Software design principles fundamentals

Author: Goran Kukic

Table of contents

Hello fellow developers and learners, 👋

Welcome to **"Software Design Principles Fundamentals"**! This mini handbook will guide you through the core principles that make code **cleaner, more maintainable, and robust**. The examples are written in JavaScript for simplicity, but the concepts are applicable to **any programming language** and paradigm, giving you a broad perspective.

Access my other tutorials on GitHub for more learning resources:

- **JavaScript Fundamentals**: https://github.com/GoranKukic/javascript-fundamentals
- **TypeScript Fundamentals**: https://github.com/GoranKukic/typescript-fundamentals

What You'll Learn Here 📘

In this handbook, you'll explore the principles that guide professional software design, including:

- **Fundamental Programming Principles** (DRY, KISS, YAGNI, SOC)
- **SOLID Principles** (SRP, OCP, LSP, ISP, DIP)
- **High-Level Architectural Principles**
- **Optimization** and **Maintainability Principles**
- **Modules**: Organizing and exporting code for better project structure
- **Behavioral** and **Team-Oriented Principles**

By the end of this guide, you'll have a solid understanding of how to structure your code in a way that's easy to maintain, extend, and collaborate on, whether you're working alone or in a team.

Who Is This For? 🎯

- **Beginners**: Who want to understand the fundamentals of professional software design
- **Team members**: Aiming to improve collaboration and code quality in shared projects
- **Interview candidates**: Preparing for software engineering interviews
- **Any developer**: Seeking to write cleaner, more maintainable, and scalable code

Let's Get Started 🚀

By the end of this handbook, I hope you'll have a solid understanding of the principles that make software **robust, readable, and maintainable**, ready to apply them in any project or language.

Write clean, maintainable code! 💡💻

Introduction

Software design principles are the foundational guidelines that help developers create maintainable, efficient, and scalable software systems. These principles serve as best practices for making informed decisions during the design and development stages, ensuring that the final product is reliable, easy to modify, and aligned with user needs. Whether you're developing a small project or working on a large-scale system, following software design principles improves the quality of your code and enhances the development process.

What Are Software Design Principles?

Software design principles are a set of guidelines that help developers structure their code and software architecture in a way that is both functional and sustainable. They cover a wide range of aspects in software development, from coding practices to high-level architectural design decisions. These principles help guide the design of systems in a way that avoids common pitfalls, promotes good practices, and makes code more understandable, maintainable, and flexible.

Why Are Software Design Principles Important?

The importance of software design principles lies in their ability to:

1. **Improve Code Quality**: By following these principles, developers can write clean, efficient, and well-organized code that is easier to understand and maintain over time.
2. **Promote Consistency**: Software design principles provide a common language and structure for developers, ensuring consistency across the codebase and between team members.
3. **Facilitate Collaboration**: When everyone on a team follows the same principles, it becomes easier to collaborate and integrate different components of the system.
4. Increase Maintainability: Code that follows good design principles is easier to modify and extend. This is especially important as software evolves and requirements change over time.
5. **Enhance Flexibility and Scalability**: By adhering to design principles, developers can create software that is adaptable and scalable, ensuring the system can grow with changing demands.
6. **Boost Efficiency**: Well-designed systems often have fewer bugs and less technical debt, which means developers can spend more time delivering features and less time fixing issues.

A Structured Approach to Design

The principles covered in this handbook are categorized into several key areas, each addressing a specific aspect of software design. These categories include:

1. **Fundamental Programming Principles**: These are the core practices that guide everyday coding decisions, such as keeping code simple and avoiding redundancy.
2. **SOLID Principles**: These principles are focused on object-oriented design and ensure that your code is modular, flexible, and easy to maintain.
3. **High-Level Architectural Principles**: These principles concern the overall structure of the software and its components, promoting modularity, encapsulation, and decoupling.
4. **Optimization and Maintainability Principles**: These guidelines emphasize the importance of balancing optimization with maintainability, ensuring that software remains efficient and adaptable in the long term.
5. **Behavioral and Team-Oriented Principles**: These principles emphasize collaboration, testing, and ensuring that systems behave as expected, which improves team efficiency and software reliability.

How to Use This Handbook

This handbook provides a comprehensive overview of key software design principles, each accompanied by examples and explanations to help you understand and apply them in your projects. Whether you're just getting started with software development or you're an experienced developer looking to refine your design approach, this guide will help you build better software, reduce technical debt, and improve collaboration within your team.

By applying the principles outlined in this handbook, you can ensure that your software not only meets its functional requirements but also adheres to the best practices that make it robust, maintainable, and scalable.

Fundamental Programming Principles

Programming is not just about getting a feature to work. It's about writing software that remains easy to understand, easy to maintain, and adaptable to change. The following fundamental principles are the foundation of good software design.

By following these principles, you can write code that is cleaner, more efficient, and easier for both you and others to work with over time.

Don't Repeat Yourself (DRY)

The DRY principle was popularized in The Pragmatic Programmer (1999) by Andy Hunt and Dave Thomas. It originated from the need to avoid unnecessary repetition in large software projects, where repeating the same code in multiple places led to errors and high maintenance costs.

Definition
The DRY principle states that you should avoid duplicating code or logic. Every piece of knowledge or logic should exist in a single, unambiguous place within your system.

Why it matters

- Maintenance: Fix a bug once and it's fixed everywhere.
- Consistency: Reduces the risk of having outdated or mismatched code.
- Clarity: Improves readability by avoiding unnecessary repetition.

Bad Example (Breaking DRY)

```
// Updating user profile in two different places
function updateUserProfilePage(user) {
  document.getElementById('name').innerText = user.name;
  document.getElementById('email').innerText = user.email;
}

function updateAdminProfilePage(user) {
  document.getElementById('name').innerText = user.name;
  document.getElementById('email').innerText = user.email;
}
```

Problem: The same logic for updating profile data is duplicated in multiple functions.

Good Example (Applying DRY)

```
function updateProfileUI(user) {
  document.getElementById('name').innerText = user.name;
  document.getElementById('email').innerText = user.email;
}

// Use it everywhere
updateProfileUI(currentUser);
```

Now: The update logic is extracted into a single reusable function. This avoids duplication, makes the code easier to maintain, and ensures consistency across all profile updates.

Illustration

```
Before DRY:
Code -> Code -> Code (3x same thing)

After DRY:
  +----------------+
  | Single Function|
  +----------------+
      /       \
   reuse     reuse
```

Keep It Simple, Stupid (KISS)

The KISS principle comes from the U.S. Navy in the 1960s. It emphasized that systems work best if they are kept simple, and that unnecessary complexity should be avoided.

Definition

The KISS principle encourages keeping solutions as simple as possible. Avoid unnecessary complexity or over-engineering.

Why it matters

- Simple code is easier to understand and debug.
- Reduces the chance of introducing bugs.
- Helps new developers quickly understand the project.

Bad Example (Too complex)

```
function getMax(a, b) {
  if (a > b) {
```

```
    return a;
  } else if (b > a) {
    return b;
  } else {
    return a;
  }
}
```

Problem:

- Too many conditions for something simple.
- Adds unnecessary branching.

Good Example (Simple and clear)

```
function getMax(a, b) {
  return Math.max(a, b);
}
```

Now:

- Much simpler, one line solution.
- Easier to read and maintain.

Illustration

```
Complex path: Start -> Maze -> Confusion -> Bug

Simple path: Start -> Done
```

You Aren't Gonna Need It (YAGNI)

YAGNI became popular through Extreme Programming (XP) in the late 1990s. It's a warning against building features based on speculation instead of actual needs.

Definition

Don't add features or logic until you actually need them. Avoid building things "just in case."

Why it matters

- Saves time and development effort.
- Keeps codebase lean and relevant.
- Reduces maintenance burden for unused features.

Bad Example (Adding unused properties)

```
class User {
  constructor(name) {
    this.name = name;
    this.favoriteColor = null; // Not needed now
  }
}
```

Problem:

- `favoriteColor` is added even though it's not used anywhere.
- This makes the class more complex than necessary and may mislead other developers into thinking it has a purpose.

