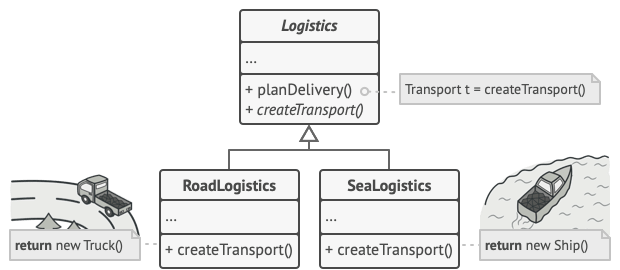
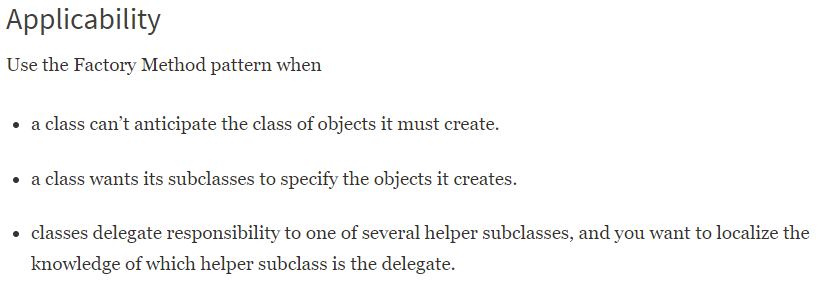
**Pattern Design Method**

# 1) Factory Method

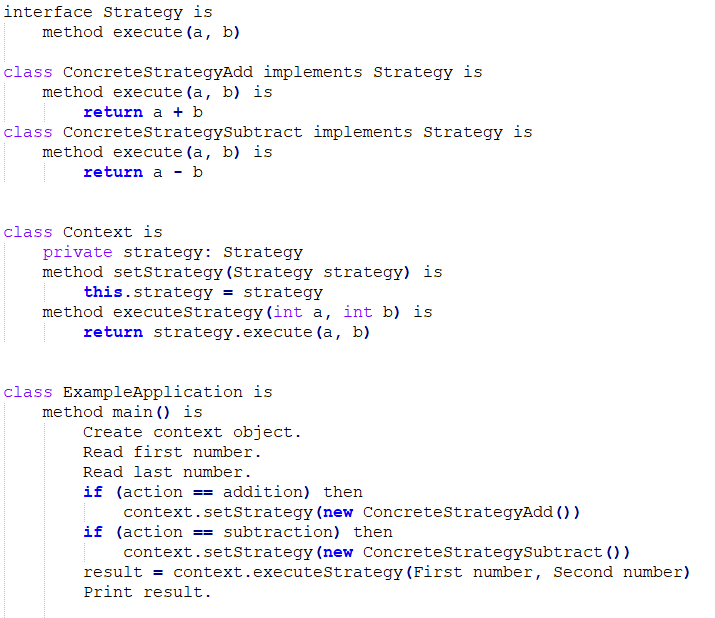
* Subclasses can alter the class object and return the subclasses result to the concrete class. This requires creating multiple class to handle the work in different way. What is interesting about this is that there will be a completed concrete class and for every new method or way, the class only need to extend/implement that concrete class and derived which ever function is there. This would avoid multiple function implementation into multiple class which serve for the same purpose.

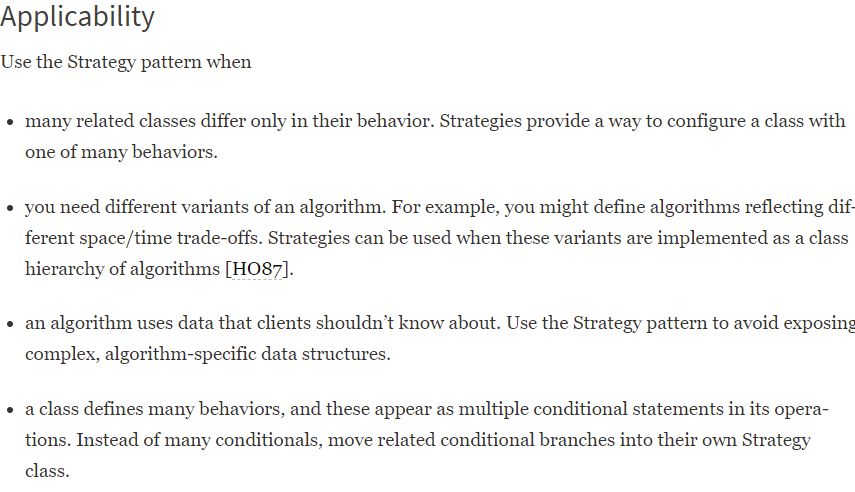


<https://refactoring.guru/design-patterns/factory-method>

# 2) Strategy Method

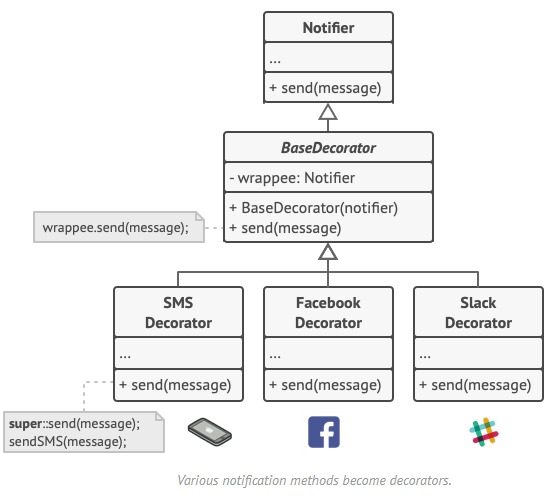
* Pretty much an extended version from Factory Method, with an extra class implement to avoid direct access to the class/concrete class. That extra class is called Strategy class, where it will take the user request and execute the request within that class. This is extremely useful when there are multiple interface or concrete class that client doesn’t know which one to use. Based on this and the appropriate implemented direction, we can avoid that hassle. When the class Strategy receive the request and understand which method to use, it will call the appropriate function according to the request in Strategy Implement class and perform the operation

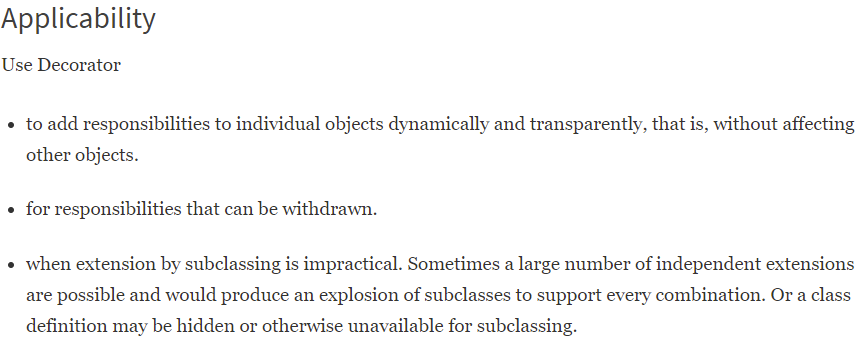


<https://refactoring.guru/design-patterns/strategy>

# 3) Decorator Method

* A concrete subclass play a role as a wrapper for multiple subclass belong to the same category. For examples, a regular Notification (which is BaseDecorator) would inherit the Notifier interface to implement its own function and notify user when necessary. However, each application has their own notification style so they would override what is on BaseDecorator to change their notification style and display their notification when their app is installed. If an application doesn’t specify its own Notification, it will just use the default one.

