1. **SOLID – I – Interface Segregation**

from abc import ABC, abstractmethod

class SmartDevice(ABC):

    @abstractmethod

    def turn\_on(self) -> None:

        pass

    @abstractmethod

    def turn\_off(self) -> None:

        pass

    @abstractmethod

    def set\_temperature(self, temperature: int) -> None:

        pass

class SmartLight(SmartDevice):

    def turn\_off(self) -> None:

        print("Turning lights ON")

    def turn\_on(self) -> None:

        print("Turning lights OFF")

    # THIS IS NOT INTERFACE SEGREFATION

    def set\_temperature(self, temperature: int) -> None:

        raise NotImplementedError("Smart light device cannot set temperature")

class SmartTherm(SmartDevice):

    def turn\_off(self) -> None:

        print("Turning temperature OFF")

    def turn\_on(self) -> None:

        print("Turning temperature ON")

    def set\_temperature(self, temperature: int) -> None:

        print(f"Setting temperature {temperature}")

smartLight = SmartLight()

smartLight.turn\_off()

smartLight.turn\_on()

smartTherm = SmartTherm()

smartTherm.turn\_off()

smartTherm.turn\_on()

smartTherm.set\_temperature(10)

from abc import ABC, abstractmethod

class ISwitchable(ABC):

    @abstractmethod

    def turn\_on(self) -> None:

        pass

    def turn\_off(self) -> None:

        pass

class ITemperatureRegulatable(ABC):

    @abstractmethod

    def set\_temperature(self, temperature: int) -> None:

        pass

class SmartLight(ISwitchable):

    def turn\_off(self) -> None:

        print("Turning light OFF")

    def turn\_on(self) -> None:

        print("Turning light ON")

class SmartTemperature(ISwitchable, ITemperatureRegulatable):

    def turn\_off(self) -> None:

        print("Turning Temp. OFF")

    def turn\_on(self) -> None:

        print("Turning Temp. ON")

    def set\_temperature(self, temperature: int) -> None:

        print(f"Setting temperature {temperature}")

smartLight = SmartLight()

smartLight.turn\_off()

smartLight.turn\_on()

smartTherm = SmartTemperature()

smartTherm.turn\_off()

smartTherm.turn\_on()

smartTherm.set\_temperature(10)

# using PROTOCOL

from typing import Protocol

class IOpoeracionFinanciera(Protocol):

    def depositar(self, monto: float) -> None: ...

    def retirar(self, monto: float) -> None: ...

    def transferir(self, monto: float, a\_cuenta: str) -> None: ...

class CuentaAhorros:

    def depositar(self, monto: float) -> None:

        print(f"Depositando { monto} en cuenta de ahorros")

    def retirar(self, amount: float) -> None:

        print(f"Retirando {amount} de cuenta de ahorros")

    # Raise una excepcion rompe ISP

class CuentaCorriente:

    def depositar(self, monto: float) -> None:

        print(f"Depositando { monto} en cuenta corriente")

    def retirar(self, amount: float) -> None:

        print(f"Retirando {amount} de cuenta corriente")

    def transferir(self, amount: float, a\_cuenta: str) -> None:

        print(f"Transfiriendo {amount} de cuenta corriente a cuenta { a\_cuenta}")

cuentaAhorros = CuentaAhorros()

cuentaAhorros.depositar(100)

cuentaAhorros.retirar(50)

cuentaCorriente = CuentaCorriente()

cuentaCorriente.depositar(100)

cuentaCorriente.retirar(50)

cuentaCorriente.transferir(20, "ABCSK189148")

from typing import Protocol

# COntratos de deposito, retiro y transferencia

class IDepositar(Protocol):

    def depositar(self, monto: float) -> None: ...

class IRetirar(Protocol):

    def retirar(self, monto: float) -> None: ...

class ITransferir(Protocol):

    def transferir(self, monto: float) -> None: ...

class CuentaAhorros:

    def depositar(self, monto: float) -> None:

        print(f"Depositando { monto} en cuenta de ahorros ")

    def retirar(self, amount: float) -> None:

        print(f"Retirando {amount} de cuenta de ahorros")

class CuentaCorriente:

    def depositar(self, monto: float) -> None:

        print(f"Depositando { monto} en cuenta corriente")

    def retirar(self, amount: float) -> None:

        print(f"Retirando {amount} de cuenta corriente")

    def transferir(self, amount: float, a\_cuenta: str) -> None:

        print(f"Transfiriendo {amount} de cuenta corriente a cuenta { a\_cuenta}")

def realizar\_pago(cuenta: ITransferir, monto: float) -> None:

    cuenta.transferir(monto)

cuentaCorriente = CuentaCorriente()

realizar\_pago(cuentaCorriente, 20)

# cuentaAhorros.depositar(100, 20)

# cuentaAhorros.retirar(50)

# cuentaCorriente = CuentaCorriente()

# cuentaCorriente.depositar(100)

# cuentaCorriente.retirar(50)

# cuentaCorriente.transferir(20, "ABCSK189148")