This guide is a comprehensive collection of classic object-oriented design problems with complete code implementations and explanations, especially aligned with Amazon interview expectations. Use this as your reference and practice book.

Updated: July 2025

---

1. Learning Resources & Preparation Links

- Neetcode OOD Course
- Grokking the LLD Interview (Educative)
- Free Grokking Python Code
- LeetCode Design Tag
- Awesome OOD Java Resources
- LeetCode Amazon OOD Discussion

---

2. Core Tips

- Focus on modular, extensible class design
- Always communicate assumptions and constraints
- Use object-oriented principles like SRP, OCP, Liskov Substitution
- Focus on real-world behavior: how the system scales, how responsibilities are distributed

---

3. Essential SOLID Principles for Amazon Interviews

1. Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change - it should have only one job.

Why Amazon loves this: Shows you can write maintainable, modular code that's easy to test and debug.

```
# ❌ BAD: Multiple responsibilities

class User:

    def __init__(self, name, email):

        self.name = name

        self.email = email


    def save_to_database(self):

        # Database logic here

        pass


    def send_email(self):

        # Email sending logic here

        pass


    def validate_email(self):

        # Email validation logic here

        pass


# ✅ GOOD: Single responsibility per class
```

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

class UserRepository:
    def save(self, user):
        # Database logic here
        pass

class EmailService:
    def send_email(self, user, message):
        # Email sending logic here
        pass

class EmailValidator:
    def validate(self, email):
        # Email validation logic here
        return "@" in email
```

2. Open/Closed Principle (OCP)

Definition: Software entities should be open for extension but closed for modification.

Interview Impact: Demonstrates you can design systems that grow without breaking existing code.

```
# ❌ BAD: Modifying existing code for new features
```

```python
class DiscountCalculator:
    def calculate_discount(self, customer_type, amount):
        if customer_type == "regular":
            return amount * 0.05
        elif customer_type == "premium":
            return amount * 0.10
        elif customer_type == "vip":  # Need to modify existing code
            return amount * 0.15

# ✅ GOOD: Open for extension, closed for modification
from abc import ABC, abstractmethod


class DiscountStrategy(ABC):
    @abstractmethod
    def calculate_discount(self, amount):
        pass


class RegularCustomerDiscount(DiscountStrategy):
    def calculate_discount(self, amount):
        return amount * 0.05


class PremiumCustomerDiscount(DiscountStrategy):
    def calculate_discount(self, amount):
        return amount * 0.10
```

```
class VIPCustomerDiscount(DiscountStrategy):  # New feature without modification
    def calculate_discount(self, amount):
        return amount * 0.15


class DiscountCalculator:
    def __init__(self, strategy: DiscountStrategy):
        self.strategy = strategy


    def calculate(self, amount):
        return self.strategy.calculate_discount(amount)
```

Top 3 Amazon Design Patterns

1. Strategy Pattern (Frequency: 5/5) 🔥

When to use: When you have multiple ways to perform a task and want to switch between them dynamically.

Amazon examples: Payment processing, shipping methods, pricing strategies

```
from abc import ABC, abstractmethod


# Strategy Interface
class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount: float) -> bool:
```

```
        """Process payment and return success status"""
        pass

# Concrete Strategies
class CreditCardPayment(PaymentProcessor):
    def __init__(self, card_number: str, expiration_date: str, cvv: str):
        self.card_number = card_number
        self.expiration_date = expiration_date
        self.cvv = cvv

    def process_payment(self, amount: float) -> bool:
        print(f"Processing credit card payment of ${amount}")
        print(f"Using card ending in {self.card_number[-4:]}")
        # Simulate payment gateway communication
        return True

class PayPalPayment(PaymentProcessor):
    def __init__(self, email: str):
        self.email = email

    def process_payment(self, amount: float) -> bool:
        print(f"Processing PayPal payment of ${amount}")
        print(f"Using PayPal account: {self.email}")
        # Simulate PayPal API call
```

```python
        return True

class CryptoPayment(PaymentProcessor):
    def __init__(self, wallet_address: str, currency: str):
        self.wallet_address = wallet_address
        self.currency = currency

    def process_payment(self, amount: float) -> bool:
        print(f"Processing {self.currency} payment of ${amount}")
        print(f"To wallet: {self.wallet_address}")
        # Simulate blockchain transaction
        return True

# Context class
class PaymentContext:
    def __init__(self, payment_processor: PaymentProcessor):
        self.payment_processor = payment_processor

    def set_payment_method(self, payment_processor: PaymentProcessor):
        self.payment_processor = payment_processor

    def execute_payment(self, amount: float) -> bool:
        return self.payment_processor.process_payment(amount)
```

```
# Usage Example

def test_strategy_pattern():

    # Client can switch payment methods dynamically

    payment_context = PaymentContext(CreditCardPayment("1234567890123456", "12/25", "123"))

    payment_context.execute_payment(100.0)



    # Switch to PayPal

    payment_context.set_payment_method(PayPalPayment("user@example.com"))

    payment_context.execute_payment(200.0)