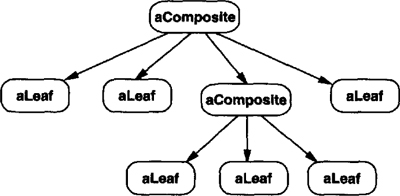
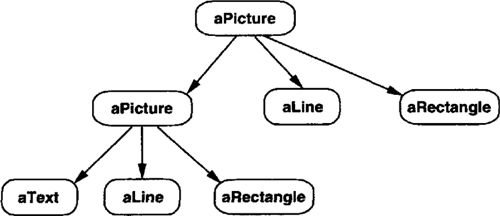


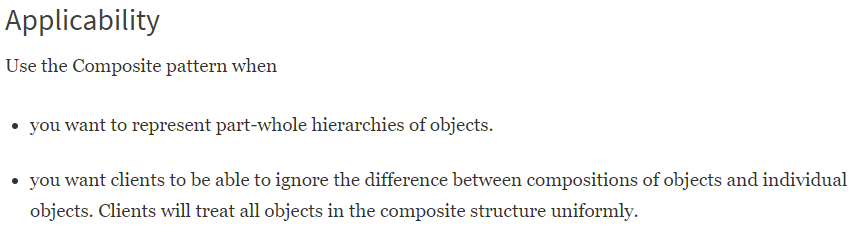


<https://refactoring.guru/design-patterns/decorator>

# 4) Composite Method

* Understand it as a loop. There will be an array of object and each object has their own job to do. What happen here is, when an object at array position 1 finish its work, only object at array position 2 can start doing it. If an object contains an array of object, then they must finish the work before coming back to the main array and continue. The pattern creates abstract classes that represents both primitives and containers. It implements default behavior for all the classes and declares interface for accessing and managing the child components. As well as allowing you to add new components easily. You could use this pattern when you want to represent hierarchies of objects for example a phone book application.

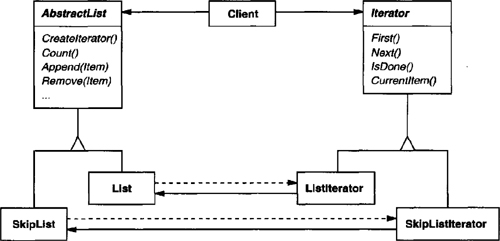


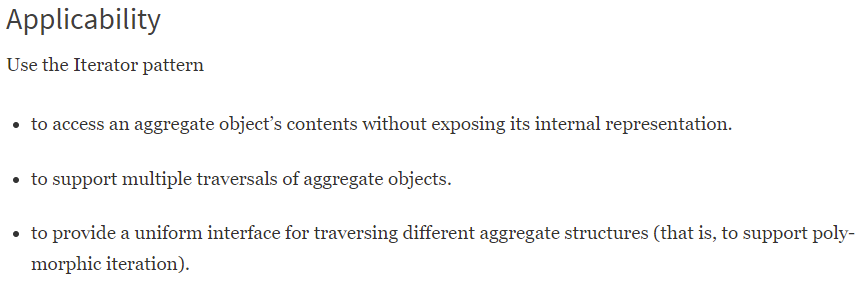


<https://refactoring.guru/design-patterns/composite>

# 5) Iterator Method

* Traverse the list either in forward or backward. The Iterator shouldn't expose the detail of the object in the list when accessing them. The Iterator consists of 4 basics method: First(), Next(), IsDone() and CurrentObject(). What it does is, user must know the list and the order of sequence to start with, but they shouldn’t know what is inside the list. If Abstract class is existing, each of the concrete subclass must have its own iterator class to traverse (same thing to Abstract)

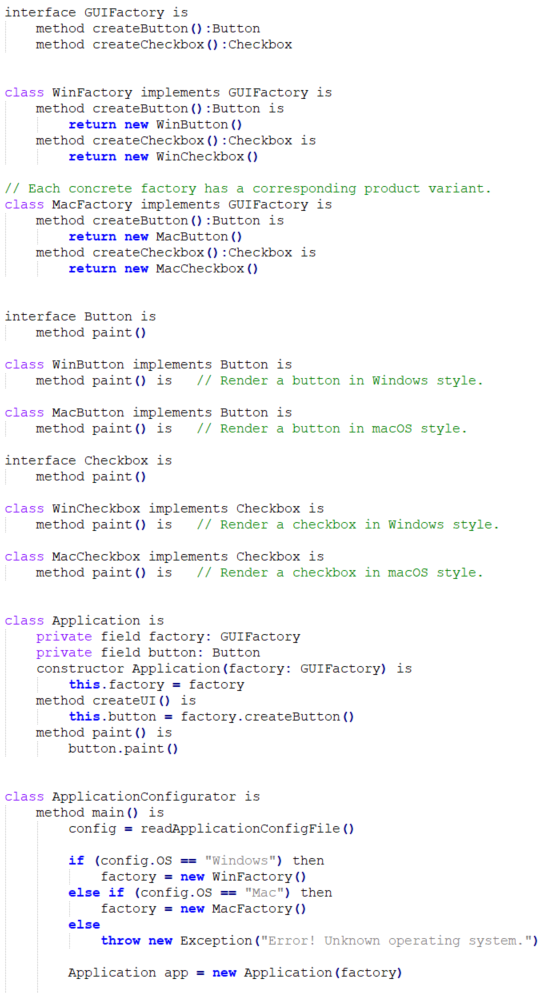


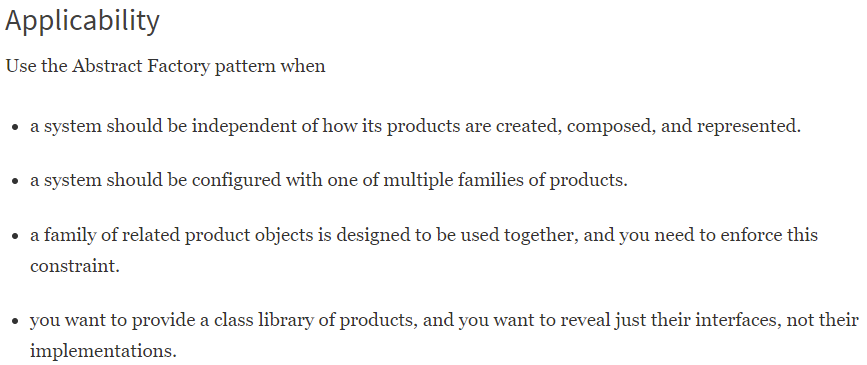


<https://refactoring.guru/design-patterns/iterator>

# 6) Abstract Factory

* Produce families of related object without specifying the exact type (ex. Sphere concrete subclass can contain multiple Sphere objects that has different shape size or alike). Consider this as multiple Factory method add together.

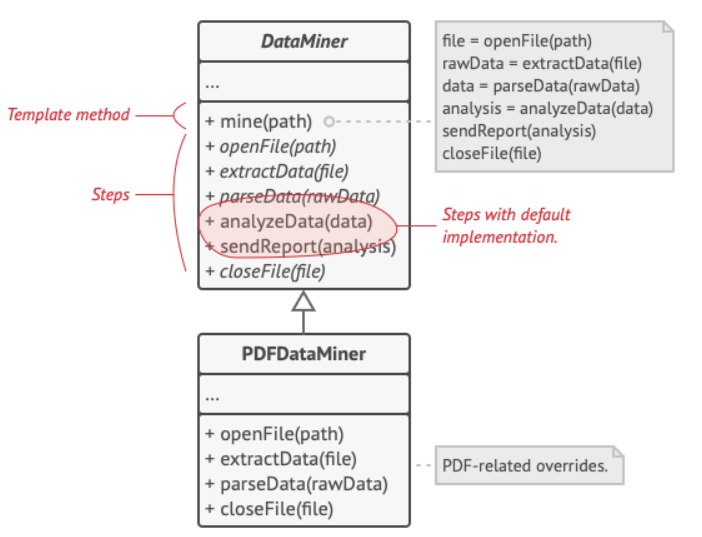


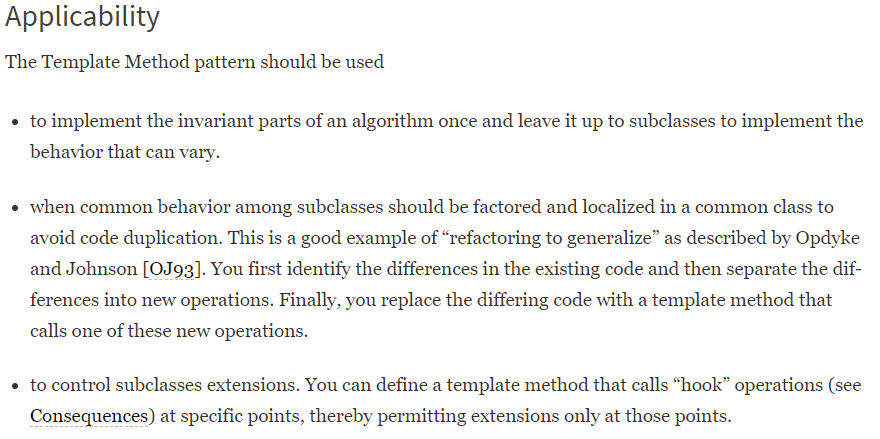


<https://refactoring.guru/design-patterns/abstract-factory>

# 7) Template Method

* Create a general Base class (with general implemented function) that other class can derived and override the method on the Base class if necessary, to perform a complete work. Example could be a card game. A card game will have the same functionality such as determine when the game end, who play next but each of the card game will have different rules. Each card game class can derive from that Base class and override the content if require.

