**Composition vs Aggregation**

**Composition**: A Car object containing four Wheel objects. Normally if you destroy the car (by calling some method to clean it up) you should also destroy the wheels in the process, since it makes little sense for them to exist outside the car (unless you move them to another Car object). More realistically, a reader object wrapping an input stream will also close the inner stream when it gets closed itself. This is composition.

**Aggregation**: A Person object contains (owns) a Radio object. If the Person dies, the Radio may be inherited by another Person i.e. it makes sense for it to exist without the original owner. More realistically, a list holding objects does not demand that all objects get disposed when the list itself is disposed. This is aggregation.

Aggregation: object A contains objects B; B can live without A. Composition: object A consists of objects B; A manages life cycle of B; B can’t live without A.

**Design patterns**

Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

Creational patterns: These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code. (Factory, Singleton, Builder)

Structural patterns: These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient (Bridge, facade, decorator,proxy, adapter).

Behavioral patterns: These patterns are concerned with algorithms and the assignment of responsibilities between objects. (strategy, observer, mediator, command, iterator)

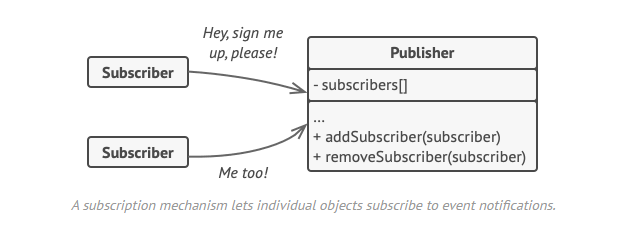
**Strategy** is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable. The strategy pattern uses composition instead of inheritance. In the strategy pattern, behaviors are defined as separate interfaces and specific classes that implement these interfaces. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes.

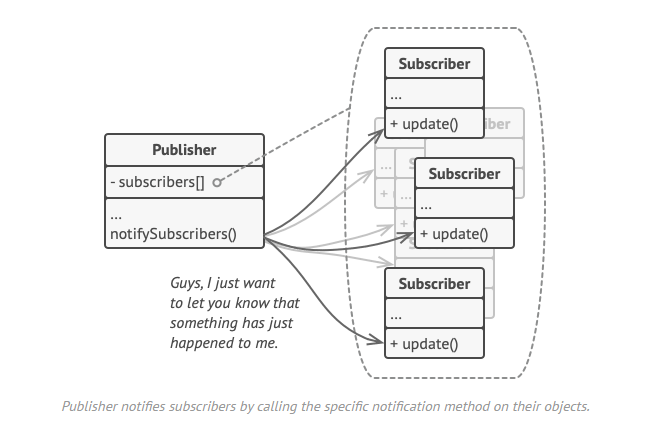
The strategy pattern (also known as the policy pattern) is a software design pattern that enables an algorithm’s behavior to be selected at runtime. The strategy pattern defines a family of algorithms, encapsulates each algorithm, and makes the algorithms interchangeable within that family.

One of the cons of the strategy pattern is that the client should know how algorithms differ from one another.

**Observer** defines a one to many dependency between objects so that when the object (the subject) changes state all of the its dependencies are notified and updated automatically. So there are observers that wait for the state to change, and there is an observable that changes state and notifies its dependencies.

It uses pushing instead of polling (as in event polling). So, Observable will have some kind of register to add dependencies, unregister, and notify methods. Observable will have a collection of observers and all observers will have some kind of update method so that when observable changes state it notifies the observer by calling the update method.

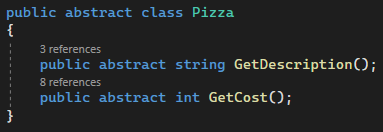




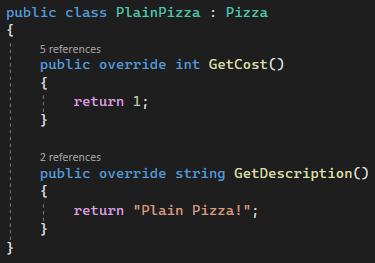
We also pass the observable object through observers’ constructor so that when observers get updated they can have access to the state of the observable.

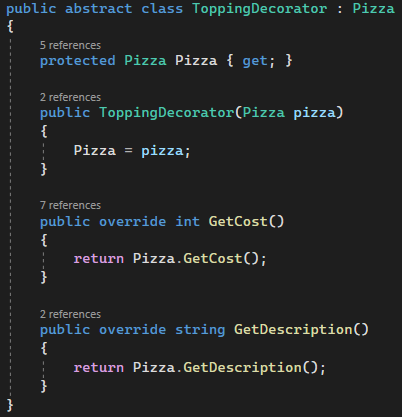
**Decorator** is a structural design pattern that lets you attach new behaviors or responsibilities to objects by placing these objects inside special wrapper objects (aka decorators) that contains the behaviors. Decorator has a component and is itself a component because it can also have a decorator outside itself. So Decorator class will have a property of type another Decorator.