    # Switch to Crypto

    payment_context.set_payment_method(CryptoPayment("1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa", "BTC"))

    payment_context.execute_payment(300.0)
```

Why Amazon loves Strategy Pattern:

- Easy to add new payment methods without changing existing code (OCP)
- Each payment method has single responsibility (SRP)
- Testable in isolation
- Mirrors real Amazon payment systems

2. Factory Pattern (Frequency: 5/5) 🔥

When to use: When you need to create objects without specifying their exact classes.

Amazon examples: Creating different vehicle types, notification services, data parsers

```
from abc import ABC, abstractmethod

from enum import Enum
```

```python
class VehicleType(Enum):
    CAR = "car"
    TRUCK = "truck"
    MOTORCYCLE = "motorcycle"

# Product Interface
class Vehicle(ABC):
    @abstractmethod
    def get_max_speed(self) -> int:
        pass

    @abstractmethod
    def get_fuel_efficiency(self) -> float:
        pass

    @abstractmethod
    def get_description(self) -> str:
        pass

# Concrete Products
class Car(Vehicle):
    def __init__(self, model: str):
        self.model = model
```

```
    def get_max_speed(self) -> int:
        return 120

    def get_fuel_efficiency(self) -> float:
        return 25.0

    def get_description(self) -> str:
        return f"Car: {self.model}"

class Truck(Vehicle):
    def __init__(self, model: str, cargo_capacity: int):
        self.model = model
        self.cargo_capacity = cargo_capacity

    def get_max_speed(self) -> int:
        return 80

    def get_fuel_efficiency(self) -> float:
        return 12.0

    def get_description(self) -> str:
        return f"Truck: {self.model} (Capacity: {self.cargo_capacity}kg)"
```

```python
class Motorcycle(Vehicle):
    def __init__(self, model: str):
        self.model = model

    def get_max_speed(self) -> int:
        return 150

    def get_fuel_efficiency(self) -> float:
        return 45.0

    def get_description(self) -> str:
        return f"Motorcycle: {self.model}"

# Factory
class VehicleFactory:
    @staticmethod
    def create_vehicle(vehicle_type: VehicleType, **kwargs) -> Vehicle:
        if vehicle_type == VehicleType.CAR:
            return Car(kwargs.get('model', 'Generic Car'))
        elif vehicle_type == VehicleType.TRUCK:
            return Truck(
                kwargs.get('model', 'Generic Truck'),
                kwargs.get('cargo_capacity', 1000)
            )
```

```python
        elif vehicle_type == VehicleType.MOTORCYCLE:
            return Motorcycle(kwargs.get('model', 'Generic Motorcycle'))
        else:
            raise ValueError(f"Unknown vehicle type: {vehicle_type}")


# Usage Example
def test_factory_pattern():
    # Create different vehicles without knowing their concrete classes
    car = VehicleFactory.create_vehicle(VehicleType.CAR, model="Toyota Camry")
    truck = VehicleFactory.create_vehicle(VehicleType.TRUCK, model="Ford F-150",
cargo_capacity=2000)
    motorcycle = VehicleFactory.create_vehicle(VehicleType.MOTORCYCLE, model="Harley
Davidson")


    vehicles = [car, truck, motorcycle]


    for vehicle in vehicles:
        print(f"{vehicle.get_description()}")
        print(f"Max Speed: {vehicle.get_max_speed()} mph")
        print(f"Fuel Efficiency: {vehicle.get_fuel_efficiency()} mpg")
        print("-" * 40)
```

Advanced Factory Pattern - Abstract Factory:

```python
# For creating families of related objects
class DeliveryVehicleFactory(ABC):
```

```python
    @abstractmethod
    def create_vehicle(self) -> Vehicle:
        pass

    @abstractmethod
    def create_driver(self) -> 'Driver':
        pass

class CarDeliveryFactory(DeliveryVehicleFactory):
    def create_vehicle(self) -> Vehicle:
        return Car("Delivery Car")

    def create_driver(self) -> 'Driver':
        return CarDriver()

class TruckDeliveryFactory(DeliveryVehicleFactory):
    def create_vehicle(self) -> Vehicle:
        return Truck("Delivery Truck", 5000)

    def create_driver(self) -> 'Driver':
        return TruckDriver()
```

3. Decorator Pattern (Frequency: 5/5) 🔥

When to use: When you want to add behavior to objects dynamically without altering their structure.

Amazon examples: Pizza toppings, insurance add-ons, feature flags

```
from abc import ABC, abstractmethod


# Component Interface
class Coffee(ABC):
    @abstractmethod
    def get_description(self) -> str:
        pass

    @abstractmethod
    def get_cost(self) -> float:
        pass


# Concrete Component
class BasicCoffee(Coffee):
    def get_description(self) -> str:
        return "Basic Coffee"

    def get_cost(self) -> float:
        return 2.0


# Decorator Base Class
class CoffeeDecorator(Coffee):
    def __init__(self, coffee: Coffee):
```

```python
        self._coffee = coffee

    def get_description(self) -> str:
        return self._coffee.get_description()

    def get_cost(self) -> float:
        return self._coffee.get_cost()