Good Example (Build for current needs only)

```
class User {
  constructor(name) {
    this.name = name;
  }
}
```

Now:

- Class contains only what is currently required.
- Clearer, easier to maintain, and avoids unnecessary assumptions

Illustration

```
Extra luggage now = heavy trip later
[You] -> [Unnecessary Code] -> Slow progress
```

Separation of Concerns (SoC)

The term Separation of Concerns was introduced by Edsger W. Dijkstra in the 1970s as a way to reduce complexity by splitting software into distinct parts with minimal overlap in functionality.

Definition

Separate a program into distinct sections, each responsible for a specific task or concern. For example, data fetching, UI rendering, and business logic should be handled in different modules.

Why it matters

- Makes code easier to maintain and extend.

- Reduces the risk of breaking unrelated features when making changes.
- Improves team collaboration by allowing parallel work on different concerns.

Bad Example (All logic in one function)

```
function renderUserProfile() {
   fetch('/api/user')
    .then(res => res.json())
    .then(user => {
      document.body.innerHTML = `<h1>${user.name}</h1>`;
    });
}
```

Problem:

- Data fetching and UI rendering are mixed together.
- Any change in API or UI requires editing the same function.

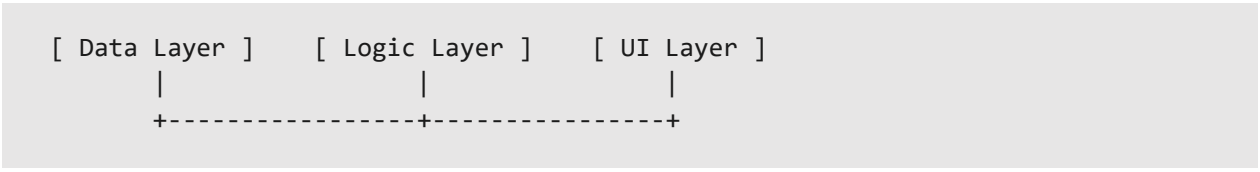
Good Example (Separate concerns)

```
// data.js
export async function getUser() {
  const res = await fetch('/api/user');
  return await res.json();
}

// ui.js
export function renderUser(user) {
  document.body.innerHTML = `<h1>${user.name}</h1>`;
}

// main.js
import { getUser } from './data.js';
import { renderUser } from './ui.js';

getUser().then(renderUser);
```

Now:

- Data fetching and UI rendering are clearly separated.
- Each module is easier to test, reuse, and maintain.

Illustration



SOLID Principles

The **SOLID** principles are a set of five guidelines that help developers design software that is easier to understand, maintain, and extend. The term was introduced by Robert C. Martin (Uncle Bob) in the early 2000s and became a **foundation for object-oriented design**.

They are not strict rules, but rather guidelines that encourage good software architecture.

Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) is the first of the SOLID principles and also the simplest to understand — yet it's one of the most commonly violated. The idea is that every class, function, or module should focus on **only one responsibility**.

Definition:
A class should have only one reason to change.
In other words, a class should do only one thing, and do it well.

Why it matters:

- **Simpler testing**: smaller units are easier to test.
- **Higher readability**: you instantly understand what the class does.
- **Safer changes**: modifying one responsibility won't break unrelated functionality.
- **Better teamwork**: multiple developers can work on different parts without conflicts.

Bad Example (violating SRP):
This class is doing too many things at once: saving to the database and sending an email.

```
class UserManager {
  saveToDB(user) {
    console.log("Saving user to database...");
  }

  sendEmail(user) {
    console.log("Sending email to user...");
  }
}
```

Problem: If tomorrow the email system changes (e.g., from SMTP to SendGrid), you'd have to modify this same class, even though database logic has nothing to do with sending emails.

Good Example (following SRP):
Separate responsibilities into two classes:

```
class UserRepository {
  saveToDB(user) {
    console.log("Saving user to database...");
  }
}

class UserNotifier {
  sendEmail(user) {
    console.log("Sending email to user...");
  }
}
```

Now each class has only one responsibility and one reason to change.

Illustration:

```
✕ One Person Does Everything
   +-----------------------+
   |   Employee            |
   |  - cookFood()         |
   |  - serveTable()       |
   |  - processPayment()   |
   +-----------------------+

✔ Clear Roles
   +---------+   +--------+   +------------+
   |  Chef   |   | Waiter |   |  Cashier   |
   | cook()  |   | serve()|   | pay()      |
   +---------+   +--------+   +------------+
```

Open/Closed Principle (OCP)

The Open/Closed Principle (OCP) is the second principle in SOLID. It encourages us to design software components that are open for extension but closed for modification.

Definition:
Software entities (classes, modules, functions) should be open for extension, but closed for modification.

This means we should be able to add new features without changing the existing, working code, but easy to extend (by adding new classes or methods). This is usually achieved through abstraction and polymorphism.

Why it matters:

- Prevents breaking existing features when adding new ones.
- Makes your code more stable and predictable.
- Encourages the use of abstraction (interfaces, base classes, strategy patterns).
- Critical in growing projects where features are constantly added

Bad Example (violating OCP):

Here, we must modify the same class whenever a new shape is added:

```
class AreaCalculator {
  calculate(shape) {
    if (shape.type === "circle") {
      return Math.PI * shape.radius * shape.radius;
    } else if (shape.type === "square") {
      return shape.side * shape.side;
    }
    // Adding a new shape? We must edit this class again!
  }
}
```

Problem: Every time a new shape (`triangle`, `rectangle`, etc.) is introduced, we have to go back and modify this class — risking breaking existing logic.

Good Example (following OCP):

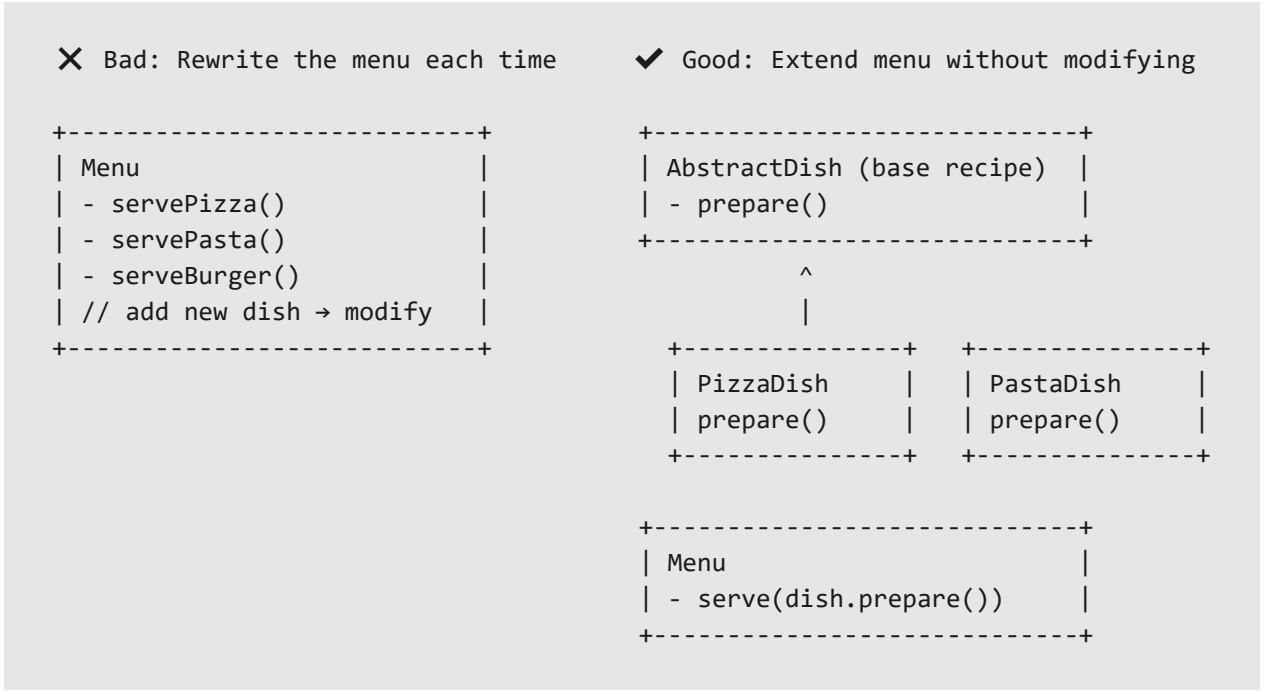
We use polymorphism: each shape knows how to calculate its own area. The calculator just calls the method.

```
class Circle {
  constructor(radius) {
    this.radius = radius;
  }
  area() {
    return Math.PI * this.radius * this.radius;
  }
}

class Square {
  constructor(side) {
    this.side = side;
```

```
  }
  area() {
    return this.side * this.side;
  }
}

class AreaCalculator {
  calculate(shape) {
    return shape.area();
  }
}

// Usage
const calculator = new AreaCalculator();
console.log(calculator.calculate(new Circle(5)));
console.log(calculator.calculate(new Square(4)));
```

Now, if we add a `Triangle class`, we don't touch `AreaCalculator` — we just extend the system with a new class.