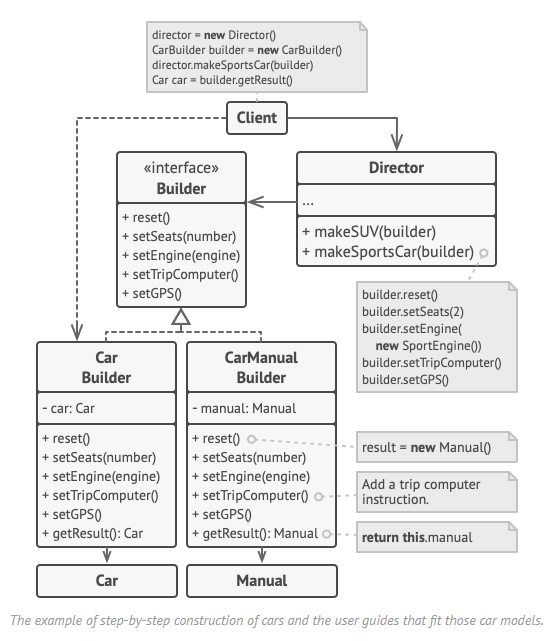


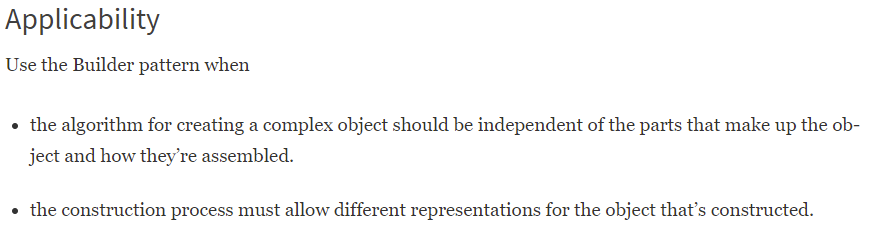


<https://refactoring.guru/design-patterns/template-method>

# 8) Builder Method

* Creational pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code. For most of the time, the structure will be the same, but the details implementation will be slightly different. This could be useful when construction a product or object that contain multiple similar information type (such as building the car, writing the car manual and develop the car information). When building another car type, same process applies.

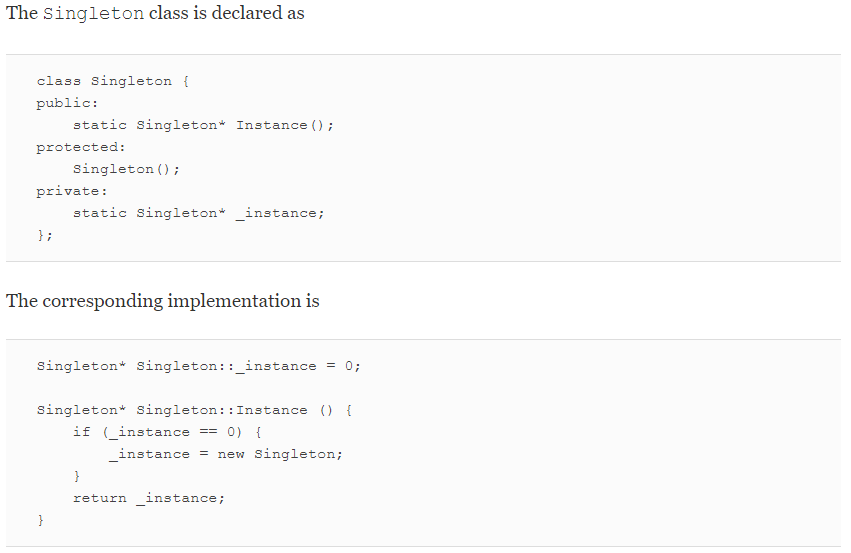


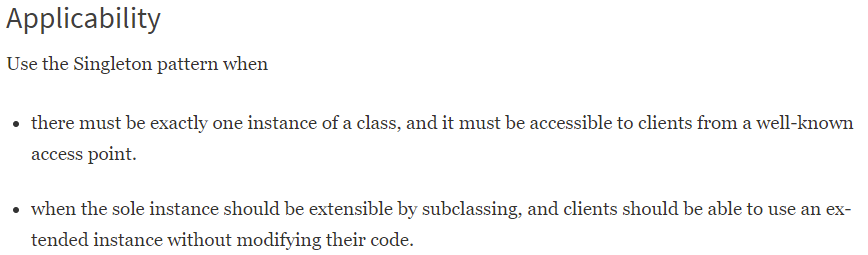


<https://refactoring.guru/design-patterns/builder>

# 9) Singleton Method

* Ensure a class only has 1 instance and no other interface, class can’t manipulate it and it can be access globally via an instance class. Put the default constructor into protected to avoid multiple initializations and only provide 1 access point via Instance() method. What will happen here is, instead of calling constructor to initialize a new object, calling Instance() method will “reset” the old object

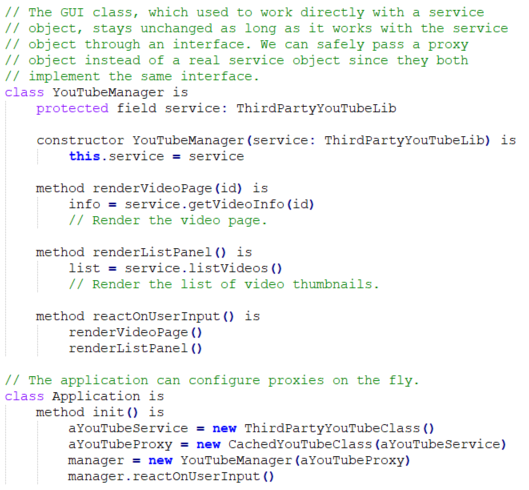
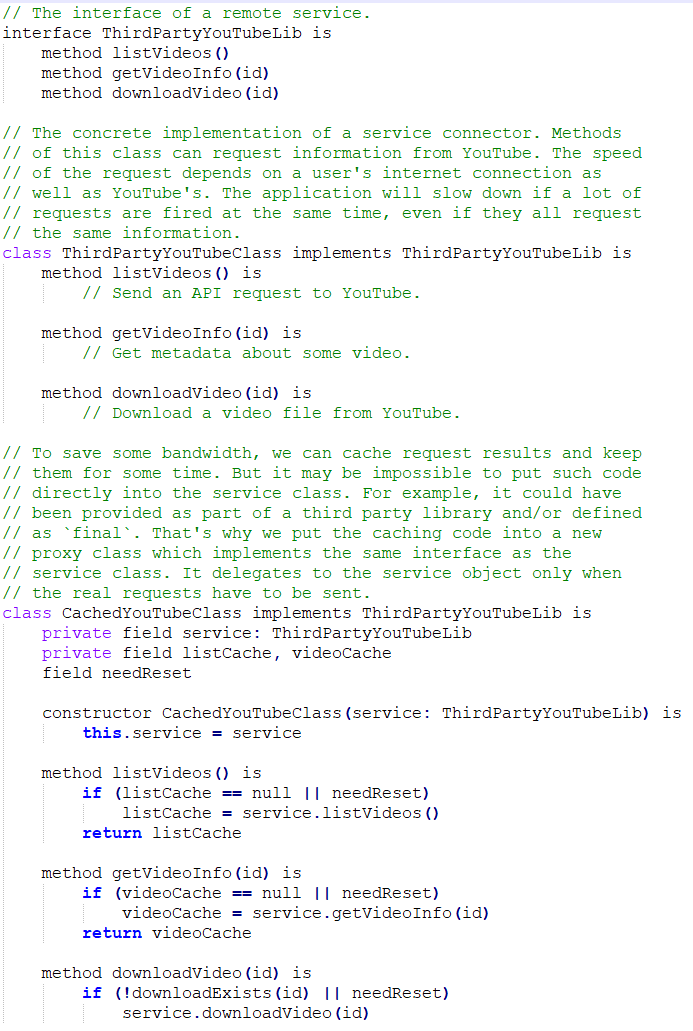