# Concrete Decorators
class MilkDecorator(CoffeeDecorator):
    def get_description(self) -> str:
        return self._coffee.get_description() + ", Milk"

    def get_cost(self) -> float:
        return self._coffee.get_cost() + 0.5

class SugarDecorator(CoffeeDecorator):
    def get_description(self) -> str:
        return self._coffee.get_description() + ", Sugar"

    def get_cost(self) -> float:
        return self._coffee.get_cost() + 0.2

class WhipCreamDecorator(CoffeeDecorator):
```

```python
    def get_description(self) -> str:
        return self._coffee.get_description() + ", Whip Cream"

    def get_cost(self) -> float:
        return self._coffee.get_cost() + 1.0

class VanillaDecorator(CoffeeDecorator):
    def get_description(self) -> str:
        return self._coffee.get_description() + ", Vanilla"

    def get_cost(self) -> float:
        return self._coffee.get_cost() + 0.7

# Usage Example
def test_decorator_pattern():
    # Start with basic coffee
    coffee = BasicCoffee()
    print(f"{coffee.get_description()}: ${coffee.get_cost():.2f}")

    # Add milk
    coffee = MilkDecorator(coffee)
    print(f"{coffee.get_description()}: ${coffee.get_cost():.2f}")

    # Add sugar
```

```
coffee = SugarDecorator(coffee)
print(f"{coffee.get_description()}: ${coffee.get_cost():.2f}")

# Add whip cream
coffee = WhipCreamDecorator(coffee)
print(f"{coffee.get_description()}: ${coffee.get_cost():.2f}")

# Add vanilla
coffee = VanillaDecorator(coffee)
print(f"{coffee.get_description()}: ${coffee.get_cost():.2f}")

# Output:
# Basic Coffee: $2.00
# Basic Coffee, Milk: $2.50
# Basic Coffee, Milk, Sugar: $2.70
# Basic Coffee, Milk, Sugar, Whip Cream: $3.70
# Basic Coffee, Milk, Sugar, Whip Cream, Vanilla: $4.40
```

Interview Tips: How to Show Design Pattern Knowledge

1. Mention Patterns by Name

Instead of: "I'll create different classes for payment methods" Say: "I'll use the Strategy pattern to handle different payment methods, which follows the Open/Closed Principle"

2. Explain the Benefits

- **Strategy**: "This allows us to add new payment methods without modifying existing code"
- **Factory**: "This encapsulates object creation and makes the system more flexible"
- **Decorator**: "This lets us add features dynamically without creating a class explosion"

3. Connect to SOLID Principles

```
# When designing, explicitly mention SOLID principles:

class PaymentProcessor(ABC):  # Interface Segregation Principle

    @abstractmethod

    def process_payment(self, amount: float) -> bool:  # Single Responsibility

        pass



class OrderService:  # Dependency Inversion Principle

    def __init__(self, payment_processor: PaymentProcessor):

        self.payment_processor = payment_processor  # Depend on abstractions, not concretions
```

4. Discuss Extensibility

"If Amazon wants to add a new payment method like 'Buy Now Pay Later', we just create a new class implementing PaymentProcessor. No existing code changes needed."

5. Address Real-World Concerns

- "For the Factory pattern, we might cache objects for performance"
- "For Strategy pattern, we could load payment processors from configuration"
- "For Decorator pattern, we need to be careful about the order of decorators"

Practice Questions

1. **Strategy Pattern**: Design a shipping cost calculator that can use different algorithms (standard, express, overnight)

2. **Factory Pattern**: Create a notification system that can send emails, SMS, or push notifications

3. **Decorator Pattern**: Design a file compression system where you can apply multiple compression algorithms

4. **Combined Patterns**: Design an e-commerce system that uses Factory for creating products, Strategy for pricing, and Decorator for adding product features

Remember: Amazon interviewers love when you can articulate WHY you chose a pattern and how it makes the system more maintainable and extensible!

---

4. Amazon OOD Real Problems – Full Code with Explanations

1. Package Manager / NPM Installer (Frequency: 5 / 5)

Create a package manager/package dependency system.
We are required to support the installation of a package and all of its dependent packages.

Here is an example of what we would need to install:

"Install package A"

A depends on B, C
B depends on D, E, F
C depends on F
F depends on G

Define the data model for a package and create a package manager implementation that can install packages
using different strategies (e.g., project-level or global) by structuring the code in a way that is as flexible as possible.

You do not have to implement the actual installation of the package on a specific platform but assume you have access to the functionality.

Problem: Build a package manager system that resolves installation order based on dependencies.

Concepts: Graph traversal, topological sort, cycle detection

```
# create a package manager/ package dependency system

from collections import defaultdict, deque

class PackageManager:

   def __init__(self):

       self.result = []

       self.graph = defaultdict(list)

       self.visiting = set() # mark a node as visiting when the node is being visited in the graph path

       self.visited = set() # mark as visited when a node is dealt with

   def build_graph(self, dependencies):  # dependencies include the src node and dst node

