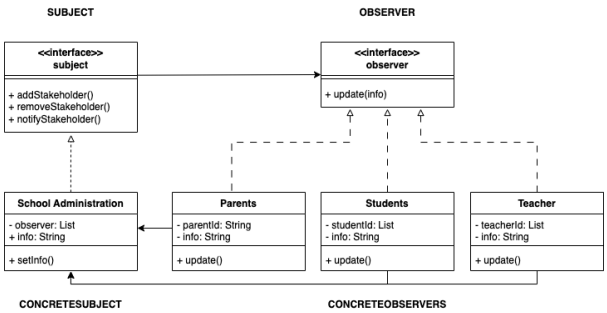
Observer

The Observer pattern, often referred to as the "publish-subscribe" model, is a behavioral design pattern used widely in software development. This design pattern emphasizes one-to-many dependency relationships between objects. In this design pattern, an object, known as the subject, notifies its dependents, also known as observers, about changes in its state which creates a strong dynamic interaction mechanism. This pattern fosters a decoupled relationship where the subject and its observers can operate independently, yet remain connected from a well-defined notification channel. This pattern encapsulates the behavior of observers and also promotes loose coupling between the subject and its observers. The subject does not need to know the concrete class of its observers and the observers can be easily added or removed without impacting the subject.

This design pattern is often hailed as the cornerstone of event-driven programming and is valued for its flexibility and reusability in software. The subject maintains a list of its observers and broadcasts notifications to them when its state changes. Observers, on the other hand, are objects that wish to be informed about changes in the subject's state, reacting with the expected behavior without directly polling the subject for updates. We can understand how the observer design pattern works by representing a school's notification system. In the context of a school, various objects such as students, parents and teachers (the observers) have a vested interest in receiving updates about events, schedule changes or emergencies (the subject's state change). In this case, the school administration is the subject. Anytime there's an update from the administration, the information is sent out to all registered observers via some communication channels (e.g. email, SMS, app notifications or PA system). This system benefits from the design pattern because number of stakeholders that need to be updated are unknown and changes dynamically. This pattern is useful in this case because the entity's update should be communicated via automatic notifications to its dependents to ensure consistency and timely updates across the system, which the stakeholders do by design by responding to events (notified by the administrators).



The figure on the right depicts how the objects in a school's notification system interact with each other. The school administration represents the subject, and the parents, students and teachers represent the observers. The direction of arrows show the flow of information through "connected pathways". This reduces coupling since the school administration isn't affected by the stakeholders, allowing for independent updates based on notification preferences i.e. parents may receive updates via phone, students via the PA systems and so on. Future changes enabled by this design pattern include being able to add new stakeholders without affecting the state. Additionally new notification methods can be added by just adding the logic to each concrete observer type, and no change to the business logic in the Subject would be needed.

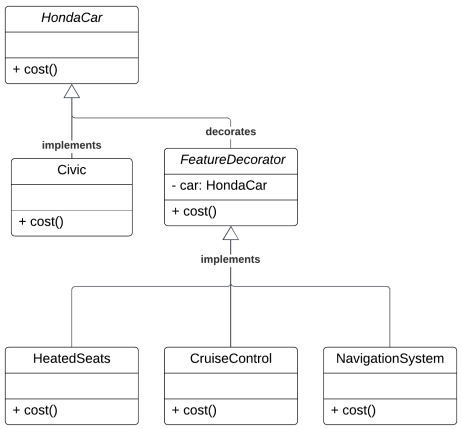
The observer pattern's implementation in a school's notification system shows its flexibility and reusability in various contexts, offering a way to design systems which prompte decoupling, scalability (to add new features and stakeholders) and also support efficient communication.

Decorator

The Decorator design pattern, famously known as the art of customization, is a structural model that lets you attach new behaviors by placing objects inside special wrapper objects that contain the behaviors. It favors composition over inheritance so that the user can compose objects in runtime instead of defining them through sub-classes, like in inheritance. When you need to add new features or behaviors to an object without modifying its class, the Decorator pattern is the go-to choice as you can extend functionality without altering the structure - this is one of its biggest advantages. This pattern involves creating a set of decorator classes that are used to wrap the original object. Each decorator adds its own functionality while still allowing the core object to function unchanged. Other benefits include enabling users to compose different decorators at runtime, which creates flexible and dynamic combinations of behaviors, and avoiding the need for creating complex sub-classes or inheritance hierarchies, which can lead to code duplication and maintenance issues.