Decorator pattern allows us to modify an object dynamically at run time. We would use it when we need the capabilities of inheritance with subclasses, but we need to add functionality at run time.

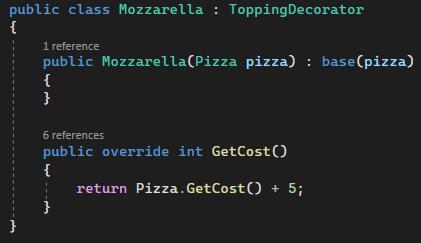


The very inside object

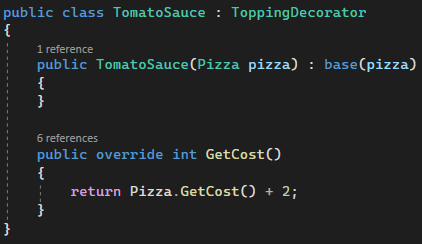




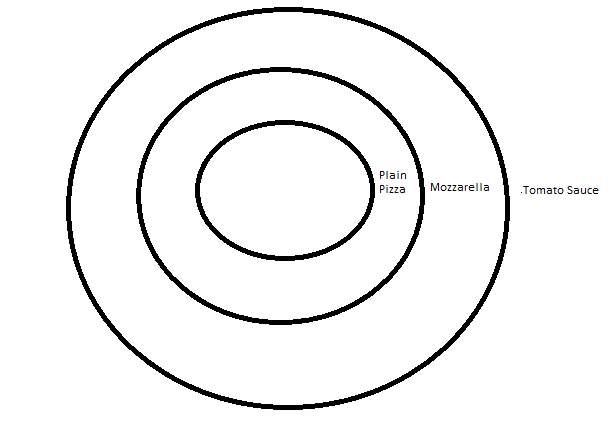
Decorator 1



Decorator 2

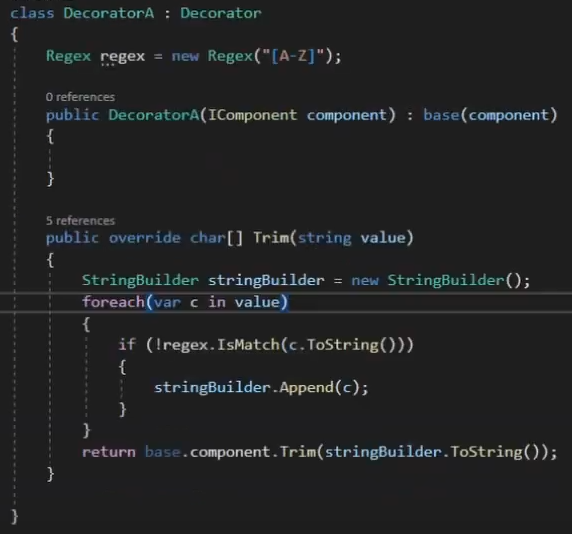






Plain Pizza is a component. In the constructor, we are getting the Pizza (object) that we want to decorate.

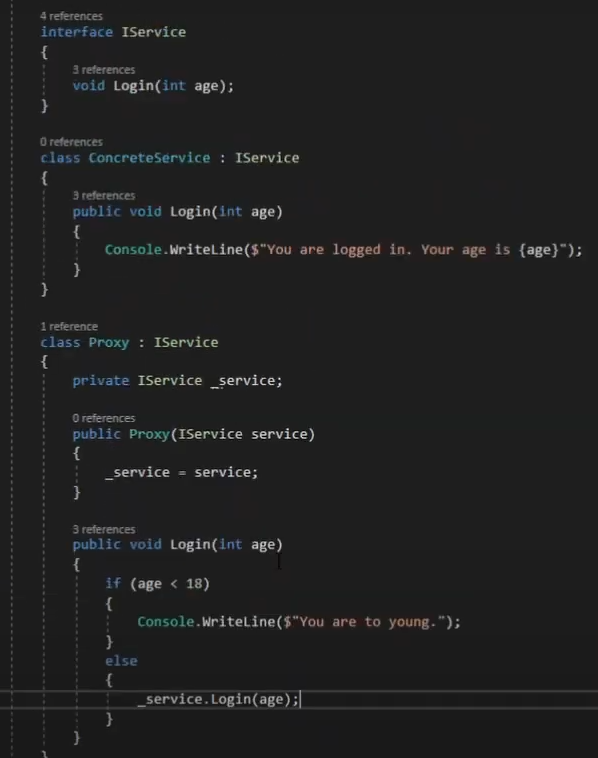
Another example of the decorator pattern:



We do so new operations before the Base trim and then we define our decorator and components from initializaiton since we need to be careful what we are constructing 🡺



**Proxy pattern** :



There is one service and two implementors: real implementor and proxy which has the real implementor (ConcreteService) injected into itself thourgh the constructor. We usually use proxy when we need to do something before the real implementation.