       # build graph using the dependencies

       for src, dst in dependencies:

           self.graph[src].append(dst)

   def install(self, package):  # this works as the main function

       # start from package and install recursively to install all the packages that this package is reliant on

       # run dfs to get the topological sort result

       if not self.dfs(package):
```

```python
            return []
        else:
            return self.result[::-1]
    def dfs(self, package):
        if package in self.visiting: # if yes, means cycle detected
            return False
        if package in self.visited: # if yes, means already dealt with this node
            return True
        self.visiting.add(package)
        for neighbor in self.graph[package]:
            if not self.dfs(neighbor):
                return False
        self.visiting.remove(package)
        self.visited.add(package)
        self.result.append(package)
        return True



# use this as a dfs function to perform topological sort
def test_package_manager():
```

```
pm = PackageManager()


# Example 1: Simple dependency graph with no cycles
dependencies = [
    ("A", "B"),
    ("A", "C"),
    ("B", "D"),
    ("C", "E"),
    ("D", "F")
]


pm.build_graph(dependencies)
try:
    result = pm.install("A")
    print("Installation order:", result)
except ValueError as e:
    print(e)


# Example 2: Cyclic dependency graph
pm = PackageManager()  # Create a new instance to reset everything
dependencies_with_cycle = [
```

```
        ("A", "B"),

        ("B", "C"),

        ("C", "A")

    ]



    pm.build_graph(dependencies_with_cycle)

    try:

        result = pm.install("A")

        print("Installation order:", result)

    except ValueError as e:

        print(e)



# Run the tests

test_package_manager()
```

---

2. Parking Lot System (Frequency: 4 / 5)

Problem: Design a parking lot system that supports cars of different sizes, tracks available spaces, and supports future extensions like payments.

Concepts: Enum, inheritance, SRP

```python
# define a few classes: Vehicle: (inherited class: large medium small
)

# driver class: pay

# parking system class: parking garage


# 1. clarification:

# 2. entities: Vehicle, parking system, parking lot, payment

from abc import ABC, abstractmethod

from enum import Enum

class VehicleSize(Enum):

   SMALL = 1

   MEDIUM = 2

   LARGE = 3

# status can also be enum

class ParkingSpotType(Enum):

   SMALL = 1

   MEDIUM = 2

   LARGE = 3

class Vehicle(ABC): # define interface

   def __init__(self, size: VehicleSize, license_plate: str):

       self._size = size
```

```python
        self._license_plate = license_plate

    def get_size(self):

        return self._size

    def get_license_plate(self):

        return self._license_plate

    @abstractmethod

    def get_parking_fee(self, hours):

        pass

class SmallCar(Vehicle): # define concrete class

    def __init__(self, license_plate: str):

        super().__init__(VehicleSize.SMALL, license_plate)

    def get_parking_fee(self, hours):

        return hours * 10

class MediumCar(Vehicle):

    def __init__(self, license_plate: str):

        super().__init__(VehicleSize.MEDIUM, license_plate)

    def get_parking_fee(self, hours):

        return hours * 5

class LargeCar(Vehicle):

    def __init__(self, license_plate: str):

        super().__init__(VehicleSize.LARGE, license_plate)
```

```python
    def get_parking_fee(self, hours):
        return hours * 3


class ParkingSpot:
    def __init__(self, spot_type: ParkingSpotType):
        self._spot_type = spot_type
        self._is_occupied = False
        self._vehicle = None
    def can_fit_vehicle(self, vehicle: Vehicle):
        if self._spot_type == ParkingSpotType.LARGE:
            return True  # Large spots can fit any vehicle
        elif self._spot_type == ParkingSpotType.MEDIUM:
            return vehicle.get_size() in [VehicleSize.MEDIUM,
VehicleSize.SMALL]
        elif self._spot_type == ParkingSpotType.SMALL:
            return vehicle.get_size() == VehicleSize.SMALL
    def park(self, vehicle: Vehicle):
        if self.can_fit_vehicle(vehicle) and not self._is_occupied:
            self._vehicle = vehicle
            self._is_occupied = True
            return True
```

```
        return False

    def remove_vehicle(self):

        self._is_occupied = False

        self._vehicle = None

    def is_occupied(self):

        return self._is_occupied


class Driver:

    def __init__(self, id, vehicle):

        self._id = id

        self._vehicle = vehicle

        self._payment_due = 0

    def get_vehicle(self):

        return self._vehicle

    def get_id(self):

        return self._id

    def charge(self, amount):

        self._payment_due += amount

    def get_payment_due(self):

        return self._payment_due
```

```python
class ParkingFloor:

    def __init__(self, spots):