Illustration:



Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP), formulated by Barbara Liskov, is the third SOLID principle. It states that subtypes must be substitutable for their base types without altering the correctness of the program.

Definition:
Objects of a superclass should be replaceable with objects of a subclass without affecting the program.

In other words, you should be able to replace a parent class with a child class without breaking the program.

Why it matters:

- Ensures **reliable polymorphism**.
- Makes **inheritance safer** and predictable.
- Prevents unexpected behavior when extending classes.
- Helps create more **robust, reusable code**.

Bad Example (violating LSP):

```
class Bird {
  fly() {
    console.log("Flying high!");
  }
}

class Penguin extends Bird {
  fly() {
    throw new Error("I can't fly!");
  }
}

function makeBirdFly(bird) {
  bird.fly(); // expects any bird to fly
}

makeBirdFly(new Bird());     // ✔ Flying high!
makeBirdFly(new Penguin());  // ✕ Error: breaks expectations
```

Problem: The `Penguin class` **breaks the contract** of the `Bird class`. Code expecting a "Bird" that flies now crashes.

Good Example (following LSP):

We separate birds into `FlyingBird` and `NonFlyingBird`, or use a `canFly` method to respect behavior:

```
class Bird {
  move() {
    console.log("Moving...");
  }
}

class FlyingBird extends Bird {
  fly() {
    console.log("Flying high!");
  }
}

class Penguin extends Bird {
  swim() {
    console.log("Swimming in water!");
  }
}

// Usage
const birds = [new FlyingBird(), new Penguin()];
birds.forEach(bird => bird.move()); // works for all birds
```

Now: Classes model natural behavior without violating expectations.

Illustration:

```
✕ Bad Example: Violates LSP
+------------------+
|      Bird        |
|------------------|
| fly()            |
+------------------+
         ^
         |
+------------------+
|    Penguin       |
| fly() throws err |
+------------------+

makeBirdFly(new Penguin());  // ✕ breaks code
```

```
✔ Good Example: Follows LSP
  +------------------+
  |      Bird        |
  |------------------|
  | move()           |
  +------------------+
           ^
           |
  +------------------+     +--------------------+
  | FlyingBird       |     | Penguin            |
  | move() = fly()   |     | move() = swim()    |
  +------------------+     +--------------------+

birds.forEach(bird => bird.move()); // ✅ works for all birds
```

Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) is the fourth SOLID principle. It emphasizes **that no client should be forced to depend on methods it does not use**.

Definition:
Many specific interfaces are better than one general-purpose interface.

In other words, instead of creating one huge interface with many methods, break it into smaller, more focused interfaces so classes only implement what they actually need.

Why it matters:

- Avoids forcing classes to implement **unused methods**.
- Reduces **unnecessary dependencies**.
- Makes code **cleaner, easier to maintain, and more flexible**.
- Improves **readability** — each class only sees what it actually uses.

Bad Example (violating ISP):
A general-purpose `Worker` interface forces all workers to implement methods they may not need.

```
  class Worker {
    work() {}
    eat() {}
  }

  class Robot extends Worker {
```

```
  work() {
    console.log("Robot is working...");
  }

  eat() {
    throw new Error("Robots do not eat!");
  }
}
```

Problem: `Robot` doesn't need `eat()`, but is forced to implement it, violating ISP.

Good Example (following ISP):

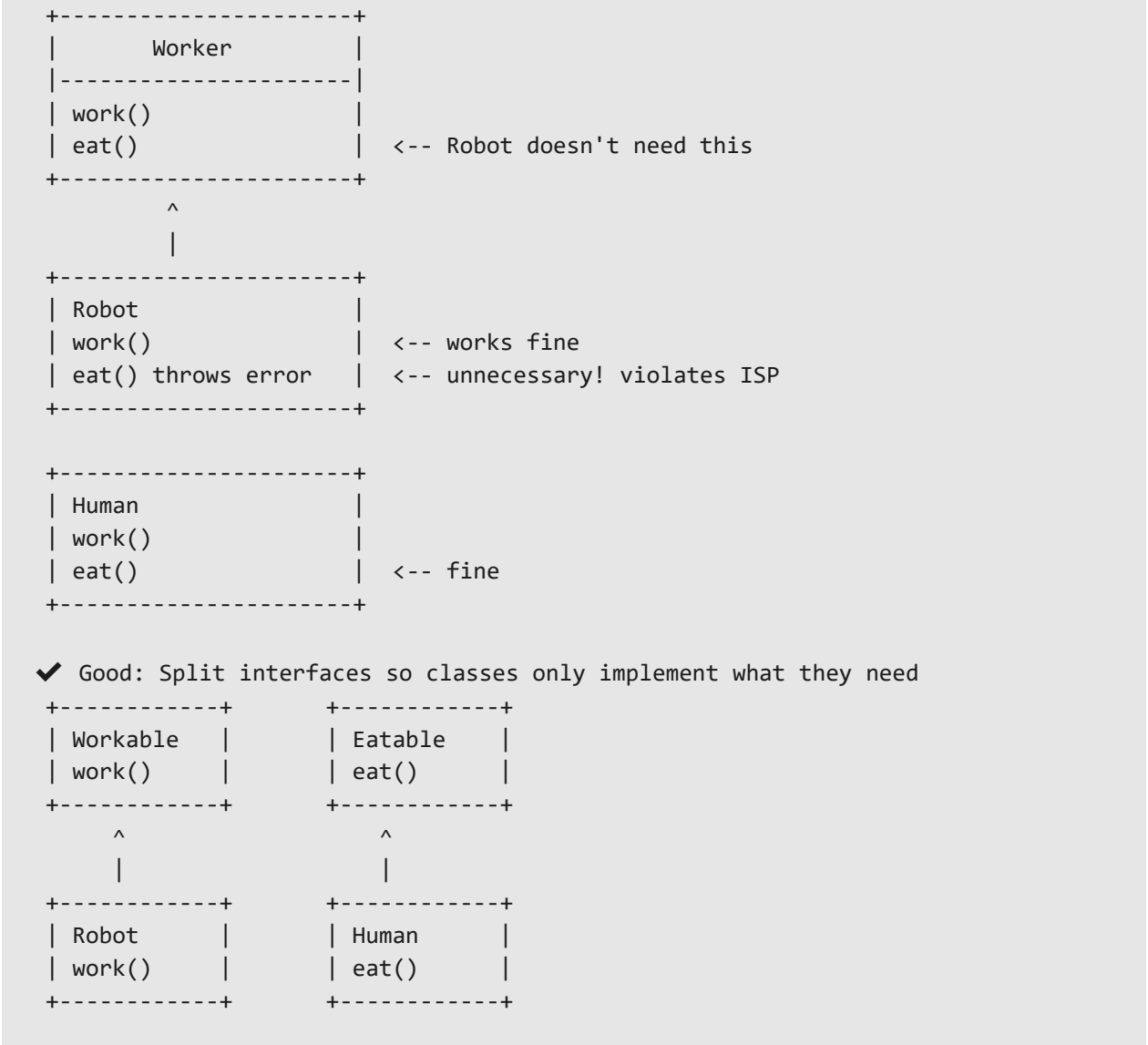
Split the interface into smaller, more focused interfaces.

```
class Workable {
  work() {}
}

class Eatable {
  eat() {}
}

class HumanWorker extends Workable {
  work() {
    console.log("Human is working...");
  }
  eat() {
    console.log("Human is eating...");
  }
}

class RobotWorker extends Workable {
  work() {
    console.log("Robot is working...");
  }
}

// Usage
const workers = [new HumanWorker(), new RobotWorker()];
workers.forEach(worker => worker.work());
```

Now: Each class only implements the methods it needs, keeping the design clean.

