**Python -pytest**

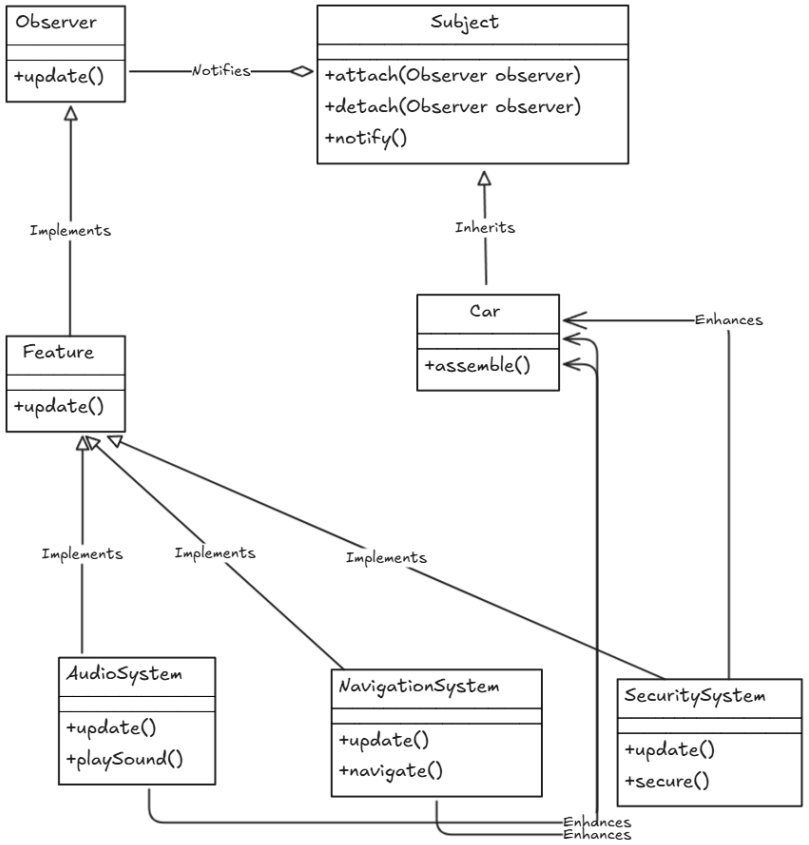
**Prompt**

Observer

| |
|---|
| Use the Observer pattern to write code in Python programming language and write pytest to test the given code that have 100% statement coverage from the requirements below.<br><br>Use Observer pattern to design automotive system development by adding features to cars, such as audio system, navigation system or security systems without changing the main structure of the car. |

**Diagram**

Observer



**Gemini 1.5 Pro**

**Observer**

รอบที่ 1

Code.py

```
from abc import ABC, abstractmethod



class Car:
    """
    The Subject (Car) maintains a list of its dependents, called observers,
    and notifies them automatically of any state changes, usually by calling
    one of their methods.
    """

    def __init__(self):
        self._observers = []
        self._features = []

    def attach(self, observer):
        """Attach an observer to the car."""
        self._observers.append(observer)

    def detach(self, observer):
        """Detach an observer from the car."""
        self._observers.remove(observer)

    def notify(self):
        """Notify all observers about an update."""
        for observer in self._observers:
            observer.update(self)

    def add_feature(self, feature):
```

```
        """Add a new feature to the car."""
        self._features.append(feature)
        print(f"Car: Added feature: {feature}")
        self.notify()

    def get_features(self):
        """Get the list of features in the car."""
        return self._features


class Observer(ABC):
    """
    The Observer interface declares the update method, used by subjects.
    """

    @abstractmethod
    def update(self, subject):
        """Receive update from subject."""
        pass


class AudioSystem(Observer):
    """
    Concrete Observers react to the updates issued by the Subject they had been
    attached to.
    """

    def update(self, subject):
        """React to the car update."""
        if "Audio System" in subject.get_features():
            print("AudioSystem: Reacting to car update: Audio system installed.")


class NavigationSystem(Observer):
```

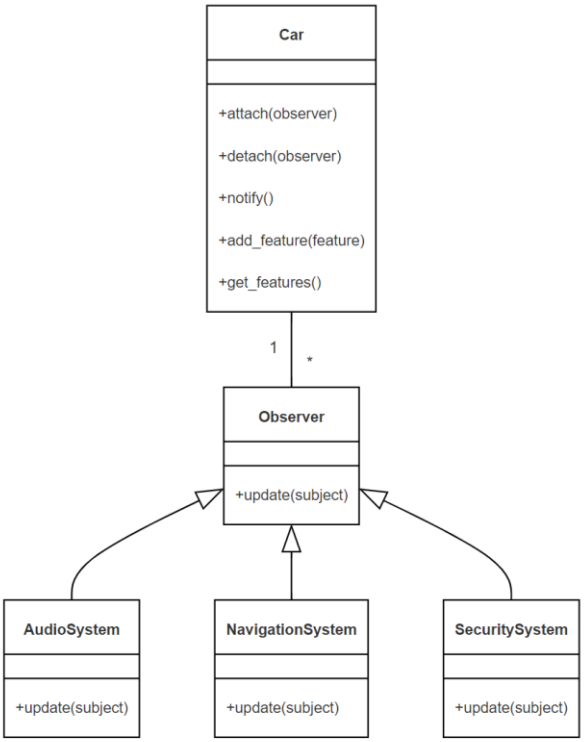
```
    """
    Concrete Observers react to the updates issued by the Subject they had been
    attached to.
    """

    def update(self, subject):
        """React to the car update."""
        if "Navigation System" in subject.get_features():
            print("NavigationSystem: Reacting to car update: Navigation system installed.")


class SecuritySystem(Observer):
    """
    Concrete Observers react to the updates issued by the Subject they had been
    attached to.
    """

    def update(self, subject):
        """React to the car update."""
        if "Security System" in subject.get_features():
            print("SecuritySystem: Reacting to car update: Security system installed.")
```

Test.py

```
import pytest
from code import Car, AudioSystem, NavigationSystem, SecuritySystem


def test_car_features():
    """Test adding features to the car and observer notifications."""
    car = Car()
    audio_system = AudioSystem()
    navigation_system = NavigationSystem()
    security_system = SecuritySystem()
```

```
    car.attach(audio_system)
    car.attach(navigation_system)
    car.attach(security_system)

    car.add_feature("Audio System")
    car.add_feature("Navigation System")
    car.add_feature("Security System")

    assert "Audio System" in car.get_features()
    assert "Navigation System" in car.get_features()
    assert "Security System" in car.get_features()


def test_observer_detach():
    """Test detaching an observer from the car."""
    car = Car()
    audio_system = AudioSystem()

    car.attach(audio_system)
    car.detach(audio_system)

    car.add_feature("Audio System")

    assert "Audio System" in car.get_features()
```

Diagram From Code:




Result:

split class in code.py

TestObserver.py(Edit)

from Car import Car

from Observer import AudioSystem, NavigationSystem, SecuritySystem

2 Pass