Safe, Non-Invasive Changes

Week 5: Discussion Session

Advanced Software Engineering

Session Timetable

Activity	Time
Greetings, Warmup	5 min
Sprout Method & Class	15 min
Exercise 1	15 min
Wrap Method & Class	15 min
Exercise 2	15 min
Summary	10 min
Closing	5 min

Warmup

- What's your current strategy when you need to change legacy code?
- Ever added a wrapper instead of modifying directly?
- Share in chat: "Sprout" or "Wrap" — which sounds safer to you?

Sprout Method

- Add new logic in a new method
- Call the new method from existing one
- No edits to original logic

Python Example

```python
class VehicleLogger:
    def log_trip(self, vehicle_type, distance_km, fuel_used_liters):
        print(f"Trip: {distance_km} km")
        if vehicle_type == "gasoline":
            efficiency = distance_km / fuel_used_liters
            print(f"Fuel efficiency: {efficiency:.2f} km/l")
        elif vehicle_type == "electric":
            efficiency = distance_km / fuel_used_liters  # misuse of field
            print(f"Energy efficiency: {efficiency:.2f} km/kWh")
```

Improved Code Example

```python
class VehicleLogger:
    def log_trip(self, vehicle_type, distance_km, fuel_or_energy_used):
        print(f"Trip: {distance_km} km")
        self._log_efficiency(vehicle_type, distance_km, fuel_or_energy_used)
```

Sprout Class

- Create a new class for new behavior
- Keeps old logic intact
- Ideal for complex changes needing state

Python Example

```python
class VehicleLogger:
    def log_trip(self, vehicle_type, distance_km, fuel_used_liters):
        print(f"Trip: {distance_km} km")
        if vehicle_type == "gasoline":
            efficiency = distance_km / fuel_used_liters
            print(f"Fuel efficiency: {efficiency:.2f} km/l")
        elif vehicle_type == "electric":
            efficiency = distance_km / fuel_used_liters
            print(f"Energy efficiency: {efficiency:.2f} km/kWh")
```

Improved Code Example

```python
class VehicleLogger:
    def log_trip(self, vehicle_type, distance_km, fuel_or_energy_used):
        print(f"Trip: {distance_km} km")
        if vehicle_type == "electric":
            EVLogger().log_efficiency(distance_km, fuel_or_energy_used)
```

Exercise 1

- When refactoring legacy code, how do you decide whether to use a Sprout Method or a Sprout Class?
- What are the trade-offs in testability, cohesion, and future maintenance?
- Time limit: 15 minutes

Wrap Method

- The original method is renamed
- A new method with the same name as the original wraps it
- New logic (e.g. logging, validation) is inserted safely
- No change to existing callers — they still call same method

Python Example

```python
class Engine:
    def calculate_torque(self, rpm, throttle):
        return (rpm * throttle) / 100.0
```

Improved Code Example

```python
class Engine:
    def _calculate_torque_original(self, rpm, throttle):
        return (rpm * throttle) / 100.0

    def calculate_torque(self, rpm, throttle):
        print(f"[LOG] Inputs: rpm={rpm}, throttle={throttle}")
        torque = self._calculate_torque_original(rpm, throttle)
        print(f"[LOG] Output: torque={torque}")
        return torque
```

Wrap Class

- Create a new class that wraps the legacy class
- Delegates calls while injecting new logic
- Helps isolate change when subclassing is risky

Python Example

```
class Engine:
    def calculate_torque(self, rpm, throttle):
        return (rpm * throttle) / 100.0
```

Improved Code Example

```
class LoggingEngine:
    def __init__(self, engine):
        self._engine = engine

    def calculate_torque(self, rpm, throttle):
        print(f"[LOG] Calculating torque: rpm={rpm}, throttle={throttle}")
        torque = self._engine.calculate_torque(rpm, throttle)
        print(f"[LOG] Torque result: {torque}")
        return torque
```

Usage

```
engine = LoggingEngine(Engine())
engine.calculate_torque(3000, 70)
```

Exercise 2

- How do you decide whether legacy behavior should remain in its current class or move into a Sprout Class?
- What separation-of-concerns signals inform that choice?
- Time limit: 15 minutes

Decorators and the Wrap Method

- Decorators are a special form of wrapping that allows for dynamic extension.
- In the context of the Wrap Method, decorators provide an elegant solution for layering additional behavior.

Python Decorator Pattern Example

```python
class Vehicle:
    def drive(self):
        return "Driving"

class VehicleDecorator(Vehicle):
    def __init__(self, vehicle):
        self._vehicle = vehicle

    def drive(self):
        return self._vehicle.drive()

class LoggingDecorator(VehicleDecorator):
    def drive(self):
        action = self._vehicle.drive()
        print(f"[LOG] Action: {action}")
        return action

```

Comparison & Benefits

Technique	Location of Change	Scope	Purpose	Risk	Code Impact
Sprout Method	Same class	One method	Isolate new logic	Low	Add method, call it
Sprout Class	New class	Functionality	Extract cohesivebehavior	Low	New class, inject it
Wrap Method	Same class	One method	Insert logic aroundmethod	Low	Rename + wrapmethod
Wrap Class	Wrapper	Multiple methods	Add behavior aroundlegacy class	Med	New wrappercreated

Summary

- **Sprout** = add new code that old code calls
- **Wrap** = write code that calls into the old code
- Use *Method* for simple logic, *Class* for complex or stateful logic
- These give you *safe entry points* into legacy code

Final Thought

"You don't need to clean up the whole kitchen to make a cup of tea. Just clear a spot."
— Inspired by Feathers' philosophy