Illustration:

```
❌ Bad: One general interface forces unrelated responsibilities
```



Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) is the fifth SOLID principle. It emphasizes that **high-level modules should not depend on low-level modules directly; both should depend on abstractions.**

Definition:
Depend on abstractions, not on concrete implementations..

This means your code should depend on interfaces or abstract classes, not specific classes, making it easier to change or extend functionality without modifying high-level code.

Why it matters:

- Reduces **tight coupling** between modules.
- Makes code **flexible, maintainable, and testable**.
- Allows swapping implementations **without changing high-level logic**.

Bad Example (violating DIP):

A Car class directly depends on GasEngine, so changing to ElectricEngine requires modifying Car.

```
class GasEngine {
  start() {
    console.log("Gas engine started");
  }
}

class Car {
  constructor() {
     this.engine = new GasEngine(); // directly depends on concrete class
  }
  start() {
    this.engine.start();
  }
}

// Usage
const myCar = new Car();
myCar.start();
```

Problem: high-level module `Car` depends on a low-level module `GasEngine`, switching to `ElectricEngine` means editing `Car`, violating DIP.

Good Example (following DIP):

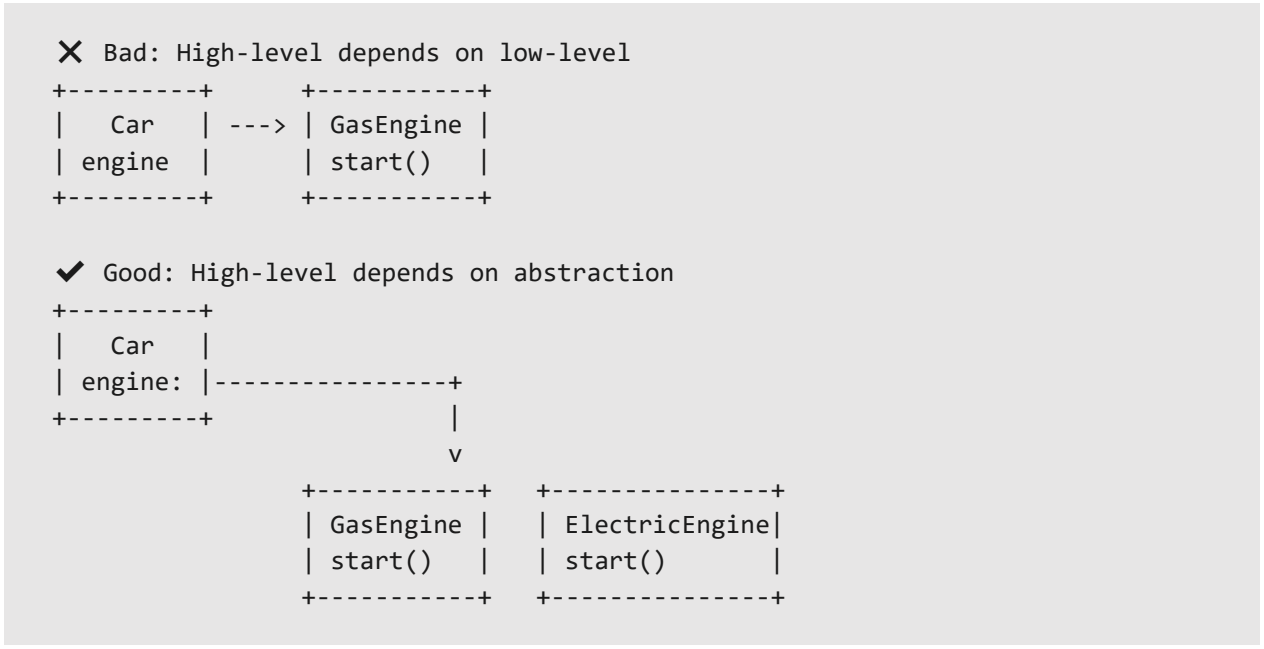
Introduce an Engine interface that Car depends on instead of a concrete engine:

```
class Engine {
  start() {} // abstraction
}

class GasEngine extends Engine {
  start() {
    console.log("Gas engine started");
  }
}

class ElectricEngine extends Engine {
```

```
  start() {
    console.log("Electric engine started");
  }
}

class Car {
  constructor(engine) {
    this.engine = engine; // depends on abstraction
  }
  start() {
    this.engine.start();
  }
}

// Usage
const gasCar = new Car(new GasEngine());
const electricCar = new Car(new ElectricEngine());

gasCar.start();        // Gas engine started
electricCar.start();   // Electric engine started
```

Now: Car depends on an abstraction, not a concrete engine, so it works with any engine type without modification. This makes the code flexible and easier to maintain.

Illustration:



High-level Architectural Principles

High-level architectural principles provide the foundation for designing robust, scalable, and maintainable systems. They serve as guiding rules that shape decision-making, ensuring consistency, performance, and long-term adaptability of the architecture..

Modularity

Modularity is a cornerstone of high-level software architecture. It's the idea of breaking down a system into smaller, independent, and interchangeable parts (modules). Each module focuses on one well-defined functionality, making the system easier to understand, maintain, and evolve..

Definition:
A modular system is composed of self-contained units (modules), each responsible for a distinct feature or concern, which can be developed, tested, and deployed independently.

Why it matters:

- **Scalability:** You can add or replace modules without touching the whole system.
- **Maintainability:** Bugs are easier to locate and fix when logic is separated.
- **Reusability:** Modules can often be reused across different projects.
- **Team efficiency:** Different teams can work on different modules in parallel.

Bad Example (violating modularity):
Here, everything — authentication, payments, and notifications — is crammed into one giant module:

```
const App = {
  login(user, password) {
    console.log("Authenticating user...");
  },
  processPayment(amount) {
    console.log("Processing payment...");
  },
  sendNotification(message) {
    console.log("Sending notification...");
  }
};
```

Problem: If you need to change the payment logic, you risk breaking authentication or notification logic because everything is tangled together.

Good Example (following modularity):
We separate responsibilities into distinct modules:

```
// Auth module
function login(user, password) {
  console.log("Authenticating user...");
}
// Payment module
function processPayment(amount) {
  console.log("Processing payment...");
}
// Notification module
function sendNotification(message) {
  console.log("Sending notification...");
}
// Composing them together
const App = {
  login,
  processPayment,
  sendNotification
};
```

Now: Each function/module has its own clear responsibility. Changing `processPayment` won't affect `login` or `sendNotification`, and new modules can be added easily.

Illustration:

```
❌ One Big Module
+-------------------------------+
|             App               |
| login() processPayment() send()|
+-------------------------------+

✔ Independent Functions
+----------+   +--------------+   +-----------------+
|  login   |   | processPayment|  | sendNotification|
+----------+   +--------------+   +-----------------+
        \             |             /
         \            |            /
              +-----------------+
              |       App       |
              +-----------------+
```

Encapsulation

Encapsulation is the practice of **hiding internal details** of a module or function, exposing only what is necessary. It protects the system from unintended interference and simplifies usage.

Definition:
A module should **control access to its internal state and logic**, exposing only public interfaces for other parts of the system to interact with.

Why it matters:

- **Prevents unintended modifications:** Internal data can't be changed accidentally.
- **Improves maintainability:** Changes inside the module don't affect external code.
- **Simplifies usage:** Users only interact with a clear, limited interface.
- **Enhances testability:** Easier to mock or replace modules.

Bad Example (violating encapsulation):
Everything is exposed globally:

```
const AuthModule = {
  loggedIn: false, // direct access from outside

  login(user, password) {
    console.log("Authenticating user...");
    this.loggedIn = true;
  },

  logout() {
    this.loggedIn = false;
  }
};

// Somewhere else in the code
AuthModule.loggedIn = true; // breaks logic
```

Problem: External code can directly modify `loggedIn`, bypassing login logic and breaking the system.

