Low-Level Design in a Hurry

Design Principles

A list of design principles for low-level design interviews.

Design principles guide your decision making to create clean, extensible, and maintainable code. When you're designing a parking lot system or a chess game in an interview, you'll constantly face decisions: Should this be a separate class? Should I use inheritance here? Is this abstraction worth it? Design principles give you a framework to make those calls and explain them.

This list of principles can feel overwhelming, but these are all concepts you learned in school and likely use everyday in your job. So don't stress over memorizing acronyms or listing off SOLID like it's the alphabet. Interviewers care that you apply the lessons, not that you can name them.

There are two categories of principles worth knowing: **General software design principles** and **object-oriented design principles**. Both show up in interviews, but OOD principles get more attention because most LLD problems expect you to design class hierarchies.

While you can find endless lists of design principles online, we've distilled them down to the most important ones for LLD interviews.

General Software Design Principles

If you only remember three general software design principles from this guide, make it KISS, DRY, and YAGNI. Those three will carry you through most interviews.

KISS - Keep It Simple, Stupid

The simplest solution that works is usually the right one. When you're designing a class or choosing between patterns, pick the straightforward approach. If you can solve the problem with a simple conditional instead of a strategy pattern, do that. If a single class handles the job without getting messy, don't split it up.

This is the single principle we see most often violated in low-level design interviews. Candidates often over-engineer because they want to show off their knowledge of design patterns. They'll introduce factories, builders, and decorators when a basic class would work fine. Interviewers notice this. They want to see that you can distinguish between problems that need sophisticated solutions and problems that need simple ones.

The time to add complexity is when simplicity stops working. If your single class grows to 500 lines with ten different responsibilities, that's when you refactor. If adding a new payment method means modifying code in five places, that's when you introduce a **strategy pattern**. But start simple.

DRY - Don't Repeat Yourself

When you find yourself writing the same logic in multiple places, pull it into one place. If three classes all validate email addresses the same way, create a shared validation method. If two services both need to convert timestamps, put that conversion in a utility function.

The benefit is maintenance. When the email validation rules change, you update one method instead of hunting through your codebase for every place you duplicated the logic. When a bug exists in the timestamp conversion, you fix it once.

But don't take DRY too far. If two pieces of code look similar but serve different purposes, sometimes duplication is fine. Forcing them to share code can create artificial coupling where changes to one break the other. The key is whether the logic is conceptually the same, not just textually similar.

DRY also conflicts with KISS. Sometimes the simplest solution is to duplicate code in two places rather than build an abstraction. There's no right answer, and showing you understand this tradeoff is what separates senior candidates. The right move in an interview is to acknowledge both sides: "I expect this validation logic to appear in multiple places, but I'm going to start by keeping it in the User class to avoid adding unnecessary complexity early. If we see it duplicated three or four times, we can pull it into a shared validator." This shows you can balance competing principles instead of blindly following rules.

YAGNI - You Aren't Gonna Need It

Build what you need now, not what you might need later. In interviews, when you're designing a parking lot system, don't add support for valet parking and electric vehicle charging stations unless the requirements specifically mention them. Don't make your classes extensible in every direction just in case.

The problem with building for future requirements is you usually guess wrong. You add complexity for scenarios that never happen, and when the actual new requirement comes, it's different from what you prepared for. Now you're stuck maintaining dead code.

> This principle doesn't mean "never think ahead" - it means don't build ahead. Design with extension in mind, but only implement what's needed now.

This comes up when interviewers ask "how would you extend this?" That's your cue to talk about how you'd modify the design if new requirements appeared. But in your initial design, stick to what's actually needed.

Separation of Concerns

Different parts of your code should handle different responsibilities, and they shouldn't know about each other's internals. Your UI layer shouldn't contain business logic. Your business logic shouldn't know how data is stored. Your data access layer shouldn't format strings for display.

Take a look at this example where we mix display logic, input handling, and game rules all in one method.

