Low-Level Design in a Hurry

Design Patterns

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

Design patterns are reusable building blocks for solving common design problems. They're names for structures you naturally create when you follow solid design principles.

The Gang of Four catalog, written in 1994 for C++ and Smalltalk, defined 23 patterns that became required reading for software engineers. The book earned legendary status, but the reality is most of those patterns don't matter anymore. Modern languages have built-in features that replaced half of them ie. iterators are primitives now, not patterns. The shift from inheritance-heavy OOP to composition and functional programming made others obsolete. And in interviews, you'll get asked about maybe five patterns total, not twenty-three.

Most online resources still dutifully list all 23 GoF patterns like they're equally important. They're not. We're cutting the historical baggage and focusing on the patterns that actually show up in modern LLD interviews. The ones below are what interviewers ask about and what you'll use in real designs.

As you learn these patterns, keep in mind that the goal isn't to force them into every solution. You should only use a pattern when the problem naturally calls for it. Forcing one is a common mistake that signals over-engineering. Patterns arise from good design decisions rather than driving them, and the most frequent error is trying to fit a pattern where it doesn't belong.

> Design patterns are handled differently depending on where you interview. In the US, most LLD interviews don't explicitly test whether you can name patterns—you're evaluated on the quality of your design. In other parts of the world, particularly India, interviewers are more likely to ask about patterns directly. While we align with the former, both approaches have merit, and we want you prepared for either reality you may encounter.

Patterns fit cleanly into three categories: creational, structural, and behavioral. Let's walk through each category together.

Creational Patterns

Creational patterns control how objects get created. They hide construction details, let you swap implementations, and keep your code from being tightly coupled to specific classes.

Factory Method

A factory is a helper that makes the right kind of object for you so you don't have to decide which one to create. They're used to hide creation logic and keep your code flexible when the exact type you need can change.

> Factories are polarizing. While very popular and idiomatic in languages like Java, some engineers see them as examples of overengineering. If you choose to implement one, take a look at your interviewer and check them for a grimace.

Factory shows up regularly in interviews, usually when requirements say "support different notification types" or "handle multiple payment methods." Instead of writing `new EmailNotification()` throughout your code, you call `notificationFactory.create(type)`. Now when you add SMS notifications, you update the factory. The rest of your code never changes.

factory_method.py — Python

```python
from abc import ABC, abstractmethod

class Notification(ABC):
    @abstractmethod
    def send(self, message: str) -> None:
        pass

class EmailNotification(Notification):
    def send(self, message: str) -> None:
        # Email sending logic
        pass

class SMSNotification(Notification):
    def send(self, message: str) -> None:
        # SMS sending logic
        pass

class NotificationFactory:
    @staticmethod
    def create(notification_type: str) -> Notification:
        if notification_type == "email":
            return EmailNotification()
        elif notification_type == "sms":
            return SMSNotification()
        raise ValueError("Unknown type")

# Usage
notif = NotificationFactory.create("email")
```

The factory centralizes creation logic. When you add push notifications, you modify one place. Factory controls which object gets instantiated. It makes the decision once and returns the right type.

This is technically called Simple Factory, not the Gang of Four Factory Method pattern. The GoF version uses abstract factory classes with subclasses that override a factory method. It's more complex and rarely shows up in real code or interviews. What we're showing here is what people actually build and what interviewers expect when they say "use a factory."

Builder

A builder is a helper that lets you create a complex object step by step without worrying about the order or messy construction details. It's used when an object has many optional parts or configuration choices.

This shows up when designing things like HTTP requests, database queries, or configuration objects. Instead of a constructor with ten parameters where half are null, you build the object incrementally.

builder.py — Python

```python
from typing import Optional

# NOTE: Builder is less common in Python. Python has better alternatives like
# dataclasses with default values, keyword arguments, or simple dictionaries.
# This pattern adds unnecessary complexity for most Python use cases.

class HttpRequest:
    def __init__(self):
        self.url: Optional[str] = None
        self.method: Optional[str] = None
        self.headers: dict[str, str] = {}
        self.body: Optional[str] = None

    class Builder:
        def __init__(self):
            self._request = HttpRequest()

        def url(self, url: str) -> 'HttpRequest.Builder':
            self._request.url = url
            return self

        def method(self, method: str) -> 'HttpRequest.Builder':
            self._request.method = method
            return self

        def header(self, key: str, value: str) -> 'HttpRequest.Builder':
            self._request.headers[key] = value
            return self
```

Builder makes construction readable and handles optional fields cleanly. It most commonly shows up in LLD interviews when you're designing API clients or complex configurations, but is very rarely used in other contexts.