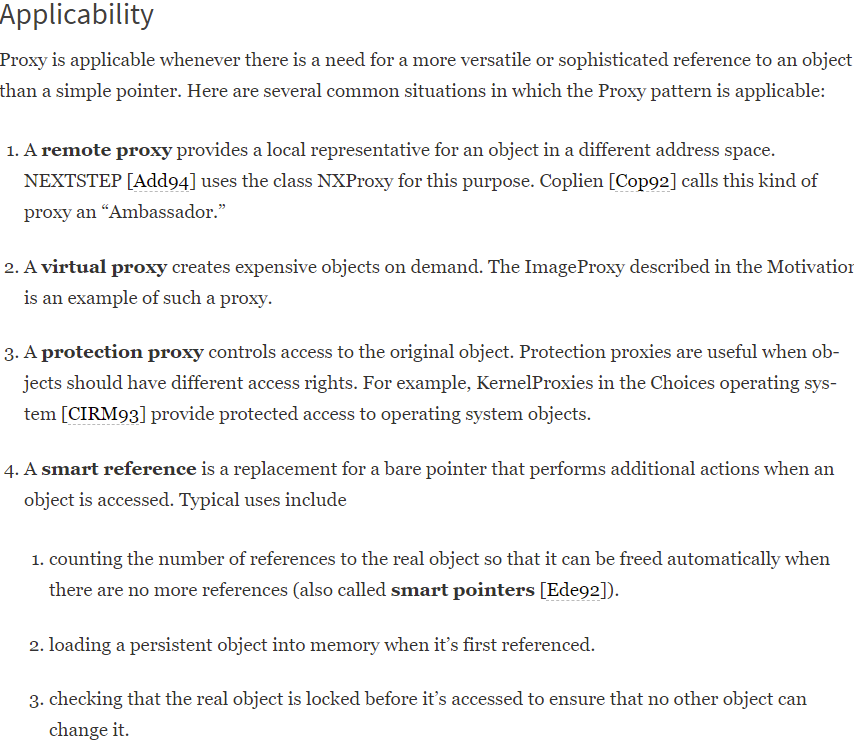


<https://refactoring.guru/design-patterns/singleton>

# 10) Proxy Method

* Provide a substitute or placeholder for an object when possible, reduce the number of times client needs to access to the real object. Another purpose of doing this is to reduce the number of times the client needs to load the object which sometimes client doesn’t use them all. Imagine Google Drive, where on the first time it takes a while to load a photo but every concurrent time, it will appear instantly. It is because the photo has generated a cached and allow client to access the cached first before getting to the photo, as if client get to the photo, it will have to download the image into user’s local cache machine which take some time. If cached is not available, direct request will be send to the real object.

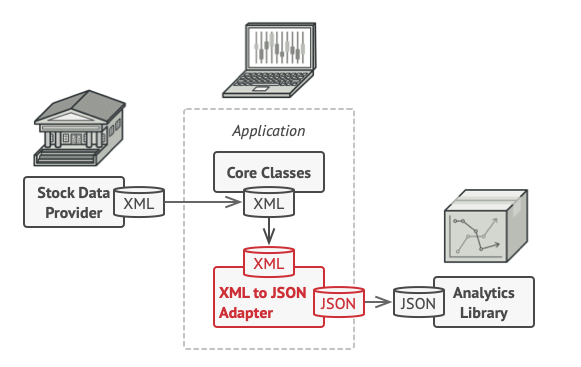


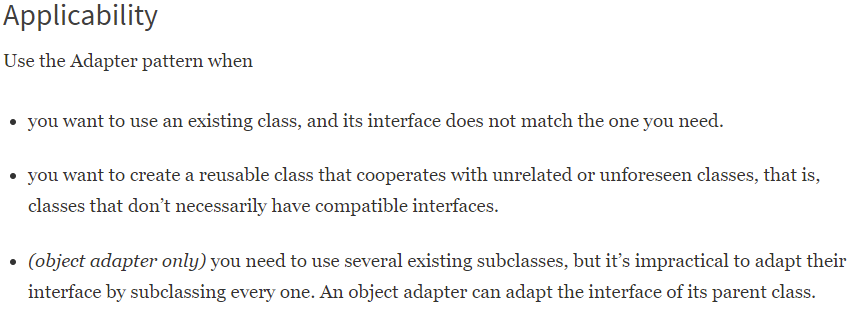


<https://refactoring.guru/design-patterns/proxy>

# 11) Adapter Method

* Allow an object with incompatible interface to work with another completed interface. Basically, there is a Class Adapter acting as an intermediate class to do the conversion. Imagine 2 class has their own implementation, and class B want to perform the logical way that class A is doing, but class B doesn't want to do the entire logic coding process again. Using Adapter can make use of implementation in Class B into Class A.