Good Example (following encapsulation):
We hide the internal state using closures:

```
function createAuthModule() {
  let loggedIn = false; // private variable
```

```
  function login(user, password) {
    console.log("Authenticating user...");
    loggedIn = true;
  }

  function logout() {
    loggedIn = false;
  }

  function isLoggedIn() {
    return loggedIn;
  }

  return { login, logout, isLoggedIn };
}

// Usage
const Auth = createAuthModule();
Auth.login("user1", "pass123");
console.log(Auth.isLoggedIn()); // true
// Auth.loggedIn = true; // ✕ Not allowed
```

Now: The internal state `loggedIn` is private. External code can't change it directly.

Illustration:

```
✕ Everything Exposed
 +------------------+
 |  AuthModule      |
 | loggedIn         |
 | login()          |
 | logout()         |
 +------------------+
 External code can modify loggedIn directly
 +------------------+
 | loggedIn = true  |
 +------------------+

✔ Encapsulated Module
 +------------------+
 |  AuthModule      |
 | [loggedIn]       | <- hidden
 | login()          |
 | logout()         |
 | isLoggedIn()     |
 +------------------+
 External code can only interact via functions
```

Coupling and Cohesion

Coupling and cohesion are complementary principles that describe **how modules relate to each other** and **how well their internal parts work together**. High cohesion and low coupling lead to maintainable and flexible systems.

Definition

- **Cohesion:** Measures how closely related the responsibilities of a single module are. High cohesion means a module has a **clear, focused purpose**.
- **Coupling:** Measures how dependent a module is on other modules. Low coupling means a module can **function independently**.

Why it matters

- **Easier maintenance:** Changes in one module are less likely to break others.
- **Improved readability:** Modules have a single, clear responsibility.
- **Better reusability:** Independent, focused modules can be reused in other projects.
- **Team efficiency:** Teams can work on separate modules with minimal conflicts.

Bad Example (tight coupling, low cohesion)

```
const App = {
  login(user, password) {
    console.log("Authenticating user...");
    this.processPayment(100); // directly calls unrelated module
  },

  processPayment(amount) {
    console.log("Processing payment...");
    this.sendNotification("Payment done"); // also calls notification
  },

  sendNotification(message) {
    console.log("Sending notification:", message);
  }
};
```

Problem:

- `login` is indirectly calling payment and notification.
- Modules are **tightly coupled** and responsibilities are **scattered**, reducing cohesion.

Good Example (low coupling, high cohesion)

```
function createAuthModule() {
  function login(user, password) {
    console.log("Authenticating user...");
    return true;
  }
  return { login };
}

function createPaymentModule() {
  function processPayment(amount) {
    console.log("Processing payment...");
    return true;
  }
  return { processPayment };
}

function createNotificationModule() {
  function sendNotification(message) {
    console.log("Sending notification:", message);
  }
  return { sendNotification };
}

// Composing App
const Auth = createAuthModule();
const Payment = createPaymentModule();
const Notify = createNotificationModule();

// Usage
if (Auth.login("user1", "pass123")) {
  Payment.processPayment(100);
  Notify.sendNotification("Payment done");
}
```

Now:

- Each module has **one focused responsibility** (high cohesion).
- Modules communicate **via public interfaces only** (low coupling).
- Easier to replace or extend any module independently.

Illustration

```
✕ Tightly Coupled & Low Cohesion
 +------------------------------+
 |           App                |
 | login() -> processPayment() ->|
 | sendNotification()           |
 +------------------------------+
```

```
✔ Loosely Coupled & High Cohesion
 +---------+   +------------+   +-----------------+
 |  Auth   |   |  Payment   |   |  Notification   |
 +---------+   +------------+   +-----------------+
        \          |              /
         \         |             /
          +----------------------+
                  App
```

Design by Contract (DbC)

Design by Contract is a principle where **modules communicate through well-defined contracts**, specifying **preconditions, postconditions, and invariants**. It ensures that each module behaves predictably and errors are detected early.

Definition

A contract defines the rules a module **must satisfy**:

- **Preconditions:** What must be true before calling a function.
- **Postconditions:** What will be true after the function executes.
- **Invariants:** Conditions that remain true during the module's lifetime.

Why it matters

- **Early bug detection:** Violations of the contract reveal errors immediately.
- **Improved reliability:** Each module clearly defines expectations.
- **Clear documentation:** Contracts serve as formal, enforceable specifications.
- **Safer integration:** Modules can interact with confidence, knowing contracts are respected.

Bad Example (no contract)

```
function divide(a, b) {
  return a / b;
}

console.log(divide(10, 0)); // ✕ returns Infinity, unexpected
```

Problem:

- No contract: function assumes b is non-zero, but nothing enforces it.
- Leads to unexpected behavior and potential bugs.

Good Example (with contract checks)

```
function divide(a, b) {
  if (b === 0) throw new Error("Precondition failed: divisor must not be zero");
  const result = a / b;
  if (!isFinite(result)) throw new Error("Postcondition failed: result must
     be finite");
  return result;
}

console.log(divide(10, 2)); // ✔ works correctly
// divide(10, 0); // ✕ throws error
```

Now:

- Preconditions prevent invalid inputs.
- Postconditions ensure expected outputs.
- The contract is **explicit**, making the function predictable and safe.

Illustration

```
✕ No Contract
+----------------+
| divide(a, b)   |
|                |
+----------------+
b = 0 -> unexpected Infinity

✔ With Contract
+----------------+
| divide(a, b)   |
| Preconditions: |
|  b != 0        |
| Postconditions:|
|  result finite |
+----------------+
 Violations throw errors immediately.
```

Event-Driven Design

Event-Driven Design organizes a system around **events** and **listeners**. Modules emit events when something happens, and other modules respond without tight coupling. This creates a **flexible, reactive architecture**.

Definition

A module (or component) should **emit events** for significant changes and allow other modules to **subscribe** to these events. This decouples modules and promotes responsiveness.

Why it matters

- **Low coupling:** Modules don't need direct references to each other.
- **High flexibility:** New event handlers can be added without changing existing code.
- **Reactive behavior:** System responds automatically to changes or user actions.
- **Better scalability:** Easy to extend features by adding new event listeners.

Bad Example (tight coupling)

```
const App = {
  login(user) {
    console.log("Authenticating user...");
    // Directly calls other modules
    Payment.processPayment(100);
    Notify.sendNotification("Payment done");
  }
};

const Payment = {
  processPayment(amount) {
    console.log("Processing payment...");
  }
};

const Notify = {
  sendNotification(msg) {
    console.log("Sending notification:", msg);
  }
};
```

Problem:

- `login` directly calls `Payment` and `Notify`.
- Modules are tightly coupled and cannot operate independently.

Good Example (event-driven)

```
const EventBus = {
  events: {},

  subscribe(event, listener) {
    if (!this.events[event]) this.events[event] = [];
```

```
    this.events[event].push(listener);
  },

  emit(event, data) {
    if (this.events[event]) {
      this.events[event].forEach(listener => listener(data));
    }
  }
};

// Modules
const Auth = {
  login(user) {
    console.log("Authenticating user...");
    EventBus.emit("loginSuccess", { user });
  }
};

const Payment = {
  init() {
    EventBus.subscribe("loginSuccess", ({ user }) => {
      console.log(`Processing payment for ${user}`);
    });
  }
};

const Notify = {
  init() {
    EventBus.subscribe("loginSuccess", ({ user }) => {
      console.log(`Sending notification for ${user}`);
    });
  }
};

// Initialize event subscriptions
Payment.init();
Notify.init();

// Usage
Auth.login("user1");
```

Now:

- Auth emits a `loginSuccess` event.
- `Payment` and `Notify` respond independently.
- Modules remain **decoupled** and easy to extend.

Illustration

Optimization and Maintainability Principles

Optimization and maintainability principles are essential for building reliable, efficient, and scalable software. While optimization ensures that applications perform smoothly and use resources effectively, maintainability focuses on creating clean, well-structured code that is easy to update, extend, and debug.

