Low-Level Design in a Hurry

OOP Concepts

A list of OOP concepts for low-level design interviews.

Where the **design principles** page taught you how to think about clean code, OOP concepts are the mechanisms your language gives you to actually implement those ideas.

We'll assume you already know how to program. This page is a focused refresher on the parts that actually matter in interviews. We'll walk through the four core concepts: encapsulation, abstraction, polymorphism, and inheritance. For each, you'll be reminded what it is, why interviewers care, and how it shows up in real LLD problems.

Encapsulation

Encapsulation means keeping an object's data private and letting the object control how that data is used. You interact with it through simple methods instead of reaching in and changing its internal details yourself.

The benefit is predictability. When your `Account` class owns its balance field and only lets you modify it through `deposit()` and `withdraw()`, you can enforce rules in those methods. You can prevent negative balances, log transactions, update related state. If the balance was public and anyone could write to it directly, you'd have no guarantee those rules get followed.

In interviews, encapsulation shows up as a basic hygiene check. Do your classes expose their fields directly, or do they provide methods? Are you returning references to mutable internal collections that callers can modify, or are you returning copies?

By leaving `spots` public, we're allowing anyone to modify it directly.

Bad: No Encapsulation — Python

```python
class ParkingSpot:
    def occupy(self, vehicle: "Vehicle") -> None:
        ...


class Vehicle:
    def __init__(self, type_: str):
        self.type = type_


class ParkingLot:
    def __init__(self):
```

```
        self.spots: list[ParkingSpot] = []  # public, mutable
```

Instead, we can make `spots` private and provide a method to add a new spot since that's the only way to modify the list.

Good: Proper Encapsulation — Python

```python
from typing import Optional


class ParkingSpot:
    def occupy(self, vehicle: "Vehicle") -> None:
        ...


class Vehicle:
    def __init__(self, type_: str):
        self.type = type_


class ParkingLot:
    def __init__(self):
        self._spots: list[ParkingSpot] = []

    def park_vehicle(self, vehicle: Vehicle) -> bool:
        spot = self._find_available_spot(vehicle)
        if spot is None:
            return False
```

> If you're designing a class and wondering whether to expose a field or write a getter, write the getter. If you need to return a collection, return an unmodifiable view or a copy.

Abstraction

Abstraction means exposing only what's essential and hiding implementation details behind clear interfaces. You define what something can do without revealing how it does it.

The benefit is simplification. An abstraction hides complexity. When your payment processing code depends on a `PaymentMethod` interface instead of concrete classes like `CreditCardProcessor` or `PayPalProcessor`, you can swap implementations without touching the code that uses them. The caller doesn't need to know whether you're hitting Stripe's API or storing payment tokens in a database. It just calls `process()` and gets a result.

Abstraction typically appears where there's complexity in your system. When you encounter a complicated area of logic or state - something with lots of variations, rules, or messy details - abstractions help simplify it. By defining a clear interface or contract, you can hide those details and make the rest of your code easier to reason about. In interviews, look for places where the logic feels tangled or the requirements suggest multiple approaches - those are good signals that introducing an abstraction will make things more manageable.

In this example, we tightly coupled the `OrderService` to the `StripeAPI` implementation, making changes to the payment system would require modifying the `OrderService` class.

Bad: No Abstraction — Python

```python
class Order:
    def __init__(self, total: float, credit_card: str):
        self.total = total
        self.credit_card = credit_card


class StripeAPI:
    def set_api_key(self, key: str) -> None:
        ...

    def create_charge(self, amount: float, card: str) -> None:
        ...


class OrderService:
    def __init__(self, api_key: str):
        self.api_key = api_key

    def checkout(self, order: Order) -> None:
```

Much better, we can rely on an abstraction like `PaymentMethod` to handle the different payment methods.

Good: Proper Abstraction — Python

```python
from abc import ABC, abstractmethod


class PaymentMethod(ABC):
    @abstractmethod
    def process(self, amount: float) -> bool:
        ...


class CreditCardPayment(PaymentMethod):
    def process(self, amount: float) -> bool:
        return True
```

```python
class PayPalPayment(PaymentMethod):
    def process(self, amount: float) -> bool:
        return True


class Order:
    def __init__(self, total: float, credit_card: str):
        self.total = total
        self.credit_card = credit_card
```

The interface defines the contract (`process(amount)`), and each implementation handles the details. OrderService doesn't care which one it gets.