        # `spots` is a list of ParkingSpot objects

        self._spots = spots

        self._vehicle_map = {}

    def park_vehicle(self, vehicle: Vehicle):

        for spot in self._spots:

            if not spot.is_occupied() and
spot.can_fit_vehicle(vehicle):

                spot.park(vehicle)

                self._vehicle_map[vehicle.get_license_plate()] = spot

                return True

        return False

    def remove_vehicle(self, vehicle: Vehicle):

        license_plate = vehicle.get_license_plate()

        if license_plate in self._vehicle_map:

            spot = self._vehicle_map[license_plate]

            spot.remove_vehicle()

            del self._vehicle_map[license_plate]

            return True
```

```
        return False


class ParkingGarage:

    def __init__(self, floors):

        self._floors = floors  # `floors` is a list of ParkingFloor
objects

    def park_vehicle(self, vehicle: Vehicle):

        for floor in self._floors:

            if floor.park_vehicle(vehicle):

                return True

        return False

    def remove_vehicle(self, vehicle: Vehicle):

        for floor in self._floors:

            if floor.remove_vehicle(vehicle):

                return True

        return False

import datetime

import math

class ParkingSystem:

    def __init__(self, parkingGarage, hourlyRate):

        self._parkingGarage = parkingGarage
```

```
        self._hourlyRate = hourlyRate
        self._timeParked = {}  # map driverId to time that they parked

    def park_vehicle(self, driver):
        currentHour = datetime.datetime.now().hour
        isParked =
self._parkingGarage.park_vehicle(driver.get_vehicle())
        if isParked:
            self._timeParked[driver.get_id()] = currentHour
        return isParked


    def remove_vehicle(self, driver):
        if driver.get_id() not in self._timeParked:
            return False
        currentHour = datetime.datetime.now().hour
        timeParked = math.ceil(currentHour -
self._timeParked[driver.get_id()])
        driver.charge(timeParked * self._hourlyRate)
        del self._timeParked[driver.get_id()]
        return
self._parkingGarage.remove_vehicle(driver.get_vehicle())


class PaymentProcessor(ABC): # consider about strategy pattern
```

```
    @abstractmethod
    def process_payment(self, amount: float) -> None:
        """Centralized contract for processing payment.
           This method must be overridden by subclasses.
        """
        pass
class CreditCardPayment(PaymentProcessor):
    def __init__(self, card_number: str, expiration_date: str, cvv: str):
        self.card_number = card_number
        self.expiration_date = expiration_date
        self.cvv = cvv

    def process_payment(self, amount: float) -> None:
        # Simulate processing a credit card payment
        print(f"Processing credit card payment of ${amount}")
        print(f"Using card number {self.card_number}, expiration date {self.expiration_date}, CVV {self.cvv}")
        # Here would be the actual logic to communicate with a payment gateway
        print("Credit card payment successful.")
```

```
class PayPalPayment(PaymentProcessor):

   def __init__(self, email: str, password: str):

       self.email = email

       self.password = password

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

       # Simulate processing a PayPal payment

       print(f"Processing PayPal payment of ${amount}")

       print(f"Using PayPal account {self.email}")

       # Here would be the actual logic to authenticate and process
the payment

       print("PayPal payment successful.")



# Creating parking spots for a floor (2 small, 1 medium, 1 large)

floor1 = ParkingFloor([ParkingSpot(ParkingSpotType.SMALL),
ParkingSpot(ParkingSpotType.SMALL),

                     ParkingSpot(ParkingSpotType.MEDIUM),
ParkingSpot(ParkingSpotType.LARGE)])

# Creating parking garage with multiple floors

parkingGarage = ParkingGarage([floor1])

# Initialize the Parking System with hourly rate

parkingSystem = ParkingSystem(parkingGarage, 5)
```

```python
# Drivers with vehicles of various types and sizes

driver1 = Driver(1, SmallCar("ABC123"))  # Small car

driver2 = Driver(2, MediumCar("XYZ789"))  # Medium car

driver3 = Driver(3, LargeCar("TRK456"))  # Large truck

# Parking vehicles

print(parkingSystem.park_vehicle(driver1))  # True (fits in a small spot)

print(parkingSystem.park_vehicle(driver2))  # True (fits in a medium spot)

print(parkingSystem.park_vehicle(driver3))  # True (fits in a large spot)

# Removing vehicles and calculating the charges

print(parkingSystem.remove_vehicle(driver1))  # True

print(parkingSystem.remove_vehicle(driver2))  # True

print(parkingSystem.remove_vehicle(driver3))  # True
```

---

3. Unix File Search System   (Frequency: 4 / 5)

Problem: Create a search API that can search through a file system for files matching certain filters (by name, size, extension).

Concepts: Tree traversal, filtering strategy

```python
from abc import ABC, abstractmethod
```

```python
from collections import deque

from typing import List

from enum import Enum

# File

class File:

   def __init__(self, name, size):

       self.name = name

       self.size = size

       self.children = []

       self.is_directory = False if '.' in name else True

       self.children = []

       self.extension = name.split(".")[1] if '.' in name else ""


   def __repr__(self):

       return "{"+self.name+"}"


# Filters

class Filter(ABC): # interface

   def __init__(self):

       pass

   @abstractmethod
```

```
    def apply(self, file):

        pass

class MinSizeFilter(Filter): # concrete class

    def __init__(self, size):

        self.size = size


    def apply(self, file):

        return file.size > self.size


class ExtensionFilter(Filter):

    def __init__(self, extension):  # "txt"

        self.extension = extension


    def apply(self, file):

        return file.extension == self.extension



# LinuxFindCommand


class LinuxFind():

    def __init__(self):
```

```
        self.filters: List[Filter] = []


    def add_filter(self, given_filter):
        # validate given_filter is a filter
        if isinstance(given_filter, Filter):
            self.filters.append(given_filter)


    def apply_OR_filtering(self, root):
        # f1 = File("root_300", 300)
        found_files = []