> If the interviewer didn't describe a complex object with lots of optional details, Builder probably isn't needed. Most interview problems involve simple domain objects with 2-4 required fields where a normal constructor works fine.

Singleton

Singleton ensures only one instance of a class exists. Use it when you need exactly one shared resource like a configuration manager, connection pool, or logger.

Most of the time you don't actually need a Singleton. You can just pass shared objects through constructors instead — it's clearer and easier to test. Singletons hide dependencies and make testing harder.

singleton.py — Python

```python
# NOTE: Singletons are not idiomatic in Python. In Python, modules
# themselves are singletons - they're only imported once. For shared
# resources, create a module-level instance instead of this pattern.

class DatabaseConnection:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def query(self, sql: str) -> None:
        # Database operations
        pass

# Usage
db = DatabaseConnection()
db.query("SELECT * FROM users")
```

In Python, module-level variables are natural singletons since modules are only loaded once. The class-based approach shown here uses `get_instance` for consistency with other languages, but you could also just use a module-level object directly.

In interviews, know what Singleton is and when not to use it. If an interviewer asks "should this be a Singleton?", the answer is usually no unless they explicitly want a single shared instance across the entire system. There are thread-safe versions of Singleton, but interviewers don't expect you to implement them in LLD interviews.

Structural Patterns

Structural patterns deal with how objects connect to each other. They help you build flexible relationships between classes without creating tight coupling or messy dependencies.

Decorator

A decorator adds behavior to an object without changing its class. Use it when you need to layer on extra functionality at runtime.

Decorator is powerful but comes up less often than Strategy or Observer. You might need this when the requirements say things like "add logging to specific operations" or "encrypt certain messages." Instead of creating subclasses for every combination (`LoggedEmailNotification` , `EncryptedEmailNotification` , `LoggedEncryptedEmailNotification`), you wrap the base object with decorators. If you see words like "optional features," "stack behaviors," or "combine multiple enhancements," think Decorator.

decorator.py Python

```python
from abc import ABC, abstractmethod

# NOTE: This is the Decorator design pattern, which is different from Python's
# @decorator syntax. The naming can be confusing since Python decorators (with @)
# are a language feature for function/class modification, while this is an
# object composition pattern for adding behavior at runtime.

class DataSource(ABC):
    @abstractmethod
    def write_data(self, data: str) -> None:
        pass

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

class FileDataSource(DataSource):
    def __init__(self, filename: str):
        self.filename = filename

    def write_data(self, data: str) -> None:
        # Write to file
        pass

    def read_data(self) -> str:
        # Read from file
        return "data from file"

class EncryptionDecorator(DataSource):
    def __init__(self, source: DataSource):
        self._wrapped = source

    def write_data(self, data: str) -> None:
```

Use a Decorator when you need to add behavior at runtime based on conditions, like wrapping a service with logging only in debug mode or adding caching only for certain requests. It lets you layer optional, combinable features without modifying the underlying class. In most other cases, use normal subclasses, where the new behavior is fixed at design time and represents a stable variation of the original type. If the behavior depends on runtime conditions, choose Decorator; if it's a predefined type difference, choose Subclass.

Each decorator adds one piece of functionality. You can stack them in any order and add or remove them without touching the base class or other decorators, though in real systems order often affects behavior.

Facade

A facade is just a coordinator class that hides complexity. You're probably already building facades in every LLD interview without calling them that. Your `Game` class in Tic Tac Toe? That's a facade. Any orchestrator that coordinates multiple components behind a clean interface? Also a facade.

