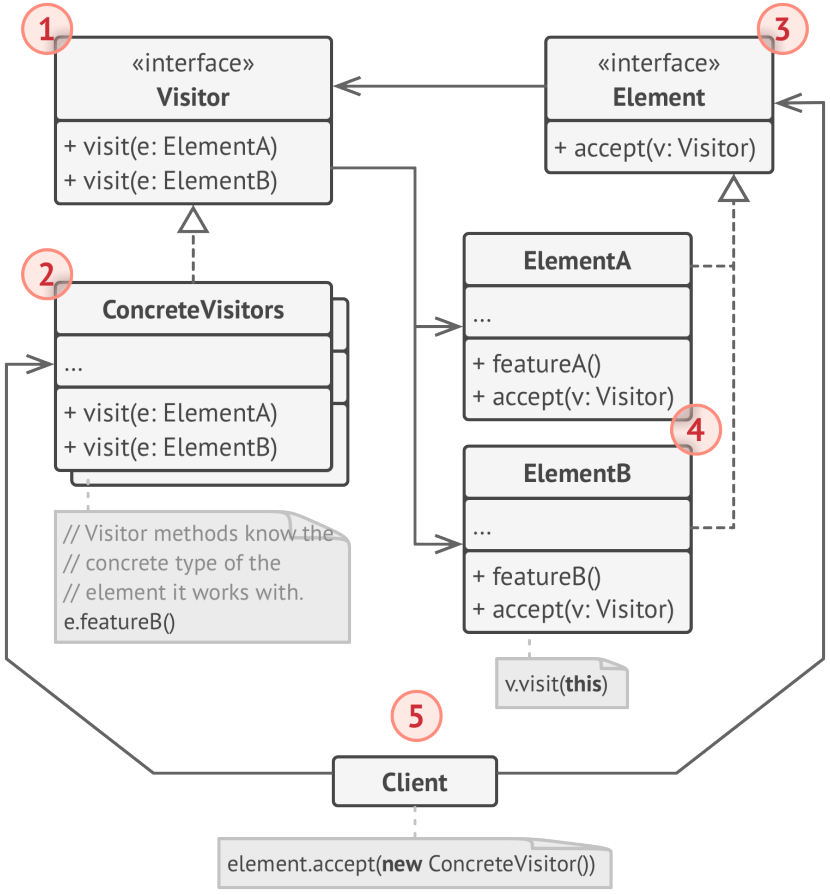


A good insurance agent is always ready to offer different policies to various types of organizations.

Imagine a seasoned 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:

- If it's a residential building, he sells medical insurance.
- If it's a bank, he sells theft insurance.
- If it's a coffee shop, he sells fire and flood insurance.

Structure



1. The **Visitor** interface declares a set of visiting methods that can take concrete elements of an object structure as arguments. These methods may have the same names if the program is written in a language that supports overloading, but the type of their parameters must be different.

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

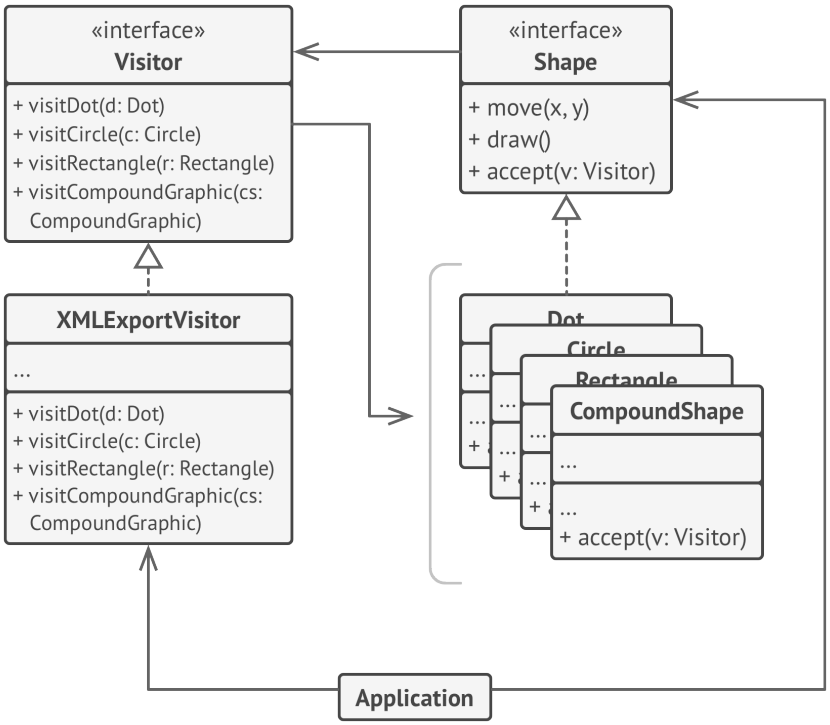
3. The **Element** interface declares a method for "accepting" visitors. This method should have one parameter declared with the type of the visitor interface.

4. 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. Be aware that even if a base element class implements this method, all subclasses must still override this method in their own classes and call the appropriate method on the visitor object.

