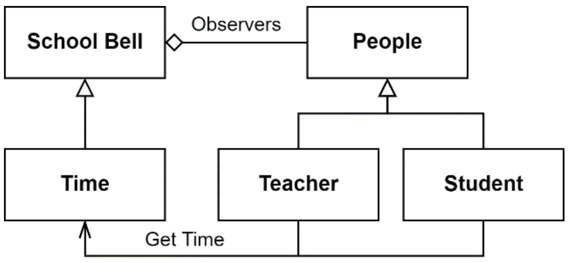
The Observer Design Pattern

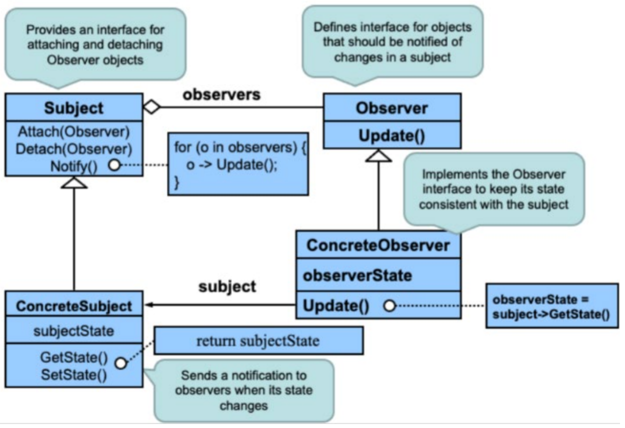
Real word example and relation to design pattern

In high schools, the school bell rings when a certain time interval ends. The bell ring notifies everyone in the school that a period of time ends. People will look at the time data and react differently based on their schedules. Teachers start or end classes, and students go in or leave their classrooms, gym fields, dining halls, etc. After notification, if the time indicates a leave school time / off-work time for teachers, people leave the school.

Here, the school bell acts as the subject class and contains the time information. People in the school are observers that can be notified by the school bell. People attach to the subject class by entering the school zone (building, gym, where there is a school bell) and detach by leaving the school zone. When the bell rings, it notifies everyone in the school zone that the time change reached a specific value. People will look at the clock to get the time and perform their own unique actions. The notification sound bell creates is the *update()* of observers in the design pattern. People accessing time via their own clock or bell's clock is the *getState()* of *ConcreateSubject* described in course notes.



Real World Example with Observer Design Pattern


Provides an interface for attaching and detaching Observer objects
Defines interface for objects that should be notified of changes in a subject
Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
for (o in observers) {
o -> Update();
}
Implements the Observer interface to keep its state consistent with the subject
ConcreteObserver
observerState
Update()
subject
observerState = subject->GetState()
ConcreteSubject
subjectState
GetState()
SetState()
return subjectState
Sends a notification to observers when its state changes

Observer Design Pattern from Course Notes

Beneficial to the system and how it addresses coupling

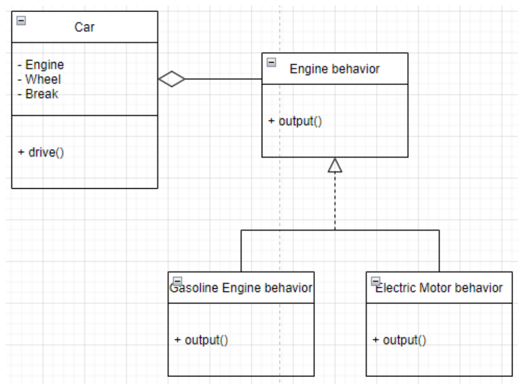
In school, people have different schedules, but people's time slots for the schedules are the same. This means people share the same time event to activate a scheduled event. To make sure nobody is losing track of time or not noticing that a certain time has reached, setting timers are a useful way to do. However, even phones and watches can access time, people need to actively look for it and check if a certain time is reached. A school bell can automatically notify everyone when some time is reached so that people won't need to frequently watch for time and worry that they may miss certain events. People are the observers, that wait for the bell's notification. This makes managing people in the school easier as a bell that notifies everyone can make sure people get the time reach message and when the time slot shifts with respect to the real-world time, only the bell ring time need to be changed without changing everyone's time schedule one by one. As only the bell change can make sure all people follow the time slot change, the coupling is achieved as people don't need to change anything to keep track of their schedule. People as observers can be easily add to the system when entering school zone without changing other implementations.