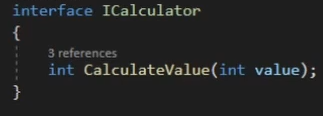


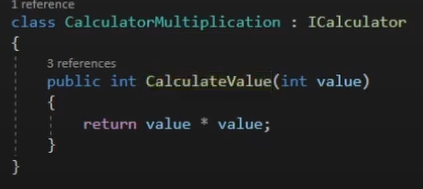


The intent of the proxy-pattern is to provide a placeholder for another object to control access to it.

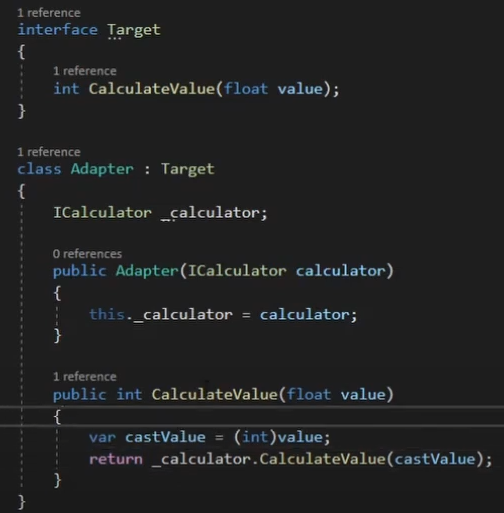
* **A remote proxy is responsible for encoding a request and its arguments and for sending (and retrieving) the request (and the response) to the real object.**
* **A virtual proxy may cache additional information about the real subject so that it can postpone the access to it.**
* **A protection proxy checks whether the caller has sufficient access permissions for perform a request.**

**Adapter**: Let’s say that we have an interface with a CalculateValue interface that takes an int as an argument. For testing, here we need to adapt this int argument since we will be passing float instead.





We will create another interface that takes a float as a parameter and inside that method we make the conversion after which a call to the real implementation is made.



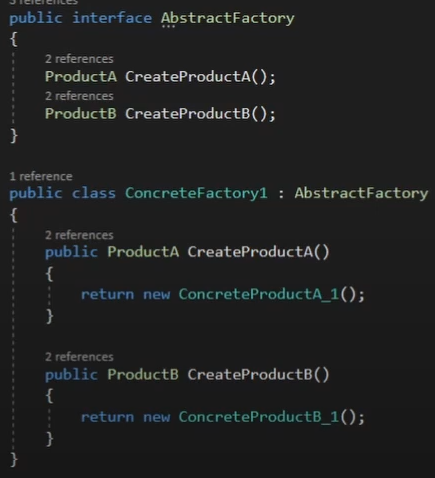
As you can see it also takes the real implementor through its constructor.

**Bridge:** The Bridge pattern is used to decouple an abstraction from its implementation so that the two can swap up implementations at runtime (IVideoCameraAccessor).

**Builder** is a creational design pattern, which allows constructing complex objects step by step.

**Abstract Factory** design pattern is one of the Creational pattern.

* The main difference between a “factory method” and an “abstract factory” is that the factory method is a single method, and an abstract factory is an object.
* The factory method is just a method, it can be overridden in a subclass, whereas the abstract factory is an object that has multiple factory methods on it.
* The Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

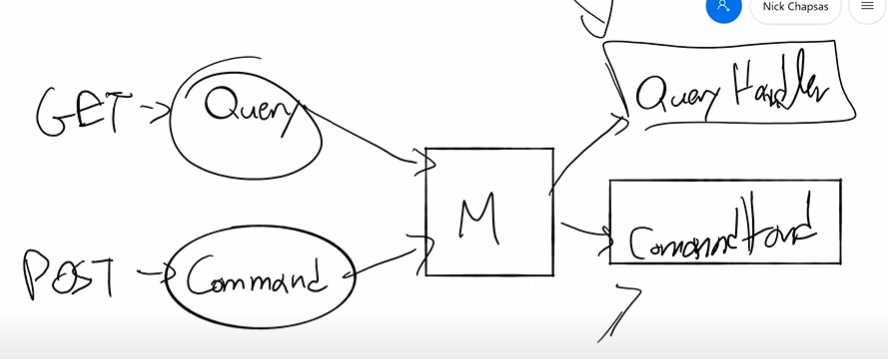


Factory Method is used to create one product only but Abstract Factory is about creating families of related or dependent products.

**CQRS pattern**

CQRS stands for Command and Query Responsibility Segregation, a pattern that separates reads and writes (update) operations for a data store.

Let’s say that we have a controller with different endpoints. The endpoints that just get something without changing the data we call query, and the endpoints that essentially post/update data we call commands. Then we separate these two concerns from each other. By doing this, for queries we can have a read-only connection string.



So when we catch Get or Post requests corresponding request is sent to the mediator and the mediator then forwards the request to the corresponding handler. Handlers are separated from controllers so we get separation of concerns.