5. 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.

Pseudocode

In this example, the **Visitor** pattern adds XML export support to the class hierarchy of geometric shapes.



Exporting various types of objects into XML format via a visitor object.

```
// The element interface declares an `accept` method that
// takes the base visitor interface as an argument.
interface Shape is
  method move(x, y)
  method draw()
  method accept(v: Visitor)

// Each concrete element class must implement the `accept`
// method in such a way that it calls the visitor's method that
// corresponds to the element's class.
class Dot extends Shape is
  // ...
```

```
  // Note that we're calling `visitDot`, which matches the
  // current class name. This way we let the visitor know the
  // class of the element it works with.
  method accept(v: Visitor) is
    v.visitDot(this)

class Circle extends Dot is
  // ...
  method accept(v: Visitor) is
    v.visitCircle(this)

class Rectangle extends Shape is
  // ...
  method accept(v: Visitor) is
    v.visitRectangle(this)

class CompoundShape implements Shape is
  // ...
  method accept(v: Visitor) is
    v.visitCompoundShape(this)


// The Visitor interface declares a set of visiting methods that
// correspond to element classes. The signature of a visiting
// method lets the visitor identify the exact class of the
// element that it's dealing with.
interface Visitor is
  method visitDot(d: Dot)
  method visitCircle(c: Circle)
  method visitRectangle(r: Rectangle)
  method visitCompoundShape(cs: CompoundShape)

```

```
// Concrete visitors implement several versions of the same
// algorithm, which can work with all concrete element
// classes.
//
// You can experience the biggest benefit of the Visitor pattern
// when using it with a complex object structure such as a
// Composite tree. In this case, it might be helpful to store
// some intermediate state of the algorithm while executing the
// visitor's methods over various objects of the structure.
class XMLExportVisitor is
  method visitDot(d: Dot) is
    // Export the dot's ID and center coordinates.

  method visitCircle(c: Circle) is
    // Export the circle's ID, center coordinates and
    // radius.

  method visitRectangle(r: Rectangle) is
    // Export the rectangle's ID, left-top coordinates,
    // width and height.

  method visitCompoundShape(cs: CompoundShape) is
    // Export the shape's ID as well as the list of its
    // children's IDs.


// The client code can run visitor operations over any set of
// elements without figuring out their concrete classes. The
// accept operation directs a call to the appropriate operation
// in the visitor object.
class Application is
  field allShapes: array of Shapes
```

```
77    method export() is
78      exportVisitor = new XMLExportVisitor()
79
80      foreach (shape in allShapes) do
81        shape.accept(exportVisitor)
```

If you wonder why we need the `accept` method in this example, my article **Visitor and Double Dispatch** addresses this question in detail.

Applicability

Use the Visitor when you need to perform an operation on all elements of a complex object structure (for example, an object tree).

The Visitor pattern lets you execute an operation over a set of objects with different classes by having a visitor object implement several variants of the same operation, which correspond to all target classes.

Use the Visitor to clean up the business logic of auxiliary behaviors.

The pattern lets you make the primary classes of your app more focused on their main jobs by extracting all other behaviors into a set of visitor classes.

Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.

You can extract this behavior into a separate visitor class and implement only those visiting methods that accept objects of relevant classes, leaving the rest empty.

How to Implement

1. Declare the visitor interface with a set of "visiting" methods, one per each concrete element class that exists in the program.

2. Declare the element interface. If you're working with an existing element class hierarchy, add the abstract "acceptance" method to the base class of the hierarchy. This method should accept a visitor object as an argument.

3. Implement the acceptance methods in all concrete element classes. These methods must simply redirect the call to a visiting method on the incoming visitor object which matches the class of the current element.

4. The element classes should only work with visitors via the visitor interface. Visitors, however, must be aware of all concrete element classes, referenced as parameter types of the visiting methods.

5. For each behavior that can't be implemented inside the element hierarchy, create a new concrete visitor class and implement all of the visiting methods.

   You might encounter a situation where the visitor will need access to some private members of the element class. In this case, you can either make these fields or methods public, violating the element's encapsulation, or nest the visitor class in the element class. The latter is only possible if you're lucky to work with a programming language that supports nested classes.

6. The client must create visitor objects and pass them into elements via "acceptance" methods.

Pros and Cons

- ✓ *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.

- ✕ 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.

⇄ Relations 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.

Conclusion

Congrats! You have reached the end of the book!

However, there are many other patterns in the world. I hope that the book will become your starting point for learning patterns and developing superhero program design abilities.

Here are a couple of ideas that will help you decide what to do next.

- </> Don't forget that you also have **access to an archive** of downloadable code samples in different programming languages.
- Read Joshua Kerievsky's "**Refactoring To Patterns**".
- Know nothing about refactoring? **I have a course for you**.
- Print out these **patterns cheat sheets** and put them somewhere where you'll be able to see them all the time.
- **Leave feedback** on this book. I'll be very excited to learn your opinion, even a highly critical one ☺