Bad: Violates Separation of Concerns

Python

```python
class TicTacToe:
    def __init__(self):
        self.board = [["" for _ in range(3)] for _ in range(3)]

    def play(self):
        while True:
            # Display mixed with game logic
            for row in self.board:
                print(row)

            # Input handling mixed in
            row = int(input())
            col = int(input())
            self.board[row][col] = "X"

            # Win checking mixed in
            if (
                self.board[0][0] == self.board[1][1]
                and self.board[1][1] == self.board[2][2]
            ):
                print("Winner!")
                break
```

Instead, we can separate each responsibility into its own classes so that we have `Board`, `Display`, and `InputHandler` classes all with their own responsibilities.

Good: Follows Separation of Concerns

Python

```python
class TicTacToe:
    def __init__(self, board, display, input_handler):
        self.board = board
        self.display = display
        self.input_handler = input_handler

    def play(self):
        while not self.board.has_winner():
            self.display.render(self.board)
            move = self.input_handler.get_next_move()
            self.board.make_move(move)
        self.display.show_winner(self.board.get_winner())
```

Now if you want to switch from console input to a GUI, you only touch InputHandler. If you want to change how the board displays, you only modify Display. If you need to add new win conditions, you only update Board. Each change is isolated to one class. This is what lets you test each part of the system independently.

Law of Demeter

Also called the principle of least knowledge. A method should only talk to its immediate friends, not reach through objects to access distant parts of the system. If you see code like `order.getCustomer().getAddress().getZipCode()`, that's violating the Law of Demeter.

The problem with deep chaining is coupling. Your code now knows the internal structure of three different objects. If any of them change how they organize their data, your code breaks. Instead, put a method on `Order` called `getCustomerZipCode()` that handles the navigation internally.

> Method chaining itself is not the problem. Fluent interfaces like `builder.setName("John").setAge(30).build()` are fine because they return the same object type. The issue is specifically when chaining leaks internal structure by traversing through multiple different object types.

In interviews, this comes up when you're defining class methods. Instead of returning complex objects that callers need to dig through, return the specific data they need or provide higher-level methods that do the work.

Object-Oriented Design Principles (SOLID)

These principles are grouped under the acronym SOLID and apply specifically when you're designing classes and their relationships. They show up constantly in LLD interviews because most problems expect you to design class hierarchies.

> SOLID principles come from Java's heyday of deep inheritance hierarchies and interface-heavy design. Outside of Java and C#, excessive application of SOLID is falling out of fashion. Modern languages favor simpler

approaches—composition over class hierarchies, functions over interfaces. Don't break KISS by forcing SOLID patterns where simpler solutions work fine. In interviews, apply these principles when the problem calls for them, but recognize when you're adding complexity for its own sake.

SRP - Single Responsibility Principle

A class should have one reason to change. If a class mixes multiple concerns, split them. This is the foundation of good class design.

Take a look at this `Report` class that handles content generation, PDF formatting, and file storage all in one place:

Bad: Violates SRP   Python

```python
class Report:
    def generate_content(self) -> str:
        return "content"

    def print_to_pdf(self) -> None:
        # PDF formatting
        pass

    def save_to_file(self) -> None:
        # file I/O
        pass
```

Instead, we can split these responsibilities into separate classes:

Good: Follows SRP   Python

```python
class Report:
    def generate_content(self) -> str:
        return "content"


class PDFPrinter:
    def print(self, report: Report) -> None:
        # PDF formatting
        pass


class FileStorage:
    def save(self, content: str) -> None:
        # file I/O
        pass
```

Now when the PDF formatting library changes, you only touch `PDFPrinter`. When you switch from files to a database, you only modify `FileStorage`. When the report content logic changes, you only update `Report`. Each change is isolated.

OCP - Open/Closed Principle

Classes should be open for extension but closed for modification. You should be able to add new behavior without changing existing code. This usually means using interfaces or abstract classes so you can add new implementations without touching the original code.

Every time you modify existing code, you risk breaking things that already work. If you design with interfaces from the start, adding new functionality becomes a matter of writing new classes that implement those interfaces. The old code never changes, so it can't break.