A car's customization system is a real world example. Customers can customize cars online with optional features, book the car, and pick it up from a store. There can be many features that can be added - heated seats, upgraded audio system, autopilot feature, etc. Typically, to implement this, the core functionality must be modified to introduce logic for a new feature or multiple inheritance hierarchies will be required, thereby reducing readability and maintainability of the code. Instead, a Decorator pattern allows you to add multiple features to the base model of a car, so that the core functionality of buying a base model is never touched. Specifically, you can have an abstract class called *HondaCar* that Honda models should implement as a concrete class. In the diagram, we can see that one class called *Civic* implements the abstract class - this represents the base customization (no add-ons) for the Honda Civic. There also exists a *FeatureDecorator* class, which is an abstract class extended from *HondaCar* for features to implement. We can see three concrete classes in the diagram, each one being a mutually exclusive feature for any model of Honda. The method *cost()* in these features adds on the price of the feature in the form of recursion, which is how it achieves decoupling different modules. Now, features can be easily added in its class instantiation.

The components of a Decorator pattern are easier to maintain as all the code regarding a certain add-on such as *HeatedSeats* is placed in the same place. There is more flexibility as you can change any add-on without affecting other add-ons or the base model. Because of this, the pattern introduces less coupling since changes to any part of the structure don't result in changes to other parts. This also enables high cohesion as each additional feature of the main class can be a concrete decorator that wraps around the main class, which can be done recursively. In the future, new add-ons can be created and added as features without changing other parts of the code to accommodate the new car features.



Strategy

The Strategy design pattern is an architectural model that lets you define a family of algorithms and put them each into separate classes. Strategy lets the algorithm work independently from the context that uses it, the choice of algorithms are to be changed dynamically at runtime based on the situation. A real world application of the strategy design pattern is Google Maps, where multiple navigational strategies are offered to the user while the process of navigation remains consistent. For example, the user can get from point A to point B in many different ways, that could include different transportation means such as by car, walk, bus. The advantage is that it enables the dynamic substitution of routing behaviours based on the users request.

The strategy components for the navigation system in the real world application include three main components, the context class, interface strategy and strategy classes. In a navigation system like Google Maps, when you want to get directions, you have different choices for how you want to travel, by car, public transportation, or walk. These choices are the strategies for reaching your destination. Each mode of transportation (Car, Train, Walk) is programmed as its own 'strategy' class which means its a set of instructions for getting you where you want to go in a certain way. The interface, Route Strategy (as titled in the diagram) calculates the route depending on the chosen transportation method. The Strategy Interface is common to all concrete strategies, it declares a method the context uses to execute a strategy. The Context class serves as the user interface, establishing the criteria for route selection. The context does not know what type of strategy it works with or how the algorithms is executed. By doing this, the application can easily swap travel modes without needing to change most of the code base. Making it easier to add new travel modes in the future.

The Strategy design pattern reduces coupling in a system by separating the design making process of which algorithms to use from the actual execution of that behaviour. By encapsulating each algorithm within its own class (Strategies), we separate the internal workings from the consuming entities. The Context Class manages the strategies without resorting the complex conditional logic, resulting in more maintainable code. For instance, in the Google Maps example, each routing algorithm, whether for driving, public transport, or walking is encapsulated within its own 'Strategy' class. The 'Context' class, representing the user's navigation session, maintains a reference to a 'Strategy' interface, allowing the interchange of routing algorithms without altering the 'Context's' operation.

When a new feature is added to the system (for example a new transportation mode, plane), the context class would not need to change and instead a new class (PlaneStrategy) would be added. This way new features can be added without modifying existing code. This not only helps maintain code and reduce coupling but also enhances the flexibility of adding new features. By implementing the shared "Strategy" interface, we can add new behaviours without altering the existing code base, preserving code and making the process easier.