Almost nobody names this pattern when they're using it. The pattern name is more useful when you're wrapping existing messy code. Like, if you inherit a complex subsystem with awkward APIs, you write a facade to make it easier to use. But in LLD interviews, you're designing clean orchestrators from scratch, which happens to be the same structure. You're likely already doing the right thing instinctively, you just don't need to announce it.

facade.py — Python

```python
from enum import Enum

class GameState(Enum):
    IN_PROGRESS = 1
    WON = 2
    DRAW = 3

class Board:
    def place_mark(self, row: int, col: int, mark: str) -> bool:
        # Place mark logic
        return True

    def check_win(self, row: int, col: int) -> bool:
        # Check win logic
        return False

    def is_full(self) -> bool:
        # Check if board is full
        return False

class Player:
    def __init__(self, mark: str):
        self.mark = mark
```

```
    def get_mark(self) -> str:
        return self.mark

class Game:
    def __init__(self):
        self.board = Board()
        self.player_x = Player("X")
        self.player_o = Player("O")
```

The pattern name just describes what good orchestrator design looks like. Build it naturally, name it if it helps communicate, but don't worry if you never mention Facade by name in an interview.

Behavioral Patterns

Behavioral patterns control how objects interact and distribute responsibilities. They're about the flow of control and communication between objects.

Strategy

Strategy replaces conditional logic with polymorphism. Use it when you have different ways of doing the same thing and you want to swap them at runtime.

> When we say "runtime," we mean the moment the program is actually running. You can choose behaviors based on conditions, inputs, or configuration as the code executes. When we say "compile time," we mean decisions baked into the code itself. The behavior is fixed in the class definition and doesn't change while the program runs.

Interviewers love Strategy. It's the single most common pattern in LLD interviews because it directly tests whether you understand polymorphism and composition over inheritance. When you see a pile of `if/else` or `switch` statements based on type, that's a strategy pattern waiting to happen. If you learn one pattern from this page, make it this one. You've already seen this in the **OOP concepts** page with the parking lot vehicle example.

strategy.py — Python

```
from abc import ABC, abstractmethod

class PaymentStrategy(ABC):
    @abstractmethod
    def pay(self, amount: float) -> bool:
        pass

class CreditCardPayment(PaymentStrategy):
    def __init__(self, card_number: str):
        self.card_number = card_number
```

```
    def pay(self, amount: float) -> bool:
        # Credit card processing logic
        print(f"Paid {amount} with credit card")
        return True

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

    def pay(self, amount: float) -> bool:
        # PayPal processing logic
        print(f"Paid {amount} with PayPal")
        return True

class ShoppingCart:
    def __init__(self):
        self.payment_strategy = None

    def set_payment_strategy(self, strategy: PaymentStrategy) -> None:
```

Instead of having checkout logic full of `if (paymentType == "credit")` statements, each payment method handles itself. This is just polymorphism with a pattern name. Strategy swaps behavior at runtime through composition. The cart holds a reference to a strategy and delegates to it. Factory decides which type to instantiate. Strategy decides which behavior to use after the object already exists.

Observer

Observer lets objects subscribe to events and get notified when something happens. Use it when changes in one object need to trigger updates in other objects.

Observer is a top-tier interview pattern. It shows up when you're designing systems where multiple components care about state changes—a stock price changes and multiple displays need to update, or a user places an order and inventory, notifications, and analytics all need to know. If the problem involves the words "notify" or "update multiple components," you're probably looking at Observer.

observer.py — Python

```
from abc import ABC, abstractmethod

class Observer(ABC):
    @abstractmethod
    def update(self, symbol: str, price: float) -> None:
        pass

class Subject(ABC):
    @abstractmethod
    def attach(self, observer: Observer) -> None:
        pass
```

```
    @abstractmethod
    def detach(self, observer: Observer) -> None:
        pass

    @abstractmethod
    def notify_observers(self) -> None:
        pass

class Stock(Subject):
    def __init__(self, symbol: str):
        self._observers: list[Observer] = []
        self.symbol = symbol
        self.price = 0.0

    def attach(self, observer: Observer) -> None:
        self._observers.append(observer)

    def detach(self, observer: Observer) -> None:
        self._observers.remove(observer)
```

When the stock price changes, every attached observer gets updated automatically. No need for the stock to know what the observers do with the information.

State Machine

A state machine handles state transitions cleanly. Use it when an object's behavior changes based on its internal state and you have complex state transition rules. You'll also see this called the "State pattern" in some references, but state machine is the more common term.

State machines are less common than Strategy or Observer, but when you need one, it's usually the centerpiece of your entire design. If there's a state machine in your solution, the interview is probably organized around it—it's the most important thing to talk through. This shows up in LLD interviews when you're designing things like vending machines, document workflows, or game states. If the word "state" appears multiple times in the requirements, you're probably looking at a state machine. Instead of scattered conditionals checking current state everywhere, you encapsulate each state's behavior in its own class.