Take a look at this `PaymentProcessor` that requires modification every time we add a new payment type:

Bad: Violates OCP Python

```python
class PaymentProcessor:
    def process(self, payment_type: str, amount: float) -> None:
        if payment_type == "credit":
            # credit card logic
            pass
        elif payment_type == "paypal":
            # paypal logic
            pass
        # Adding crypto means modifying this method
```

Instead, we can use interfaces to allow adding new payment types without modifying existing code:

Good: Follows OCP Python

```python
from abc import ABC, abstractmethod


class PaymentMethod(ABC):
    @abstractmethod
    def process(self, amount: float) -> None:
        ...


class CreditCardPayment(PaymentMethod):
    def process(self, amount: float) -> None:
        # credit card logic
        pass


class PayPalPayment(PaymentMethod):
    def process(self, amount: float) -> None:
        # paypal logic
        pass


class CryptoPayment(PaymentMethod):
    def process(self, amount: float) -> None:
```

Now when you need to add cryptocurrency payments, you just create a new `CryptoPayment` class. The existing `PaymentProcessor` code never changes.

LSP - Liskov Substitution Principle

Subclasses must work wherever the base class works. If you have a method that accepts a `Bird`, passing in a `Penguin` shouldn't break things even though penguins can't fly. This means your subclasses can't violate the expectations set by the parent class.

Said differently, if your code uses a parent class or interface, it should be able to use any subclass without knowing which specific subclass it is. The subclass can add new behavior, but it can't remove or break behavior that the parent promised. When a subclass throws an exception for a method the parent class provides, that's a red flag you're violating LSP. If a subclass forces callers to add special-case logic (e.g., `if (bird instanceof Penguin)`), you violated LSP.

Take a look at this classic example where `Penguin` extends `Bird` but breaks the expectation that all birds can fly:

Bad: Violates LSP Python

```python
class Bird:
    def fly(self) -> None:
        # flying logic
        pass
```

```python
class Penguin(Bird):
    def fly(self) -> None:
        raise NotImplementedError("Penguins can't fly")
```

Instead, we can separate the flying behavior into its own interface so only birds that can actually fly need to implement it:

Good: Follows LSP

Python

```python
from abc import ABC, abstractmethod


class Bird(ABC):
    @abstractmethod
    def eat(self) -> None:
        ...


class FlyingBird(Bird):
    @abstractmethod
    def fly(self) -> None:
        ...


class Sparrow(FlyingBird):
    def eat(self) -> None:
        pass

    def fly(self) -> None:
        pass
```

This comes up in interviews when you're designing class hierarchies. Think carefully about what methods belong in the base class versus subclasses.

ISP - Interface Segregation Principle

Prefer small, focused interfaces over large, general-purpose ones. Don't force classes to implement methods they don't need. If a class only needs two methods from an interface with ten methods, that interface is too big.

The problem with fat interfaces is that classes are forced to implement methods they'll never use. This leads to empty implementations or methods that throw exceptions, which is a code smell. Split large interfaces into smaller, cohesive ones. Classes can implement multiple small interfaces if they need to, but they're not stuck implementing irrelevant methods.

Bad: Violates ISP

Python

```python
class Worker:
    def work(self) -> None:
        ...

    def eat(self) -> None:
        ...

    def sleep(self) -> None:
        ...


class Robot(Worker):
    def work(self) -> None:
        pass

    def eat(self) -> None:
        # robots don't eat
        pass
```

Good: Follows ISP

Python

```python
class Workable:
    def work(self) -> None:
        ...


class Feedable:
    def eat(self) -> None:
        ...


class Restable:
    def sleep(self) -> None:
        ...


class Human(Workable, Feedable, Restable):
    def work(self) -> None:
        pass

    def eat(self) -> None:
        pass
    def sleep(self) -> None:
```

DIP - Dependency Inversion Principle

High-level modules shouldn't depend on low-level modules. Both should depend on abstractions. This means your business logic shouldn't directly instantiate concrete classes - it should depend on interfaces.