Avoid Premature Optimization

"Avoid premature optimization" is a timeless principle in software engineering, famously highlighted by Donald Knuth, who said: *"Premature optimization is the root of all evil."* The idea is simple: don't try to make your code faster or more efficient before you know where the actual bottlenecks are.

Definition

Avoiding premature optimization means focusing first on writing clear, correct, and maintainable code — and only optimizing when performance issues are proven and measurable.

Why it matters

- **Clarity first:** Readable code is easier to understand and extend.
- **Real bottlenecks:** Most of the time, slowdowns come from a small part of the code — not the whole system.
- **Maintainability:** Over-optimizing early often leads to complex code that's hard to debug.
- **Efficiency where it counts:** Measuring first ensures you spend time on what truly improves performance.

Bad Example (premature optimization)

Here, the developer writes a complicated loop to avoid using the simpler built-in method, "just to make it faster," without knowing if speed is even an issue.

```
// Trying too hard to optimize before knowing if needed
function sumArray(arr) {
  let sum = 0;
  for (let i = 0, len = arr.length; i < len; i++) {
    sum += arr[i];
  }
  return sum;
}
```

Problem: This code is harder to read and maintain, and in modern JavaScript engines, it's not actually faster than using built-ins.

Good Example (avoiding premature optimization)
Write clear code first, then optimize later if measurements show it's necessary.

```
// Clean, readable code first
function sumArray(arr) {
  return arr.reduce((sum, num) => sum + num, 0);
}
```

If performance becomes an issue with very large arrays, then you can benchmark and optimize intelligently.

Illustration

```
❌ Bad: Optimizing too early
+-----------------------------------+
| Builder spends weeks polishing    |
| door hinges before foundation is  |
| even finished.                    |
+-----------------------------------+
✔ Good: Focus on essentials first
+-----------------------------------+
| Build strong foundation and walls |
| THEN optimize details like hinges |
| or paint.                         |
+-----------------------------------+
```

Fail Fast

"Fail fast" is a principle in software design that encourages systems to detect and report errors as early as possible instead of continuing execution with invalid data or assumptions. The goal is to stop problems before they spread and become harder to debug.

Definition
Failing fast means immediately stopping execution and raising an error when something goes wrong, rather than silently ignoring it or letting the program proceed in an invalid state.

Why it matters

- **Early detection**: Problems are caught where they happen, making them easier to diagnose.

- **Prevents data corruption**: The system avoids operating on invalid or incomplete data.
- **Simplifies debugging**: Errors point directly to the root cause instead of appearing later.
- **Robust systems**: Encourages explicit handling of unexpected cases, improving reliability.

Bad Example (not failing fast)

Here, the function ignores invalid inputs and continues, which may lead to corrupted results later.

```
// Silently ignores invalid input
function divide(a, b) {
  if (b === 0) {
    return 0; // pretend it's fine
  }
  return a / b;
}

console.log(divide(10, 0)); // outputs 0 (misleading!)
```

Problem: Returning `0` hides the error. The program continues as if everything is fine, which can create subtle bugs down the line.

Good Example (fail fast)

```
Instead, we throw an error immediately when invalid input occurs.
// Fails fast with clear error
function divide(a, b) {
  if (b === 0) {
    throw new Error("Division by zero is not allowed!");
  }
  return a / b;
}

console.log(divide(10, 2)); // 5
console.log(divide(10, 0)); // Error: Division by zero is not allowed!
```

Now: The error is obvious and caught early. Debugging is straightforward.

Illustration

```
✕ Bad: Hiding the problem
+-----------------------------------+
| Car makes strange noise, driver   |
| turns up the radio and ignores it |
+-----------------------------------+
```

```
✔ Good: Fail fast
+-----------------------------------+
| Car makes strange noise, driver   |
| stops immediately to check engine |
+-----------------------------------+
```

Code Reusability

Code reusability is the practice of writing code that can be used multiple times across different parts of a program or even in different projects. This reduces duplication, improves maintainability, and speeds up development.

Definition

Code reusability means creating functions, modules, or classes that encapsulate a specific functionality so they can be reused wherever needed without rewriting the same logic.

Why it matters

- **Reduces duplication**: Same logic doesn't need to be written multiple times.
- **Easier maintenance**: Fixing a bug in one reusable module fixes it everywhere it's used.
- **Faster development**: Reusing tested code saves time.
- **Consistency**: Ensures the same behavior across different parts of the system.

Bad Example (no reusability)

Here, the same logic for calculating the total price is repeated in multiple places:

```
// Repeated code
function order1Total(items) {
  let total = 0;
  for (let i = 0; i < items.length; i++) {
    total += items[i].price;
  }
  return total;
}

function order2Total(items) {
  let total = 0;
  for (let i = 0; i < items.length; i++) {
    total += items[i].price;
  }
  return total;
}
```

Problem:

- The same code is duplicated.
- Any future change (like adding tax) requires modifying multiple places.
- Higher chance of bugs and inconsistency.

Good Example (reusable code)

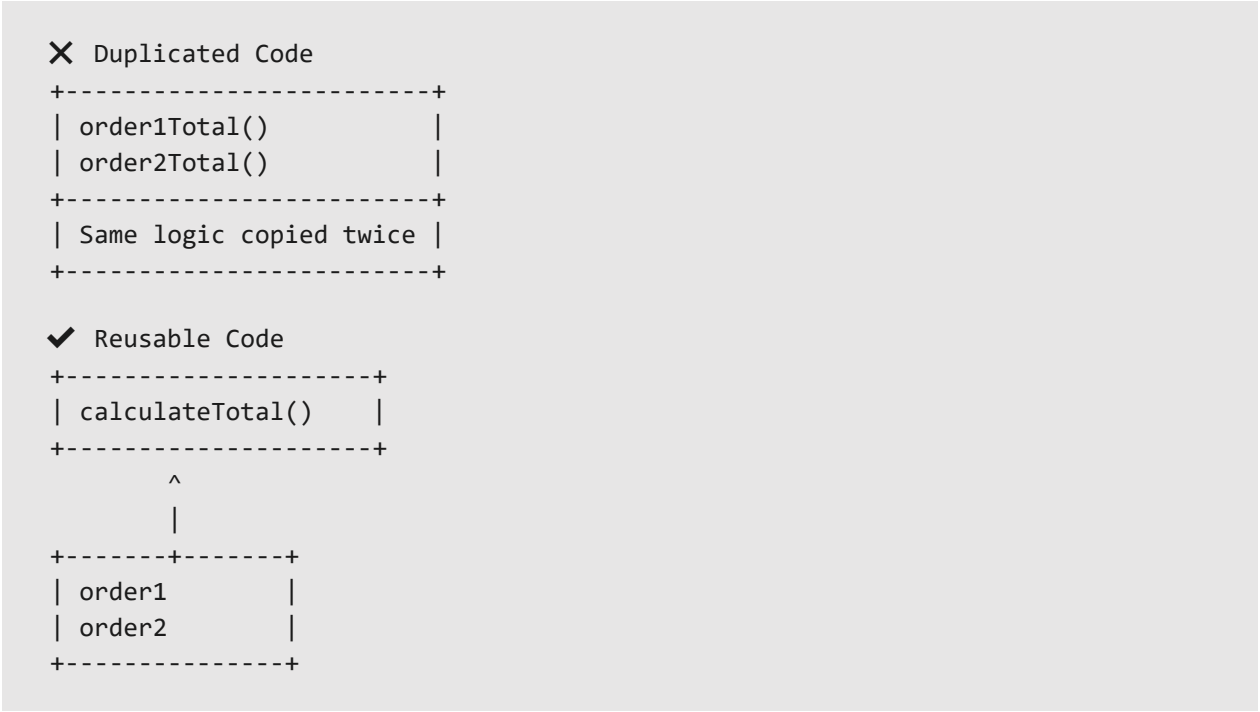
We create a single reusable function for calculating totals:

```
function calculateTotal(items) {
  return items.reduce((sum, item) => sum + item.price, 0);
}

// Reuse in multiple orders
let order1 = calculateTotal([{ price: 10 }, { price: 20 }]);
let order2 = calculateTotal([{ price: 5 }, { price: 15 }, { price: 10 }]);

console.log(order1); // 30
console.log(order2); // 30
```

Now:

- The logic is centralized in one function.
- Changes like adding tax can be done in one place.
- Less duplication, easier maintenance, and safer code.