> Drawing a state diagram is one of the best ways to communicate a state machine design in an interview. Show the states as circles, transitions as arrows labeled with actions, and it becomes immediately clear how the system works. Interviewers appreciate the visual—it shows you're thinking clearly about the problem.

state_machine.py — Python

```
from abc import ABC, abstractmethod
from typing import TYPE_CHECKING
```

```
if TYPE_CHECKING:
    from __future__ import annotations

class VendingMachineState(ABC):
    @abstractmethod
    def insert_coin(self, machine: 'VendingMachine') -> None:
        pass

    @abstractmethod
    def select_product(self, machine: 'VendingMachine') -> None:
        pass

    @abstractmethod
    def dispense(self, machine: 'VendingMachine') -> None:
        pass

class NoCoinState(VendingMachineState):
    def insert_coin(self, machine: 'VendingMachine') -> None:
        print("Coin inserted")
        machine.set_state(HasCoinState())

    def select_product(self, machine: 'VendingMachine') -> None:
        print("Insert coin first")

    def dispense(self, machine: 'VendingMachine') -> None:
        print("Insert coin first")

class HasCoinState(VendingMachineState):
    def insert_coin(self, machine: 'VendingMachine') -> None:
        print("Coin already inserted")
```

Each state knows which state comes next and what actions are valid. No giant switch statements checking current state in every method.

Wrapping Up

Patterns only help when they match the problem you're solving. Most interview-ready designs use no patterns, or at most one or two. If you're reaching for three or more, you're probably forcing it and over-engineering.

Here's the quick pattern cheat sheet, grouped by category:

Creational Patterns

- **Factory** → Use when callers shouldn't care which concrete class gets created.
- **Builder** → Use when an object has lots of optional fields or messy construction details.
- **Singleton** → Use when you truly need one global instance (rare in interviews).

Structural Patterns

- **Decorator** → Use when you need to layer optional behaviors at runtime without subclass explosion.
- **Facade** → Use when you want to hide internal complexity behind a simple entry point.

Behavioral Patterns

- **Strategy** → Use when you're replacing if/else logic with interchangeable behaviors.
- **Observer** → Use when multiple components need to react to a single event.
- **State Machine** → Use when an object's behavior depends on its current state and transitions get messy.

Focus on solving the problem cleanly and name the pattern afterward if it fits.

Test Your Knowledge

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

Start Quiz

Login to track your progress

Next: Introduction →

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 24 comments

Sort By
Popular

Phuoc Nguyen

Top 10% • 1 month ago

Could you add 'before' and 'after' to make them easier to compare?

23

M

MagicAquaRooster979

Premium • 1 month ago

Something to note is that a pattern (if implemented by the user) can be considered to be an anti-patten / bad practice for a given language.

For example the singleton is a necessity in Java, but a user implemented singleton pattern in Python would 99%

of the time be considered bad practice / a death sentence in a Python interview, there is perhaps a single exception (e.g. a singleton to spin an expensive resource) but even then this would be an extreme edge-case.

3

Stefan Mai

Admin • 1 month ago

Strongly agree. Looking for the right way to incorporate this into the content here, expect some adjustments in the short-term!

10

Umair Shabbir

Premium • 18 days ago

I have never implemented a singleton in python but did not know it was an anti-pattern.
MagicAquaRooster979 do you have any book/talk recommendation about python that you found helpful?

1

N

NaturalBrownSkink773

Premium • 1 month ago

Shall the notifyObservers() method in Subject (Observer pattern) be private instead of public? I would expect it to not be called directly but rather through "official" business logic which requires suscribers' notification?

2

S

Sarthak Gupta

Premium • 18 days ago

The quiz contains a question about composite pattern. But it is not covered here. Are you planning to add it?

1

D

DistinctiveMagentaPuppy279

Premium • 29 days ago

'Factory Method' and 'Strategy' pattern looks quite similar. It would be great if you could include a small paragraph under 'Strategy' pattern pointing out differences between those.

1

C

CoherentBrownCricket324

Premium • 21 days ago

Strategy (behavioral) : How do you do something ?
Factory (creational) : How do you create something ?

1

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