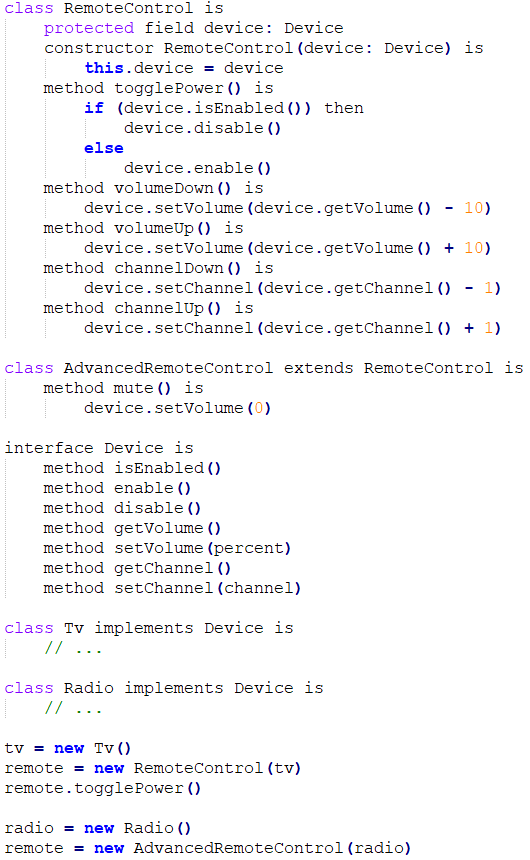


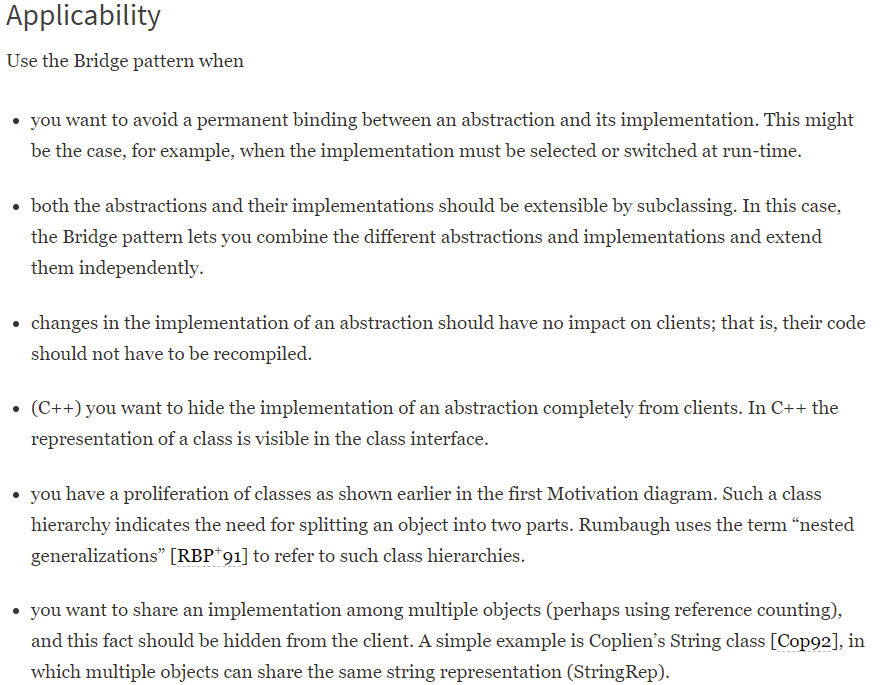


<https://refactoring.guru/design-patterns/adapter>

# 12) Bridge Method

* Split a large structural class into a set of closely related classes (usually 2 or 3) which can independently develop and provide abstraction/implementation to each other's. The main benefit on this is, when changing structure of one class, we can avoid the changing for other class. Easiest to understand is imagine a structure class ***Shape*** as 2 subclass ***Square*** and ***Circle***. If we want each of them to has a color, we will have to extend the class ***Square*** and ***Circle*** further down, which lead to multiple subclass. However, if we bridge them together, let say class Color in class Shape, then whenever we create an object, we can use the constructor in class Color to specific the Color.

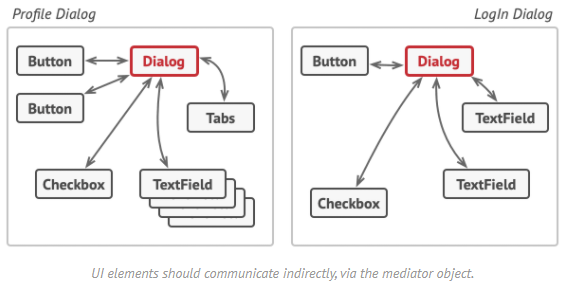


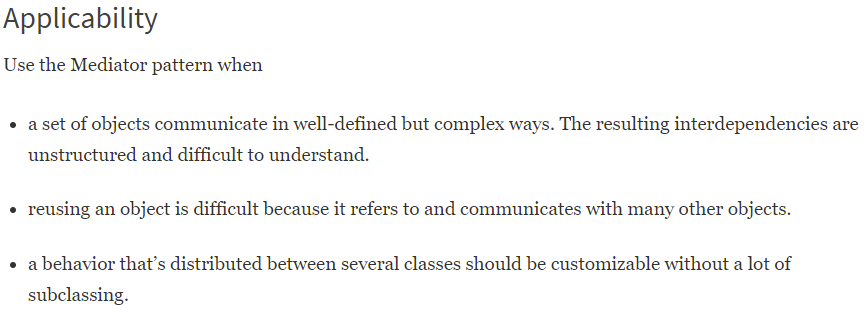


<https://refactoring.guru/design-patterns/bridge>

# 13) Mediator Method

* Object to Mediator and Mediator to object communication. Mediator object knows about every other object, but the other object only needs to know about the Mediator. This avoids the hassle of multiple class communication can lead to confusion and unnecessary connection. Doing this can allow “traffic” to go where they want to go and avoid the object. Assume Mediator as a window with multiple selection option. Whenever an option is clicked, the box will send the request to the appropriate function and fetch the respond from it.

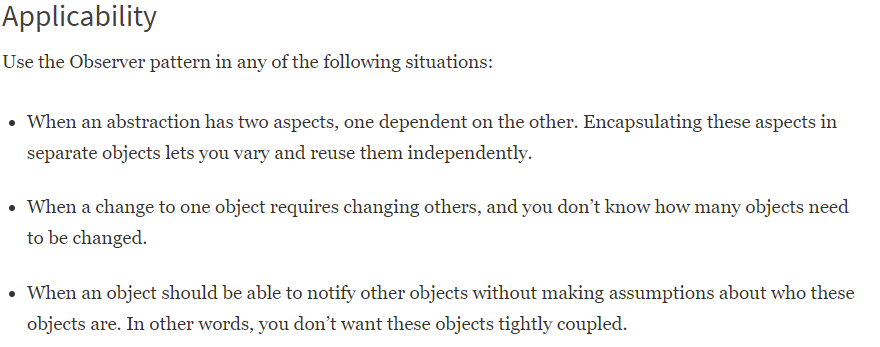
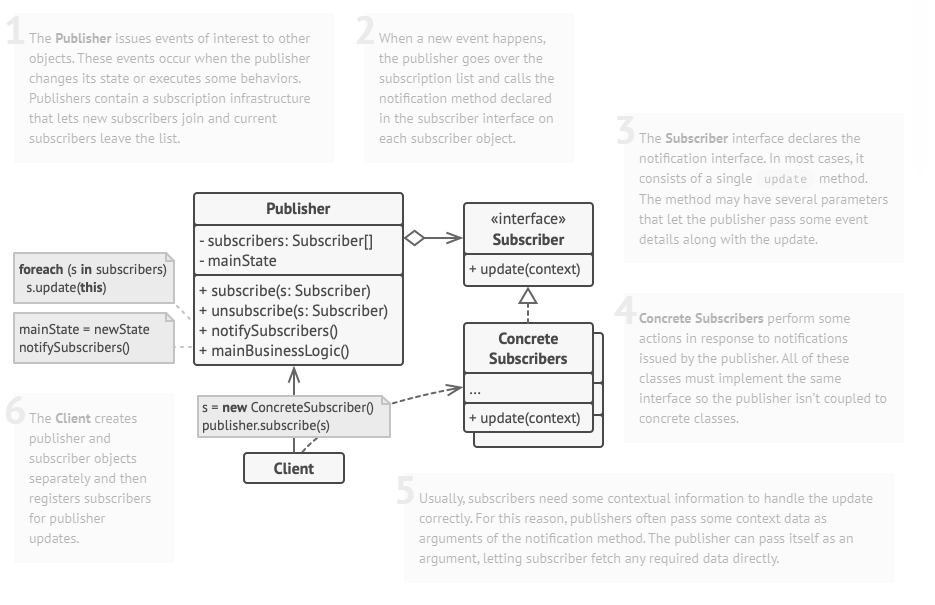




<https://refactoring.guru/design-patterns/mediator>

# 14) Observer Method

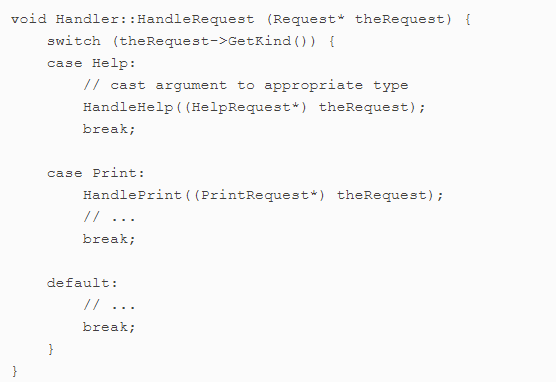
* Pretty much like Mediator, where an object registers to an Observer object to be notified about a change of a certain object. If an object status is changed or something going on, it will notify all the subscriber about a change. What different this from Mediator is that Mediator is a 2 ways connection between object, while this is just between the Observer and the object.