> The hard part is choosing the right level of abstraction. Too abstract and your interface becomes meaningless (`doWork()` , `handleRequest()`). Too specific and you haven't actually abstracted anything. Think about what operations the caller needs to perform, not how those operations happen internally.

Polymorphism

Polymorphism is what replaces `if (type == "credit")` or `switch (vehicleType)` statements. Instead of checking types, you call the same method and let each object handle itself. Different objects respond to the same action in their own way.

> Highly polymorphic code can be difficult to trace and debug, especially as the number of implementations grows. Each company has a different tolerance for polymorphism. Some prefer clear, explicit branches for each type, while others embrace the extensibility polymorphism offers. In interviews, always be ready to explain the tradeoff. Polymorphism offers flexibility and extensibility, but it can also make code flows less obvious and harder to follow when debugging or onboarding new engineers.

Polymorphism naturally follows from abstraction. Once you define an interface like `PaymentMethod` or `Vehicle` , each implementation can provide its own behavior. When you call a method on an interface, the actual implementation that runs depends on the concrete type you're working with. Each type knows how to handle itself. No type checking required.

Bad: No Polymorphism — Python

```python
from typing import Optional


class ParkingSpot:
    pass
```

```python
class Vehicle:
    def __init__(self, type_: str):
        self.type = type_


class ParkingLot:
    def park_vehicle(self, vehicle: Vehicle) -> bool:
        if vehicle.type == "car":
            spot = self._find_spot_by_size("regular")
            return spot is not None
        elif vehicle.type == "motorcycle":
            spot = self._find_spot_by_size("motorcycle")
            return spot is not None
        elif vehicle.type == "truck":
```

Good: Using Polymorphism

Python

```python
from enum import Enum
from typing import Optional


class SpotSize(Enum):
    REGULAR = "regular"
    MOTORCYCLE = "motorcycle"
    LARGE = "large"


class ParkingSpot:
    pass


class Vehicle:
    def get_required_spot_size(self) -> SpotSize:
        raise NotImplementedError


class Car(Vehicle):
    def get_required_spot_size(self) -> SpotSize:
        return SpotSize.REGULAR
```

Now when you add a new vehicle type, you just create a new class that implements `Vehicle`. The `ParkingLot` code never changes.

> Use polymorphism when behavior varies by type. If you see yourself writing type checks or switch statements on an enum, that's a sign you should be using polymorphism instead.

Inheritance

Inheritance lets one class be a more specific version of another, automatically getting the parent's data and behavior. It's a tool for sharing implementation, but it comes with a big cost: tight coupling.

When a subclass inherits the parent's fields and methods, any change in the parent can break every child. That's the "fragile base class" problem, and it's why inheritance often creates more rigidity than it solves.

A safer alternative is composition + interfaces. An interface defines the behavior, and each class implements it independently. You still get abstraction and polymorphism, but without forcing classes into a parent-child relationship or sharing state they shouldn't.

When Inheritance Works

Inheritance makes sense when you have stable, shared implementation that multiple subclasses genuinely need. Bank accounts are a good example. A `SavingsAccount` and `CheckingAccount` both track balances, handle deposits and withdrawals, and maintain transaction history. That logic is identical across all account types.

Good: Inheritance for Shared Implementation — Python

```python
class BankAccount:
    def __init__(self):
        self.balance = 0.0

    def deposit(self, amount: float) -> None:
        self.balance += amount

    def withdraw(self, amount: float) -> bool:
        if self.balance < amount:
            return False
        self.balance -= amount
        return True

    def get_balance(self) -> float:
        return self.balance


class SavingsAccount(BankAccount):
    def __init__(self, interest_rate: float):
        super().__init__()
        self.interest_rate = interest_rate


class CheckingAccount(BankAccount):
```

Here, the shared implementation is stable and meaningful. Both subclasses genuinely are forms of `BankAccount`, and they don't need to override the inherited behavior in ways that break the parent's contract. So inheritance is a good fit.

When Inheritance Breaks Down

The classic mistake made in interviews is using inheritance to model behavior differences. If subclasses need to override methods to provide completely different implementations, that's a sign you're using the wrong tool.