        # bfs
        queue = deque()
        queue.append(root)
        while queue:
            # print(queue)
            curr_root = queue.popleft()
            if curr_root.is_directory:
                for child in curr_root.children:
                    queue.append(child)
            else:
```

```
                for filter in self.filters:
                    if filter.apply(curr_root):
                        found_files.append(curr_root)
                        print(curr_root)
                        break
        return found_files

    def apply_AND_filtering(self, root):
        found_files = []

        # bfs
        queue = deque()
        queue.append(root)
        while queue:
            curr_root = queue.popleft()
            if curr_root.is_directory:
                for child in curr_root.children:
                    queue.append(child)
            else:
                is_valid = True
                for filter in self.filters:
```

```
                if not filter.apply(curr_root):

                    is_valid = False

                    break

            if is_valid:

                found_files.append(curr_root)

                print(curr_root)


    return found_files




f1 = File("root_300", 300)


f2 = File("fiction_100", 100)

f3 = File("action_100", 100)

f4 = File("comedy_100", 100)

f1.children = [f2, f3, f4]


f5 = File("StarTrek_4.txt", 4)

f6 = File("StarWars_10.xml", 10)

f7 = File("JusticeLeague_15.txt", 15)

f8 = File("Spock_1.jpg", 1)
```

```
f2.children = [f5, f6, f7, f8]


f9 = File("IronMan_9.txt", 9)

f10 = File("MissionImpossible_10.rar", 10)

f11 = File("TheLordOfRings_3.zip", 3)

f3.children = [f9, f10, f11]


f11 = File("BigBangTheory_4.txt", 4)

f12 = File("AmericanPie_6.mp3", 6)

f4.children = [f11, f12]



greater5_filter = MinSizeFilter(5)

txt_filter = ExtensionFilter("txt")


my_linux_find = LinuxFind()

my_linux_find.add_filter(greater5_filter)

my_linux_find.add_filter(txt_filter)


print(my_linux_find.apply_OR_filtering(f1))

print(my_linux_find.apply_AND_filtering(f1))
```

4. Pizza Ordering System (Frequency: 5 / 5)

Problem: Design a customizable pizza ordering system using the Decorator pattern.

Concepts: Design pattern (Decorator), composition over inheritance

```
from abc import ABC, abstractmethod

# don't need to write @staticmethod。in many cases，@staticmethod is only a way to structure code

# Step 1: Define the base Pizza interface

class Pizza(ABC): # foundation class and foundation interface for us

    # __init__ is not necessary, only when the class need to store some entity is needed

    @abstractmethod

    def cost(self) -> float:

        pass


    @abstractmethod

    def description(self) -> str:

        pass


# Step 2: Create concrete Pizza implementations
```

```python
class Margherita(Pizza):

    def cost(self) -> float:

        return 8.0


    def description(self) -> str:

        return "Margherita"


class Pepperoni(Pizza):

    def cost(self) -> float:

        return 10.0


    def description(self) -> str:

        return "Pepperoni"


class Veggie(Pizza):

    def cost(self) -> float:

        return 9.0


    def description(self) -> str:

        return "Veggie"
```

```python
# Step 3: Create a Decorator class

class ToppingDecorator(Pizza):  #

   def __init__(self, pizza: Pizza):  # base class

       self._pizza = pizza



   def cost(self) -> float:

       return self._pizza.cost()



   def description(self) -> str:

       return self._pizza.description()



# Step 4: Add specific toppings as decorators

class Cheese(ToppingDecorator):

   def cost(self) -> float:

       return self._pizza.cost() + 2.0



   def description(self) -> str:

       return self._pizza.description() + ", Cheese"



class Mushroom(ToppingDecorator):

   def cost(self) -> float:
```

```
        return self._pizza.cost() + 1.0


    def description(self) -> str:

        return self._pizza.description() + ", Mushroom"


class Pepper(ToppingDecorator):

    def cost(self) -> float:

        return self._pizza.cost() + 1.5


    def description(self) -> str:

        return self._pizza.description() + ", Pepper"


# Step 5: Factory to create base pizzas

class PizzaFactory:

    # @staticmethod

    def create_pizza(pizza_type: str) -> Pizza:

        if pizza_type == "Margherita":

            return Margherita()

        elif pizza_type == "Pepperoni":

            return Pepperoni()

        elif pizza_type == "Veggie":
```

```
            return Veggie()
        else:
            raise ValueError("Invalid pizza type")


# Usage Example
if __name__ == "__main__":
    # Create a base pizza using the factory
    base_pizza = PizzaFactory.create_pizza("Margherita")


    # Adding toppings dynamically is very important! ! !
    pizza_with_toppings = Cheese(base_pizza)
    pizza_with_toppings = Mushroom(pizza_with_toppings)
    pizza_with_toppings = Pepper(pizza_with_toppings)