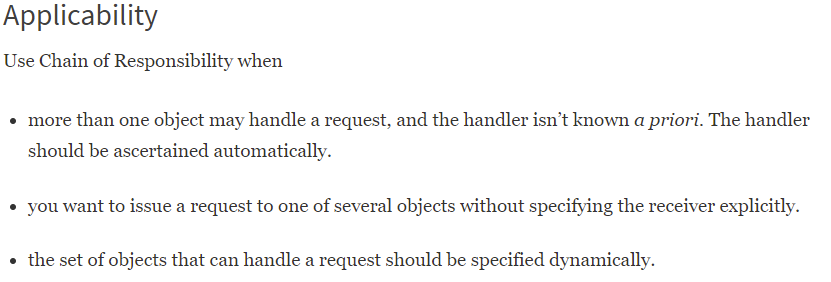


<https://refactoring.guru/design-patterns/observer>

# 15) Chain of Responsibility Method

* Pass a request along a chain of handlers and let each of the handlers decide whether it will take this request or will pass to the next appropriate one. Ensure that when using this, there must be a handler able to receive this, otherwise it doesn't make sense.

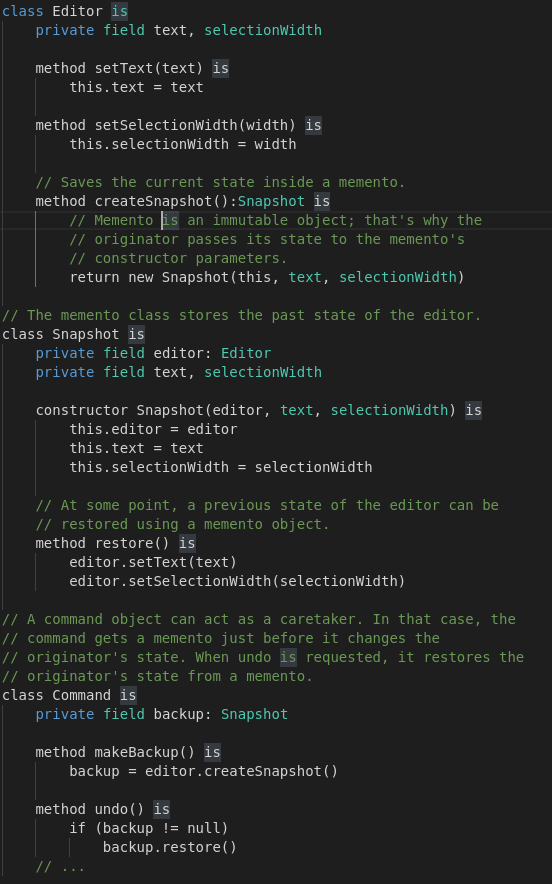


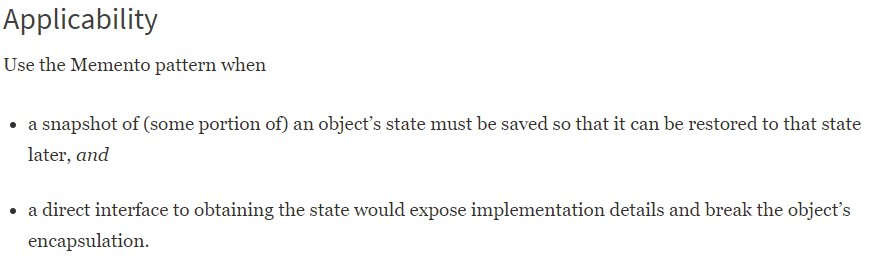


<https://refactoring.guru/design-patterns/chain-of-responsibility>

# 16) Memento Method

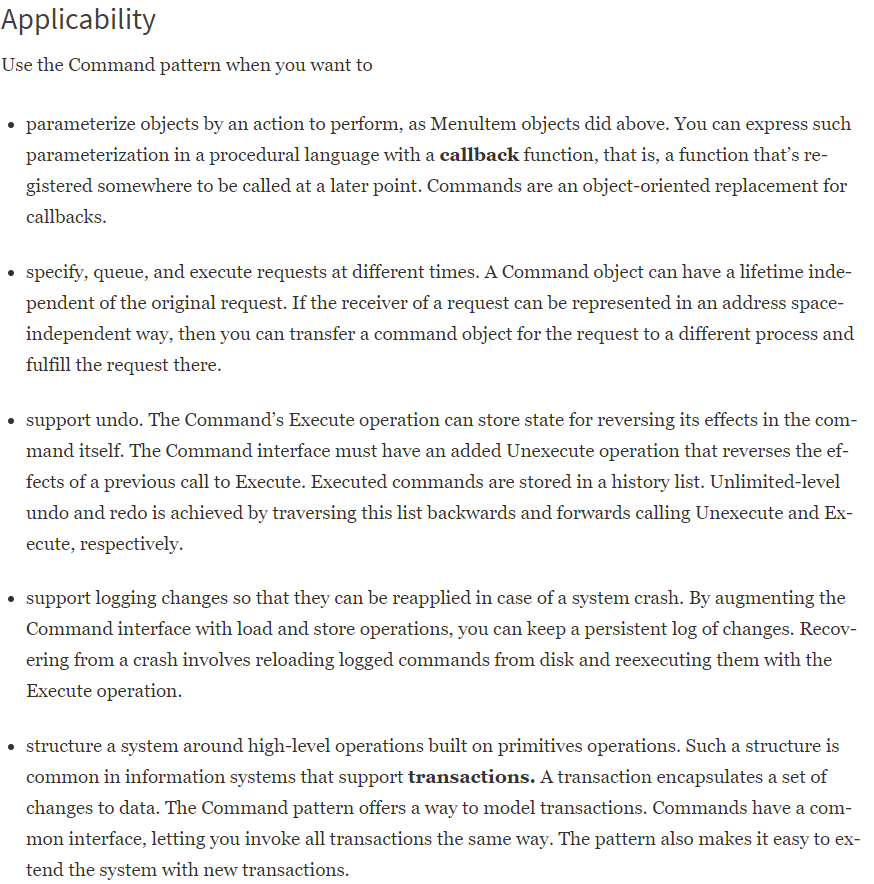
* Create a save point to restore the data state in previous action. Like what undo will do. User will not know the details of implementation or data types, but they will know the content for each data types and expecting that.