Bad: Inheritance for Behavior Variation — Python

```python
class Car:
    def start_engine(self) -> None:
        # gasoline engine start logic
        ...


class ElectricCar(Car):
    def start_engine(self) -> None:
        # electric motor startup logic - completely different
        ...
```

Electric cars don't have engines. They don't share useful engine logic. You're forcing a behavior difference into a class hierarchy, which creates fragile code. When you add a hybrid car, do you extend `Car` or `ElectricCar`? Neither works cleanly.

When behavior varies, the better approach is to isolate that behavior into its own abstraction and compose it.

Good: Composition for Behavior Variation — Python

```python
from abc import ABC, abstractmethod


class Drivetrain(ABC):
    @abstractmethod
    def start(self) -> None:
        ...


class GasEngine(Drivetrain):
    def start(self) -> None:
        # gas engine startup logic
        ...
```

```
class ElectricMotor(Drivetrain):
    def start(self) -> None:
        # electric motor startup logic
        ...


class Car:
    def __init__(self, drivetrain: Drivetrain):
```

Now you can model any kind of car without breaking the hierarchy or overriding logic awkwardly. Want a hybrid? Give it two drivetrains. Want a hydrogen car? Add a new `Drivetrain` implementation. The `CarWithDrivetrain` class never changes.

> For your interview, default to interfaces with composition. Only use inheritance when you genuinely need to share implementation across classes and the relationship is stable. In most LLD interviews, you don't need inheritance at all.

Putting It Together

Just like with design principles, you don't need to recite these terms during your interview. If you forget the word "polymorphism," it doesn't matter. What matters is that when you see requirements like "support multiple payment methods," you know to define an interface. When you're designing a class, you know to keep fields private and expose methods. When you see yourself writing type checks, you know to use an interface instead.

The concepts show through in how you design, not in what you name. Focus on applying them naturally:

- **Encapsulation**: Hide state, expose behavior. Make fields private, provide methods for access
- **Abstraction**: Define interfaces for variations. Multiple payment methods? Different vehicle types? Create an interface
- **Polymorphism**: Let objects handle themselves. No type checking, no switch statements on types
- **Inheritance**: Compose behavior, don't inherit it. Reach for interfaces first, use inheritance only when sharing stable implementation

Test Your Knowledge

Take a quick 15 question quiz to test what you've learned.

Start Quiz

How would you rate the quality of this article?

Login To Join The Discussion

Your account is free and you can post anonymously if you choose.

Search 14 comments

Sort By
Popular

Phuoc Nguyen

Top 10% • 1 month ago

Very cool! Thanks for new section. Do you have any plan for fundamentals: OS, Process/Thread, Security, Linux, Docker, Git,...?

6

M

MagicAquaRooster979

Premium • 1 month ago

Could the typing be updated to modern Python standards:

```
from typing import Optional
# you don't need List anymore
...
      self.spots: list[ParkingSpot]
...
```

3

Evan King

Admin • 1 month ago

Good shout, I'll update throughout

1

M

MagicAquaRooster979

Premium • 1 month ago

> *@property*
> *def spots(self) -> List[ParkingSpot]:*
> *return list(self._spots)*

This is a python specific thing, but again in an interview setting a candidate putting in @property shows they understand the language. Also in general maybe avoid using get_*** in the getters for python.

2

A

SunflowersInaVase

Premium • 16 days ago

In the follow-up question to Composition over Inheritance (`Car` + `DriveTrain`) for creating a hybrid car: it is suggested "*to give it two drivetrains*".
Will it better (or clearer) to say the new HybridDriveTrain will use the 2 `GasEngine` and `ElectricMotor` drivetrains?

Also typo: The `CarWithDrivetrain` class never changes. -> The `Car` class never changes.

1

A

SunflowersInaVase

Premium • 16 days ago

In the (good way) example of Polymorphism, making `Vehicle` an Interface (ABC in python) instead of a base class that throws `NotImplementedError` for `get_required_spot_size` would be better i think. Would also help in supporting the stmt: *Polymorphism naturally follows from abstraction*

1

Show All Comments

Learn Behavioral

Learn ML System Design

Learn Low Level Design

Guided Practice

Links

FAQ

Pricing

Gift Mock Interviews

Gift Premium

Become a Coach

Our Coaches

Hello Interview Premium

Legal

Terms and Conditions

Privacy Policy

Contact

About Us

Product Support

7511 Greenwood Ave North
Unit #4238 Seattle
WA 98103