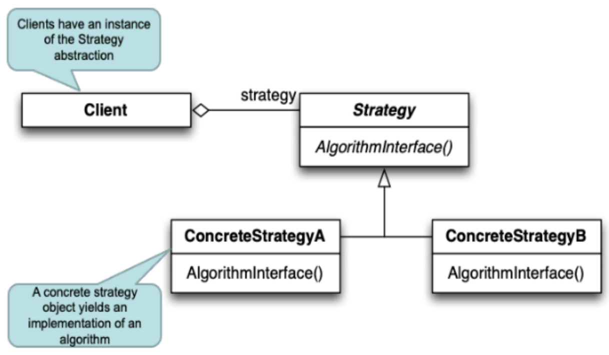
The Strategy Design Pattern

Real world example and relation to design pattern

A car has many parts, and those components work together. Some components can be changed for the same car model to have different functionality. Usually, a car model has various packages that offer a range of prices to customers. For example, the Honda Civic has Civic Sedan, Civic Hatchback, and Civic Si Sedan. Those cars are just slightly different in the power with the same looks. In my diagram, I just abstract this by using a different Engine. These cars should act the same, just with different engine output. When Honda builds these cars, they should keep the other components the same and switch to a different engine. They should keep the interface of engines the same, so cars don't need to be toned for different engines, and they can be driven based on the engine.

This car design can be abstract to the Strategy Design Pattern. The car is the client class. The car has an instance of an engine. The engine in this case is the abstract strategy class. There are two concrete engine classes which are gasoline engine and electric motor. A car can be driven as long as it contains an engine and gets the output from it. The car calls *drive()*, then *drive()* will access to *Engine.output()*. Different engines will give different outputs.




Clients have an instance of the Strategy abstraction
Client
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
A concrete strategy object yields an implementation of an algorithm

Beneficial to the system and how it addresses coupling

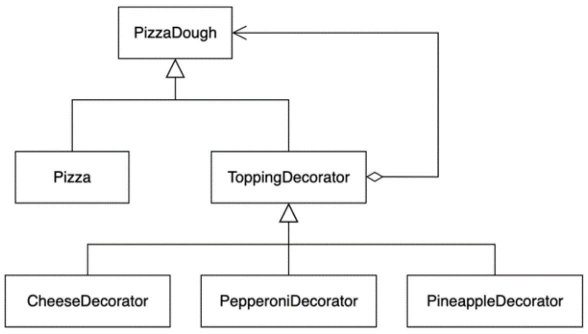
This design gives flexibility and generality to the car system. The car has an aggregation relationship with the abstract Strategy interface. So, it does not necessary to know which engine is going to be used in the design. The car system can still work when the engine is replaced. The coupling is largely reduced. Without this design, everything is embedded in the *car.drive()*, then a small change will require the car system to be adjusted. Next, the component of a car now has generality. Usually, these parts are built separately. In today's global market, they are probably built across countries. They may not necessarily know what exactly other parts will be, but they just know what they should do in a car system. In my example, I leave the wheel and break as the future change. They can also be the strategy classes as the engine. When a customer wants a sports vehicle, they can choose a set of bigger wheels and an engine with more horsepower.

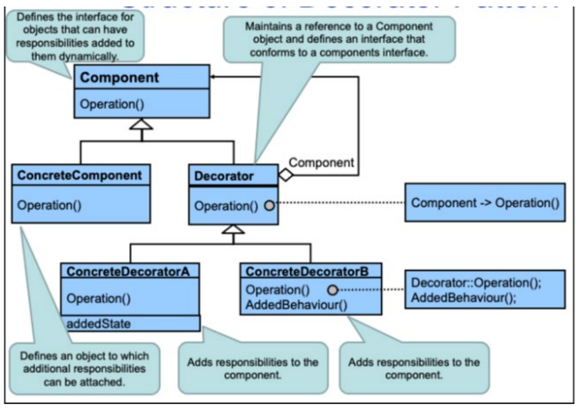
The Decorator Design Pattern

Real word example and relations to design pattern

In real life, we can buy pizzas from restaurants daily. Restaurants make various types of pizzas; you might wonder how cooks make pizzas. As all different kinds of pizza share the same instruction for making doughs, it would be much easier to make all doughs together first, then different toppings can be added on needs. For example, if the customer orders a Hawaiian pizza, the cook can just add cheese and pineapple to the pizza dough.

As described above, the pizza-making process shares the same idea as the decorator design pattern. Pizza doughs are the component, pizzas are the concrete components, and toppings like cheese, pepperoni, and pineapples are decorators. The cook adds a concrete topping decorator onto the component to make it concrete.




Defines the interface for objects that can have responsibilities added to them dynamically.
Maintains a reference to a Component object and defines an interface that conforms to a components interface.
Component
Operation()
Component
ConcreteComponent
Operation()
Decorator
Operation()
Component -> Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehaviour()
Decorator::Operation();
AddedBehaviour();
Defines an object to which additional responsibilities can be attached.
Adds responsibilities to the component.
Adds responsibilities to the component.

Beneficial to the system and how it addresses coupling

The benefits of the design pattern are timesaving and flexibility. For timesaving, all pizzas use the same pizza doughs, so the cook can prepare ahead of time when there are not many customers waiting. Therefore, when a customer orders a pizza, the cook does not need to start from making doughs. The cook can even freeze doughs so that they can store longer. For flexibility, the cook can decide what toppings he/she wanted to add to pizzas when the order comes. He/she does not need to plan and make a specific type of pizza ahead and worry about wasting it.

The design pattern helps address coupling because cooks can prepare different toppings separately and add them together later. Also, the lack of one type of topping does not affect making other pizzas that do not need it.