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

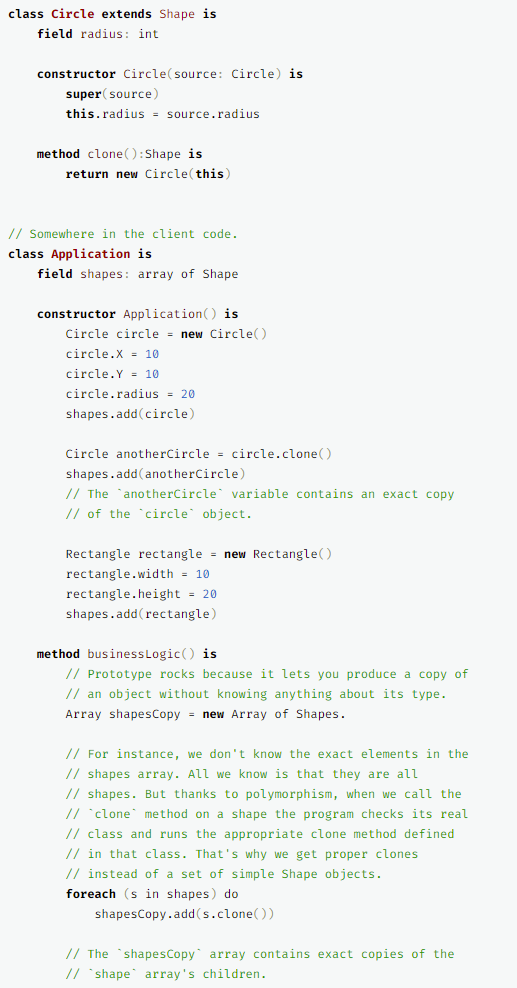
# 17) Command Method

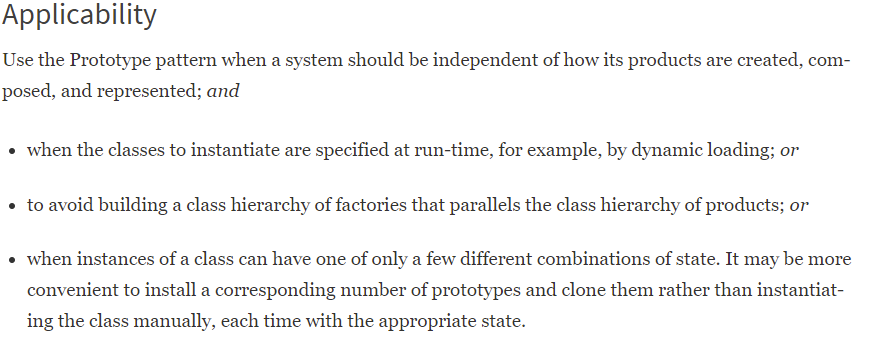


<https://refactoring.guru/design-patterns/command>

# 18) Prototype Method

* Duplicate an existing object without depending on the class structure. In the concrete class, a clone method is required to clone an existing object (with data), but it is unknown to the client about its structure and existence. When a class interface is called, it will call the appropriate object according to what is on the data. Let’s say there is a Shape interface with concrete class of Circle and Rectangle. Shape array can contain Circle and Rectangle, but the user doesn’t know the detail about the array, assume it is only an array of Shape. When looping through the entire array, the Shape will know the identity of each one inside and call clone method in the appropriate concrete class.

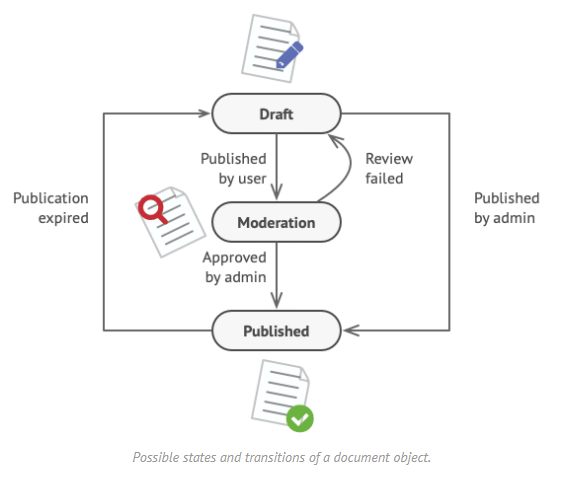


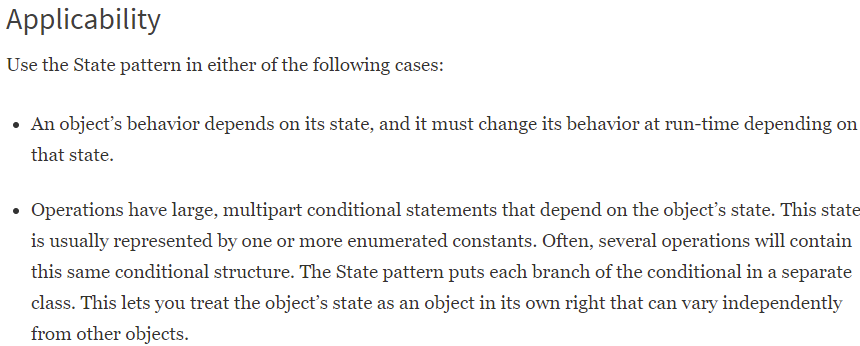


<https://refactoring.guru/design-patterns/prototype>

# 19) State Method

* Change an object behavior when internal state change. For example, in production, there are multiple stage of approved that a “work” must pass in order to be released. Considering the “work” as an object, where it will need to pass the testing stage, QA stage, documentation stage and many other. How does an object know which stage to go? Simply alter the behavior of the “work” object in each of the stage, like changing the flag to allow the object to change its state and move to another function. (Check the code in the link)

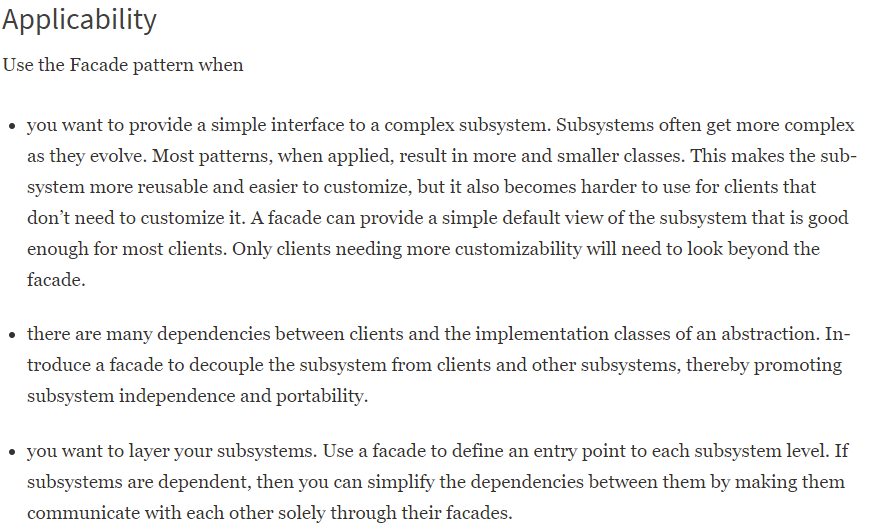


<https://refactoring.guru/design-patterns/state>

# 20) Façade Method

* Fake covering. Showing the entire project or operation as something else, which only reveal the most important information and functions.



<https://refactoring.guru/design-patterns/facade>

**Review Question**

1. **With respect to the ClassAdapter, GOF states that a class adapter "adapts Adaptee to Target by committing to a concrete Adaptee class". Explain what this means with respect to your implementation. What would you need to do to adapt the other linked list instead? Is there a way to configure the ClassAdapter at run time to choose the other linked list?**

* ClassAdapter use interface from Target lass to make it “compatible” with the Adaptee class. By calling the appropriate Adaptee class in each of the Target function, we can somehow convert the incompatibilities. When the Target want to adapt another LinkedList, simply replace the class name to a new one. If the other class is also a concrete class of the same class, function calling might stay the same.

1. **With respect to the ObjectAdapter, GOF states that an object adapter "lets a single adapter work with many adaptees - that is the Adaptee itself and all its subclasses". Does this apply to your current implementation of the ObjectAdapter? If yes, explain how this applies to your implementation and how you can use the same adapter to adapt the other linked list? If no, what changes would you need to make so that the adapter can be configured at run time to adapt the other linked list instead.**

* Right now, it is. The ObjectAdapter class contain an Adaptee of DList object which will call the appropriate function in DList to current Target class. Simply replace DList with another Adaptee class and make the change in function calling

1. **Why is it important not to deallocate the Observer when detaching from this application? Does this apply in general? Why or Why not?**

* When an Observer is detaching from the application, it doesn’t mean the Observer is no longer wanting to be notified from the application. There might be a chance that Observer function are temporally detached to avoid any unwanted notification that it can manually update and synchronize itself later. Deallocate the Observer could remove the Observer entirely from the application which could cause some trouble to it when trying to re-attach or synchronize itself.

1. **GOF discusses how the Memento has a wide/narrow interface. Explain what is meant by this and how your implementation of the Memento pattern accomplishes this.**
2. **Does your Memento store the same information as the Timer? Explain why each value stored was necessary for the restoration of the Timer to its former state.**
3. **Does our Timer being a Singleton prevent other Subjects/Observers from being created? Explain your answer.**

* Yes. Right now, the default constructor has been moved into protected state and regards to Singleton definition, a method of Instance has been created to return the instance of Timer.

Template

1. **Try using your own words to explain the purpose of using the Template method?**

* Create a general Base class (with general implemented function) that other class can derived and override the method on the Base class if necessary, to perform a complete work. Some of the function may come with an implementation that provide similarities to all concrete class.

1. **Following this, according to the GOF book, there are three applicability. Choose two and explain them using your own words once again.**

* *When several classes have similarities, with minor differences, something should be done*: Template can have pre-defined and implemented functions that all concrete classes to use.
* *When there are pre-defined steps to create an algorithm*: Template provide a class with general functions/methods that other class can derive and follow its structure and algorithms.