    # Output final pizza description and cost
    print(pizza_with_toppings.description())  # Output: Margherita, Cheese, Mushroom, Pepper
    print(pizza_with_toppings.cost())          # Output: 12.5
```

---

5. Elevator System (Frequency: 4 / 5)

Problem: Simulate an elevator system that handles multiple requests, directions, and floors.

Concepts: State management, scheduling

```
from collections import deque

import heapq

import time

from enum import Enum


class State(Enum):

    IDLE = 1

    UP = 2

    DOWN = 3



class RequestOrigin(Enum):

    INSIDE = 1

    OUTSIDE = 2


class Request:

    def __init__(self, origin, origin_floor, destination_floor=None):

        self.origin = origin

        self.origin_floor = origin_floor
```

```
        self.destination_floor = destination_floor

    def __lt__(self, other):
        return self.destination_floor < other.destination_floor  #
class Elevator:
    def __init__(self, current_floor=1):
        self.current_floor = current_floor  # current floor
        self.state = State.IDLE  # begin state
        self.up_queue = []  # minheap
        self.down_queue = []  # minheap

    # open elevator
    def open_doors(self):
        print(f"Doors are OPEN on floor {self.current_floor}")

    # close elevator
    def close_doors(self):
        print(f"Doors are CLOSED")

    # up request to queue
    def add_up_request(self, request):
```

```
        heapq.heappush(self.up_queue, request)


    # down request to queue
    def add_down_request(self, request):
        heapq.heappush(self.down_queue, request)


    def process_up_requests(self):
        while self.up_queue:
            request = heapq.heappop(self.up_queue)
            self.move_to_floor(request.destination_floor)


    def process_down_requests(self):
        while self.down_queue:
            request = heapq.heappop(self.down_queue)
            self.move_to_floor(request.destination_floor)


    def move_to_floor(self, floor):
        if self.current_floor != floor:
            print(f"Moving from floor {self.current_floor} to floor
{floor}")
            time.sleep(1)  # 模拟移动时间
```

```
            self.current_floor = floor

            print(f"Arrived at floor {floor}")

        self.open_doors()

        time.sleep(1)   # 模拟开门时间

        self.close_doors()


    def operate(self):

        if self.up_queue or self.state == State.UP:

            print("Processing UP requests...")

            self.process_up_requests()

        if self.down_queue or self.state == State.DOWN:

            print("Processing DOWN requests...")

            self.process_down_requests()

        self.state = State.IDLE  # done and idle

        print("Elevator is now IDLE.")


class Controller:

    def __init__(self):

        self.elevator = Elevator()

    def send_up_request(self, origin_floor, destination_floor):
```

```
        request = Request(RequestOrigin.OUTSIDE, origin_floor, 
destination_floor)
        self.elevator.add_up_request(request)


    def send_down_request(self, origin_floor, destination_floor):
        request = Request(RequestOrigin.OUTSIDE, origin_floor, 
destination_floor)
        self.elevator.add_down_request(request)


    # start processing
    def handle_requests(self):
        self.elevator.operate()


class Main:
    @staticmethod
    def main():
        controller = Controller()


        # up and down
        controller.send_up_request(1, 5)
        controller.send_down_request(4, 2)
```

```
        controller.send_up_request(3, 6)


        # process requests

        controller.handle_requests()


        print("New requests...")

        controller.send_up_request(1, 9)

        controller.send_down_request(5, 2)


        # process new requests

        controller.handle_requests()


if __name__ == "__main__":

    Main.main()
```

---

6. Amazon Locker System (Frequency: 4 / 5)

Problem: Simulate a locker storage system with size-based allocation and expiration.

Concepts: Object mapping, inventory

```
from enum import Enum

from collections import defaultdict
```

```
import random

import string


# how to implement the locker size?

# how to implement the locker status?

# how to implement the locker id?


class LockerSize(Enum):

    SMALL = 1

    MEDIUM = 2  # 20x20x20

    LARGE = 3   # 30x30x30




class LockerStatus(Enum):

    AVAILABLE = 1

    OCCUPIED = 2

    EXPIRED = 3




class Locker:

    def __init__(self, locker_id, size):
```

```python
        self.locker_id = locker_id

        self.size = size

        self.status = LockerStatus.AVAILABLE

        self.package = None  # Stores package object when occupied

        self.code = None  # Unique code for retrieving package


    def assign_package(self, package, code):

        self.status = LockerStatus.OCCUPIED

        self.package = package

        self.code = code


    def release_package(self):

        self.status = LockerStatus.AVAILABLE

        self.package = None

        self.code = None




class Package:

    def __init__(self, package_id, size):

        self.package_id = package_id

        self.size = size
```

```
class CodeGenerator:
    @staticmethod
    def generate_code():
        return ''.join(random.choices(string.ascii_uppercase +
string.digits, k=6))


class LockerManager:
    def __init__(self):
        self.lockers = defaultdict(list)  # Lockers grouped by size
        self.code_to_locker = {}  # Map retrieval code to Locker

    def add_locker(self, locker):
        self.lockers[locker.size].append(locker)

    def find_available_locker(self, package_size):
        for locker in self.lockers[package_size]:
            if locker.status == LockerStatus.AVAILABLE:
                return locker
```