Instead of your `NotificationService` creating a new `EmailSender` directly, it should receive a `MessageSender` interface through its constructor. This makes your code more flexible and testable. You can swap out email for SMS without changing `NotificationService` at all. In interviews, this comes up when discussing dependency injection and how to make your design flexible.

Take a look at this `NotificationService` that's tightly coupled to a specific email implementation.

Bad: Violates DIP   Python

```python
class EmailSender:
    def send(self, message: str) -> None:
        # send email
        pass


class NotificationService:
    def __init__(self) -> None:
        self.email_sender = EmailSender()

    def notify(self, message: str) -> None:
        self.email_sender.send(message)
```

Instead, we can depend on an abstraction and inject the specific implementation through the constructor.

Good: Follows DIP   Python

```python
from abc import ABC, abstractmethod


class MessageSender(ABC):
    @abstractmethod
    def send(self, message: str) -> None:
        ...
```

```python
class EmailSender(MessageSender):
    def send(self, message: str) -> None:
        # send email
        pass


class NotificationService:
    def __init__(self, sender: MessageSender) -> None:
        self.sender = sender

    def notify(self, message: str) -> None:
        self.sender.send(message)
```

Now you can swap email for SMS without changing NotificationService.

Putting It All Together

Remember, you don't need to name these principles constantly. Use them to guide your decisions and reference them briefly when explaining tradeoffs. The principles are simply tools for thinking, rather than a checklist to recite.

Here is a quick cheat sheet for the principles you should know.

General Principles

- KISS → Start simple, add complexity only when needed
- DRY → Reduce duplication, simplify maintenance
- YAGNI → Build for today, not hypothetical futures
- Separation of Concerns → Enable independent testing and changes
- Law of Demeter → Reduce coupling, hide internal structure

SOLID Principles

- SRP → Keep classes focused on one responsibility
- OCP → Support future requirements without modifying existing code
- LSP → Prevent brittle hierarchies that break at runtime
- ISP → Keep interfaces clean and focused
- DIP → Keep code flexible and testable through abstraction

Focus on the reasoning behind your choices. The principles will show through naturally.

Test Your Knowledge

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

Start Quiz

Login to track your progress

Next: OOP Concepts →

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 15 comments

Sort By
Popular

krishnakanth eswaran
Premium • 1 month ago
Very informative! It would be helpful if you could provide some real world production systems like examples for these sort of situations, would be more helpful to convey in interviews!

14

Ashish khare
Premium • 19 days ago
i agree

1

Hamad Khan
★ Top 5% • 1 month ago
FIRST

8

HARDIK SHARMA
★ Top 10% • 23 days ago
This aint youtube buddy!

9

Hamad Khan
★ Top 5% • 1 month ago
ok but actually thanks for the content

5

J

Justin Duda

Premium • 1 month ago

> *A class should have one reason to change*

?

1

N

Naman Dhanotia

• 27 days ago

It means the class should have only 1 responsibility.

1 responsibility = 1 reason to change, in case of any change in logic.

If more responsibilities then class can change due to many reasons.

0

A

Alex

Premium • 1 month ago

> *This means your subclasses can't violate the expectations set by the parent class.*

This sentence use negative wording ("can't"). It would be clearer and stronger to phrase it positively, for example: "This means your subclasses must follow the expectation set by the parent class."

1

K

Kavish Khullar

Premium • 30 days ago

I like the negative wording, easier to remember.

1

U

UnderlyingYellowCat786

Premium • 7 days ago • edited 7 days ago

It would be great if you could cover GRASP principles. Some of patterns are outdated but I feel some are still pretty useful in daily work (information expert, creator, etc).

0

Show All Comments

Schedule a mock interview

Meet with a FAANG senior+ engineer or manager and learn exactly what it takes to get the job.

Schedule A Mock Interview

Questions

Meta SWE Interview Questions

Amazon SWE Interview Questions

Google SWE Interview Questions

OpenAI SWE Interview Questions

Engineering Manager (EM) Interview Questions

Learn

Learn System Design

Learn DSA

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