Mediator

1. **Describe the Mediator Pattern.**

* Mediator object knows about every other object, but the other object only needs to know about the Mediator. This avoids the hassle of multiple class communication can lead to confusion and unnecessary connection. Doing this can allow “traffic” to go where they want to go and avoid the object.

1. **What are its uses and mention briefly about 3 uses?**

* Use Mediator when there is a communication existing between multiple objects to each other's
* Avoid reusing class in other object which is difficult and sometimes easy to make error.
* Behaviors that distributed between many classes should be customizable without many subclassing (GOF)

1. **How you would implement the pattern. Also describe 2 disadvantages of the pattern**

* Centralize control which cause a communication delay between objects. If there are too many objects that requires communication, even Mediator object can be confuse itself and hard to maintain.

1. **How would you remedy these disadvantages if possible?**

* Only use Mediator when there are less than 10 objects for best communication. If there are more than 10, we can categorize them and split them into 2 group and let 2 Mediator working together to transfer the connection. For example, if Class 1 want to access Class 10, it will have to go through Class 1’s group Mediator, then that Mediator class will communicate with the Mediator class in Class 10’s group to transfer the request and response.

State

1. **Describe at least three advantages the State Pattern has over the conventional method of putting all state-specific behavior into one class by using internal data values and conditionals.**

* Alter the behavior of the object when accessing a step and change its state internally.
* Allow creation of new subclasses to update its state in case of object changes.
* Remove the use of the “if else” or any related conditional logic which can cause confusion and hard to keep track.

Builder

1. **Describe what a builder pattern is?**

* Creational pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code. For most of the time, the structure will be the same, but the details implementation will be slightly different.

1. **What are the advantages of this pattern?**

* The process of constructing new Object must allow different form or object representations
* Algorithms to construct each object should be independent. What the Builder provide is an interface and a step-by-step guide how to build the object.

1. **Where do we use this pattern?**

* One of the examples I can think of is using when building multiple class that share the same structure, such as building the car, where there are many processes that share the same structure, such as manufacturing the car, writing owner’s manual, and implementing the car detail into the car system. All of those required the same structure, pretty much the same algorithms, just slightly different in each of the step.

1. **Name a pattern that is like the builder pattern.**

* Factory Method could be the one

1. **Describe their similarities and differences**

* Similarities is both patterns have a main class that provide an interface and their concrete class that represents their own structure and algorithms
* Differences would be Builder pattern describe the creational of the object in step, whereas Factory pattern is about families of objects products.

1. **When a builder pattern would be more applicable.**

* Builder would be applicable when creating multiple complex object and each of them has its own representation to modify different part at the time.

Memento

1. **Explain the roles of the Originator, the Memento, and the Caretaker in the Memento pattern.**

* **Originator (constrain solver):** create a memento object to store current data and use it to restore the state.
* **Memento (solver state):** store the Originator object (could be full store or partial store). Caretaker can only pass the Memento to other objects, known as the narrow interface. Originator can access the data to restore itself to the previous state, known as the wide interface.
* **Caretaker (undo mechanism):** keep the memento data safe, only use to store the data of memento, doesn’t modify or examinate

1. **Give an example to demonstrate how using the Memento pattern helps preserve the object's encapsulation boundary.**

* When the restore is triggered, the Memento object will take a snapshot of the Originator’s data and store it there. When the undo is triggered, the Caretaker will ask the Memento for the restore state of Originator object at the checkpoint. Memento will return the Originator object containing data from the earliest save point, replace with what Originator is having.

1. **Explain the drawbacks of using the Memento pattern, what type of problem it can solve, and give a simple example of this pattern using code, and/or a diagram that is clearly labelled with the participants/roles.**

* The problem it can solve is allow the program to go back to its previous state if there is something wrong happen. This is becoming a standard in most of the applications as people are getting used to the undo option. The drawbacks of doing this is it will delay the application by a bit if the problem current state has too much data/information and it might has a limitation on how many undo it can store.

Composite

1. **Explain how the Composite Pattern works?**

* Treat individual objects or group of objects uniformly. Although Composite is about handling multiple objects under it, it is still treating them as individual objects. Imagine Composite as a Tree, where there would be some LeafNode, but it could contain another Composite under that, where it is also treated as an individual object just like LeafNode.

1. **When would you use this pattern?**

* When treating similar type object together, such as Shape as a Composite, under it we can have: Round, Square, Rectangle, Triangle. Under Rectangle we have Regular Rectangle, Parallelogram, Rhombus, … Under Triangle we have Square Triangle, Equilateral Triangle, Regular Triangle.

1. **Provide an example on when you would use the Composite Pattern.**

* Calculating total value of a box item. Inside the box, there are many items but there is also a box containing other items.

Iterator

1. **Describe what an Iterator is?**

* Iterators allow clients to traverse the list of data forward or backward. It shouldn’t expose the object detail in the list, only provide what is that object is about. For example, there is a list of Car type, it will only display to client the car type such as Sedan, SUV, Van, but it doesn’t provide the detail such as which car or car details.

1. **What are 2 problems that an Iterator pattern can solve**

* Iterator can help client to access an aggregate object and provide traversing forward/backward.
* Provide interface to traverse multiple aggregate objects without exposing the detail inside.

Observer

1. **The Notify function is used in the Observer pattern to ensure that a subject and its observers maintain consistent states. The Gang of Four textbook suggests two options for choosing which object should call this function. List both options and describe their advantages and disadvantages.**

* **Subject**: know the observers and its number. Allow Observer to attach itself to be notified or detach itself when no longer needed. When there is a change, the Subject will notify the Observer. This can sometimes annoying because it will notify all the observers when there is even a small change. When Observer no longer want to receive notification automatically, it can detach from Subject for manual operation to avoid any unnecessary notification.
* **Observer**: create an interface for objects to be notified when there is a change in subject. A good thing about this is Observer can manually check the Subject for any new change to synchronize its state with the Subject when the Subject is performing a task that is not relevant to the Observer. However, by doing this can sometimes cause Observer to miss important changes, especially when Observer forgot to call Subject for checking.

Proxy

1. **Explain the different situations where you would apply a proxy pattern?**

* Using proxy pattern to provide substitute for an object when that object is being accessed a lot by another object. Doing this will avoid direct communication and chance of screwing over the actual object and to provide better speed (without must load the object from the beginning to the end, especially when it is a big object). For example, when playing a video on YouTube for the first time, there will be a cache of that video store in your machine. What will happen is, next time, when you are accessing the same video, it will be playing from the Cache, not directly from the video.

1. **What are the participants in a proxy pattern and explain what each participant does?**

* **Proxy (ImageProxy):** provides an interface similar to Subject to represent the RealSubject by taking references from RealSubject. If Subject and RealSubject are the same, it will just take the interface of Subject for simplicity. Proxy might be able to modify the RealSubject if necessary, to update the new data over.
* **Subject (Graphic):** defines the common interface for RealSubject and Proxy so that the placeholder (which is the Proxy) can be use whenever RealSubject needs
* **RealSubject (Image):** defines the real object that proxy is representing.

1. **What is the kind of proxy and explain?**

* **Remote Proxies:** encoding a request and its arguments and sending the encoded request to RealSubject.
* **Virtual Proxies:** cache additional information about RealSubject to avoid any unnecessary extra accessing to the object.
* **Protection Proxies:** check if the caller function has the access permissions before requesting.

Adapter

1. **There are two main types of adapters. What are the two types of adapter?**

* There is Class Adapter and Object Adapter.

1. **What is the main difference between their composition?**

* Class Adapter provide a “compatible” between 2 difference classes but might share the same theory and structure. There is 1 class contain an interface and 1 class contain the structure implementation. When doing this, the function in the interface class will be accessing the functions with the implementation. With this Class, the implemented function will be directly call using “Class::Function()”
* Object Adapter is pretty much like Class Adapter, with an exception that there will be an object and implemented function will be calling through that object.