Illustration



Minimize Dependencies

Minimizing dependencies means reducing the number of modules, classes, or functions that rely directly on each other. Less dependency makes code easier to maintain, test, and extend.

Definition

Minimize dependencies is the practice of designing modules so that they interact through **well-defined interfaces or abstractions**, rather than relying on internal details of other modules.

Why it matters

- **Easier maintenance**: Changes in one module are less likely to break others.
- **Better testability**: Modules with fewer dependencies are easier to test in isolation.
- **Flexibility**: Modules can be replaced or extended without affecting the entire system.
- **Reduced complexity**: Lower interconnection reduces risk of unexpected bugs.

Bad Example (high dependency)

Here, the `Car` module directly creates and relies on a specific `GasEngine` implementation:

```
const GasEngine = () => ({
  start: () => console.log("Gas engine starting...")
});

const Car = () => {
  const engine = GasEngine(); // tightly coupled
  return {
    drive: () => {
      engine.start();
      console.log("Car is driving");
    }
  };
};
```

Problem:

- `Car` is tightly coupled to `GasEngine`.
- Switching to `ElectricEngine` requires modifying `Car`.
- Harder to test `Car` independently.

Good Example (low dependency)

We introduce an abstraction by **passing the engine as a parameter**, letting `Car` depend only on something with a `start()` method:

```
const GasEngine = () => ({ start: () => console.log("Gas engine starting...") });
const ElectricEngine = () => ({ start: () => console.log("Electric engine
starting...") }));

const Car = (engine) => ({
  drive: () => {
    engine.start();
    console.log("Car is driving");
  }
});

// Usage
const gasCar = Car(GasEngine());
const electricCar = Car(ElectricEngine());

gasCar.drive();      // Gas engine starting... Car is driving
electricCar.drive(); // Electric engine starting... Car is driving
```

Now:

- `Car` can work with **any engine** implementing `start()`.
- Changing engine type does **not require modifying `Car`**.
- Code is more **flexible, testable, and maintainable**.

Illustration

```
✕ High Dependency
+---------+      +-----------+
|   Car   | ---> | GasEngine |
| drive() |      | start()   |
+---------+      +-----------+

✔ Low Dependency
      +---------+
      |   Car   |
      | drive() |-----------------+
      +---------+                 |
                                  v
                     +-----------+   +---------------+
                     | GasEngine |   | ElectricEngine|
                     | start()   |   | start()       |
                     +-----------+   +---------------+
```

Behavioral and Team-Oriented Principles

Behavioral and Team-Oriented Principles focus on how software is developed in practice, not just how it's structured. They emphasize collaboration, communication, testing practices, and conventions that help teams build reliable, understandable, and maintainable systems together.

Convention Over Configuration

"Convention over configuration" means preferring sensible defaults over requiring developers to write extensive configuration for common tasks. By following conventions, teams can focus on the unique aspects of their applications instead of boilerplate setup.

Definition

Convention over configuration is a design principle where a system makes assumptions about defaults, requiring configuration only when behavior deviates from those defaults.

Why it matters

- **Less boilerplate**: Reduces repetitive setup code.
- **Consistency**: Teams follow the same patterns without extra discussion.
- **Faster onboarding**: New developers can be productive quickly.
- **Focus on essentials**: Developers spend time on unique business logic.

Bad Example (too much configuration)

Here, every function requires explicit configuration, even when defaults would work:

```
function connectToDatabase(config) {
  const host = config.host || "localhost";
  const port = config.port || 5432;
  const user = config.user || "admin";
  const password = config.password || "password";

  console.log(`Connected to DB at ${host}:${port} as ${user}`);
}

// Usage (developer must configure everything, even defaults)
connectToDatabase({
  host: "localhost",
  port: 5432,
  user: "admin",
  password: "password"
});
```

Problem: Developer has to specify values that are already the defaults. More boilerplate, more room for mistakes.

Good Example (conventions first)

Here, defaults are applied automatically, and configuration is only needed if we want to override them:

```
function connectToDatabase(config = {}) {
  const defaults = {
    host: "localhost",
    port: 5432,
    user: "admin",
    password: "password"
  };

  const finalConfig = { ...defaults, ...config };

  console.log(
    `Connected to DB at ${finalConfig.host}:${finalConfig.port}
    as ${finalConfig.user}`
  );
}

// Usage (use defaults)
connectToDatabase();

// Usage (override only what's necessary)
connectToDatabase({ user: "alice", password: "securePass" });
```

Now:

- Defaults handle the common case.
- Developers configure only when they need non-standard behavior.
- Code is shorter, clearer, and more maintainable.

Illustration

```
✕ Configuration Overload
+------------------------------+
| Must define EVERY setting    |
| even if 90% are defaults     |
+------------------------------+

✔ Convention Over Configuration
+------------------------------+
```

```
| Sensible defaults in place    |
| Only override when needed     |
+-------------------------------+
```

You Can’t Fix What You Don’t Measure

In software engineering, assumptions about performance or reliability often turn out to be wrong. Without measuring, you don’t know where the real problems are — and trying to “fix” blindly can waste time or even make things worse.

Definition
“You can’t fix what you don’t measure” means you must collect data (metrics, logs, benchmarks) before attempting to improve performance, reliability, or user experience.

Why it matters

- **Evidence-based decisions**: Changes are guided by data, not guesswork.
- **Accurate bottleneck detection**: Focus effort where it matters most.
- **Prevents wasted effort**: Avoids optimizing parts of the system that don’t need it.
- **Builds accountability**: Teams can prove improvements through measurable results.

Bad Example (guessing without measurement)
A developer receives complaints that “the app loads the user list slowly” and immediately changes the filtering function in code, assuming it’s the problem::

```
function filterUsers(users) {
  // Developer assumes this loop is slow
  let result = [];
  for (let i = 0; i < users.length; i++) {
    if (users[i].active) result.push(users[i]);
  }
  return result;
}
```

Problem:

- The developer optimizes `filterUsers` without knowing if it’s the real bottleneck.
- Maybe the actual problem is the **database query** fetching users.
- Time and effort are wasted, and the code becomes unnecessarily complex.

.

Good Example (measure, then fix)

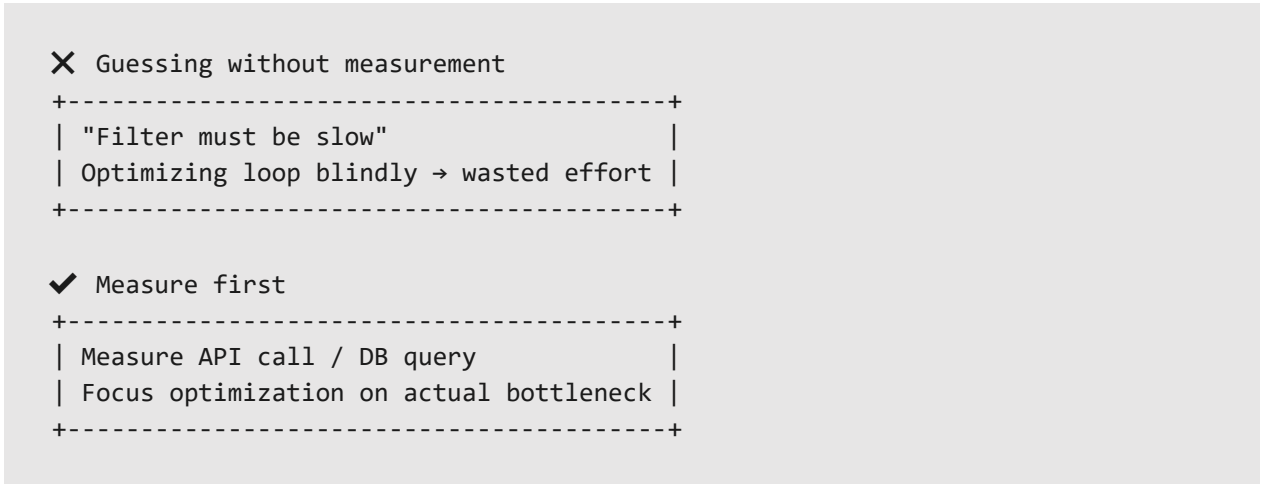
The developer first measures API response time before touching the filtering logic:

```
async function fetchUsers() {
  console.time("fetchUsers"); // Measure total API call
  const users = await getUsersFromDB(); // Could be the slow part
  console.timeEnd("fetchUsers");

  const activeUsers = users.filter(u => u.active); // Filter in memory
  only if needed
  return activeUsers;
}

// Usage
fetchUsers();
```

Now:

- The developer sees that fetching from the database is the slow part, not filtering.
- Optimization efforts focus on **DB query or indexing**, not the loop in memory.
- Time is saved, and changes are targeted and meaningful.