```
        return None


    def assign_package_to_locker(self, package):

        locker = self.find_available_locker(package.size)

        if not locker:

            raise Exception("No available locker for the package
size.")

        code = CodeGenerator.generate_code()

        locker.assign_package(package, code)

        self.code_to_locker[code] = locker

        return code


    def retrieve_package(self, code):

        if code not in self.code_to_locker:

            raise Exception("Invalid code or package already
retrieved.")

        locker = self.code_to_locker[code]

        if locker.status != LockerStatus.OCCUPIED:

            raise Exception("Locker is not occupied.")

        package = locker.package

        locker.release_package()
```

```python
        del self.code_to_locker[code]

        return package



# Example usage:

if __name__ == "__main__":

    # Initialize lockers

    manager = LockerManager()

    manager.add_locker(Locker("L1", LockerSize.SMALL))

    manager.add_locker(Locker("L2", LockerSize.MEDIUM))

    manager.add_locker(Locker("L3", LockerSize.LARGE))


    # Assign package to locker

    package1 = Package("P1", LockerSize.SMALL)

    code = manager.assign_package_to_locker(package1)

    print(f"Package assigned to locker with code: {code}")


    # Retrieve package

    retrieved_package = manager.retrieve_package(code)

    print(f"Retrieved package ID: {retrieved_package.package_id}")
```

7. Design Amazon (Frequency: 3 / 5)

Problem: ''' amazon should have

1. user information

2. cart information

3. order information

4. payment information

5. product information

user can have 1 cart and many orders.

cart can have many products.

every order can have 1 payment.

brainstorm:

user should have id, name, email, and a cart and previous orders.

cart should have products.

products should have stock of products.

order should have quantity. '''

Concepts: **Entity Modeling, Single-Responsibility / SOLID, State-Lifecycle Management**

```
class User:
```

```
    def __init__(self, userID, name, email):
        self.cart = Cart(self)  #we will create a cart class later
        self.userID = userID
        self.name = name
        self.email = email
        self.orders = []

    def addtocart(self, cart, product, quantity):
        self.cart.addproduct(product, quantity) #we will create an add product function in a product class

    def removefromcart(self, cart, product, quantity):
        self.cart.removeproduct(product, quantity) #we will create a remove product function

    def placeorder(self):
        order = self.Order(self.cart.products)  #we will create an order later
        self.orders.append(order)
        self.cart.emptycart() #we will create empty cart function in cart
        return order
```

```
    def vieworder(self):
        return self.orders


class Product:
    def __init__(self, productID, name, price, stock):
        self.productID = productID
        self.name = name
        self.price = price
        self.stock = stock


class Cart:
    def __init__(self, user):
        self.user = user
        self.products = {}

    def addproduct(self, product, quantity):
        if quantity > product.stock:
            print(f'There are only {product.stock} left.')
```

```
        else:
            self.products[product] = quantity +
self.products.get(product, 0)
            product.stock -= quantity


    def removeproduct(self, product, quantity):
        if quantity > product.stock:
            print(f'There are only {product.stock} in your cart.')
        else:
            self.products[product] -= quantity
            if self.products[product] == 0:
                del self.products[product]
            product.stock += quantity


    def viewcart(self):
        return self.products


    def emptycart(self):
        self.products.clear()
```

```python
class Order:

   ordercount = 0

   def __init__(self, products, orderid):

       self.orderid = orderid

       Order.ordercount += 1

       self.products = products

       self.status = 'Placed'




class Payment:

   def __init__(self, order, amount, paymenttype):

       self.order = order

       self.amount = amount

       self.paymenttype = paymenttype

       self.status = 'Pending'


   def processpayment(self):

       self.status = 'Completed'
```

8. Lowest Common Ancestor in Organization Chart (Frequency: 3 / 5)

Problem: Find the lowest common manager in a company org chart.

Concepts: N-ary tree traversal, recursive DFS

```
class Employee:
    def __init__(self, name):
        self.name = name
        self.subordinates = []

    def add_subordinate(self, emp):
        self.subordinates.append(emp)

def find_lca(root, emp1, emp2):
    if root is None or root == emp1 or root == emp2:
        return root

    count = 0
    temp = None
    for sub in root.subordinates:
        res = find_lca(sub, emp1, emp2)
        if res:
            count += 1
            temp = res

    if count == 2:
```

```
    return root
  return temp
```

---

4. Design Interview Techniques

- Emphasize class responsibilities
- State assumptions clearly
- Think out loud during design
- Discuss how you'd extend the system if requirements change
- Address concurrency and scalability

---

5. How to Use This Guide

- Rebuild each example from scratch, then optimize it
- Use it to prep for Amazon, Meta, Google, and startups
- Create your own variants to test extensibility

Want to go further? Add logging, unit tests, error handling, and performance notes to these skeletons. That's what interviewers love to see.