Python -pytest

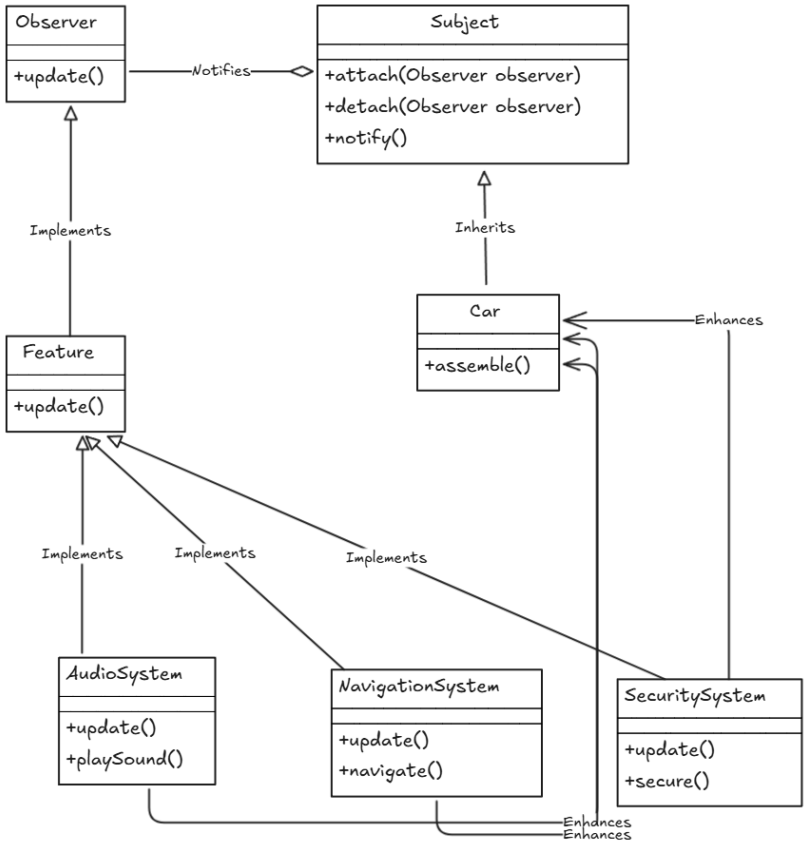
Prompt

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. 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 3

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."""
        if observer not in self._observers:  # Prevent duplicate additions
            self._observers.append(observer)

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

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

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

    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, feature):
        """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, feature):
        """React to the car update."""
        if feature == "Audio System":
            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, feature):
        """React to the car update."""
        if feature == "Navigation System":
            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, feature):
        """React to the car update."""
        if feature == "Security System":
            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)  # Detach the observer

    car.add_feature("Audio System")

    assert "Audio System" in car.get_features()



def test_observer_update_no_feature():
    """Test that update is called with the correct feature."""
    car = Car()
```

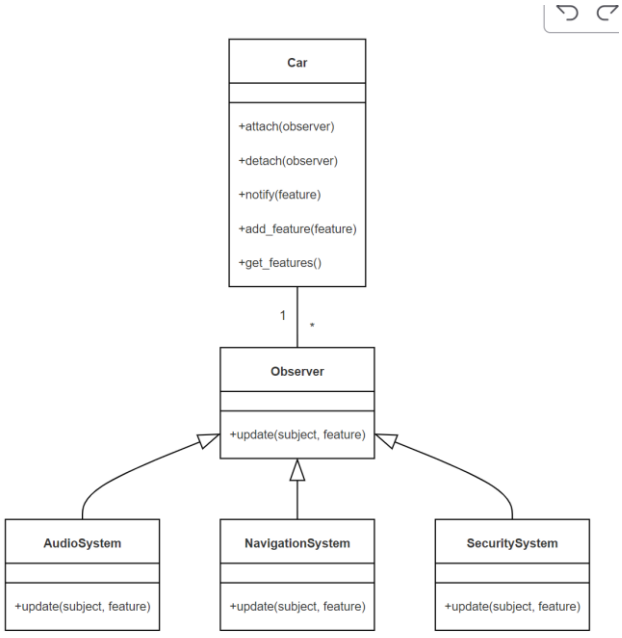
```
    audio_system = AudioSystem()
    car.attach(audio_system)
    car.add_feature("Other Feature")  # Trigger update but should not print anything
    assert "Other Feature" in car.get_features()

def test_attach_duplicate_observer():
    """Test that attaching the same observer twice doesn't result in duplicate notifications."""
    car = Car()
    audio_system = AudioSystem()

    car.attach(audio_system)
    car.attach(audio_system)  # Try to attach the same observer again

    car.add_feature("Audio System")
    # Count the number of times "Audio system installed" is printed (should be 1)
    assert car.get_features().count("Audio System") == 1
```

Diagram From Code:



Result:

split class in code.py

TestObserver.py(Edit)

```
from Car import Car
from Observer import AudioSystem, NavigationSystem, SecuritySystem
```

4 Pass