Illustration



Least Astonishment Principle (LAP)

The Least Astonishment Principle encourages designing software so that its behavior is intuitive and matches the expectations of its users or other developers. When code behaves in an unexpected way, it can lead to bugs, confusion, and wasted time.

Definition
Design software so that the behavior is obvious and predictable. If someone reads your code or uses your API, the results should align with common sense and expectations.

Why it matters

- **Reduces bugs**: Surprising behavior often leads to errors.
- **Improves maintainability**: Code is easier for other developers to understand.
- **Better user experience**: Users interact with predictable interfaces.
- **Encourages consistency**: Aligns with established conventions and patterns.

Bad Example (violating LAP)
A function for removing a user **silently deletes their account**, instead of returning a confirmation or throwing an error:

```
function removeUser(users, id) {
  // Deletes user silently
  const index = users.findIndex(u => u.id === id);
  users.splice(index, 1);
  // No feedback, no warning
  return users;
}

// Usage
const userList = [{id: 1, name: "Alice"}, {id: 2, name: "Bob"}];
removeUser(userList, 3); // Oops, user 3 doesn’t exist, but no warning is given
```

Problem:

- The developer or user is surprised when nothing happens or data disappears silently.
- Hard to debug issues caused by unexpected behavior.

Good Example (following LAP)
Provide clear feedback and predictable behavior:

```
function removeUser(users, id) {
  const index = users.findIndex(u => u.id === id);
  if (index === -1) {
    throw new Error(`User with id ${id} not found`);
  }
  users.splice(index, 1);
  return users;
}

// Usage
```

```
try {
  const userList = [{id: 1, name: "Alice"}, {id: 2, name: "Bob"}];
  removeUser(userList, 3);
} catch (err) {
  console.log(err.message); // User with id 3 not found
}
```

Now:

- Behavior matches expectations: you can't remove a non-existing user silently.
- Easier debugging and safer code.
- API is predictable and consistent for all developers.

Illustration

```
✕ Surprising / Astonishing behavior
+---------------------------+
| removeUser silently fails  |
| No error, no feedback      |
+---------------------------+

✔ Predictable / Intuitive behavior
+---------------------------+
| removeUser checks existence|
| Throws error if not found  |
| Feedback is clear          |
+---------------------------+
```

Test-Driven Development (TDD)

Test-Driven Development is a software development approach where tests are written **before** the actual code. The idea is to define the expected behavior first, then implement code to pass those tests. This ensures correctness and guides design.

Definition

Write automated tests **before** writing the production code. Development proceeds in short cycles:

1. Write a failing test.
2. Write code to make the test pass.
3. Refactor if needed.

Why it matters

- **Ensures correctness**: Code is validated as it's written.

- **Guides design**: Writing tests first helps you think about interfaces and responsibilities.
- **Reduces bugs**: Early detection of problems.
- **Facilitates refactoring**: Tests act as safety nets.

Bad Example (no TDD)

Code is written first without tests; issues may appear later:

```
// Production code
function addUser(users, user) {
  users.push(user);
  // Forgot to check for duplicate IDs
  return users;
}

const userList = [{id: 1, name: "Alice"}];
addUser(userList, {id: 1, name: "Bob"});
console.log(userList);
// [{id: 1, name: "Alice"}, {id: 1, name: "Bob"}] -> duplicate ID inserted
```

Problem:

- Bugs like duplicate entries go unnoticed until later.
- Harder to track down problems in larger systems.

Good Example (following TDD)

Write tests first, then implement the functionality:

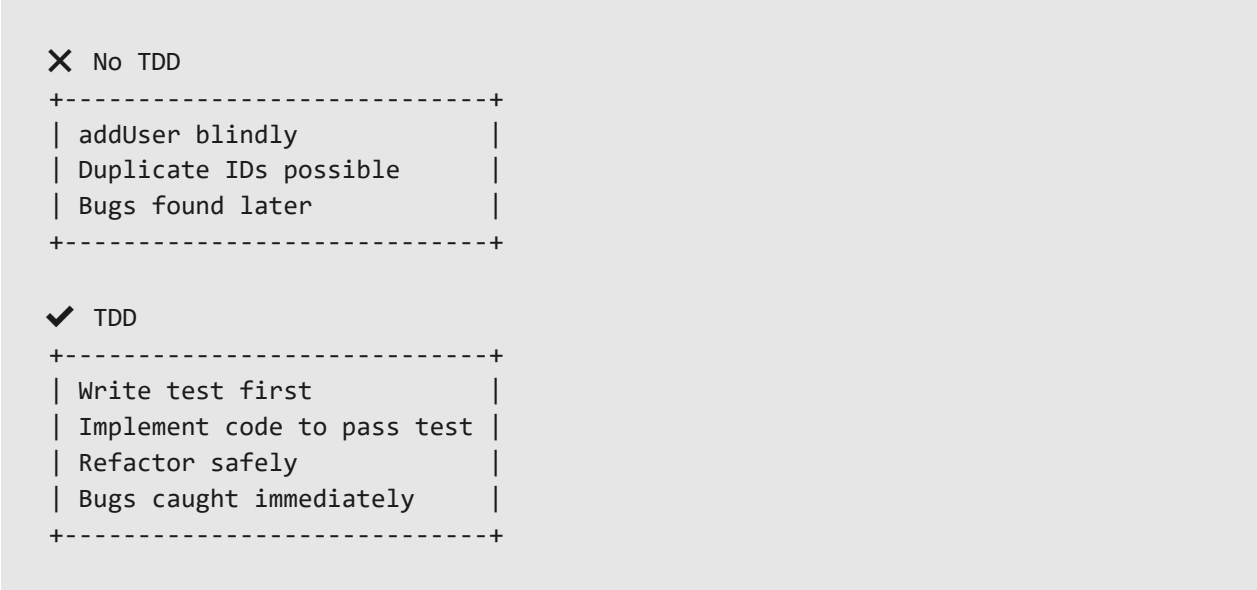
```
// Test first
function testAddUser() {
  const users = [{id: 1, name: "Alice"}];
  const newUsers = addUser(users, {id: 2, name: "Bob"});
  if (newUsers.length !== 2) throw new Error("User not added");
  console.log("Test passed!");
}

// Production code to pass test
function addUser(users, user) {
  const exists = users.some(u => u.id === user.id);
  if (exists) throw new Error(`User with id ${user.id} already exists`);
  return [...users, user];
}

// Run test
testAddUser();
```

Now:

- Duplicate IDs are prevented automatically.

- You know immediately when a new change breaks functionality.
- Encourages writing modular, testable code.

Illustration



Behavior-Driven Development (BDD)

Behavior-Driven Development is an evolution of TDD that emphasizes **specifying the behavior of software in a readable, human-friendly way**. It focuses on collaboration between developers, testers, and non-technical stakeholders to ensure the software does what users expect.

Definition
Write specifications (tests) that describe **expected behavior** in plain language. Code is then implemented to satisfy these behavior specifications.

Why it matters

- **Improves communication**: Everyone understands what the system should do.
- **Ensures correct behavior**: Tests are expressed as user scenarios.
- **Facilitates acceptance testing**: Non-technical stakeholders can validate behavior.
- **Encourages better design**: Thinking in terms of behavior often produces cleaner, modular code.

Bad Example (no BDD)
Code is written without considering expected behavior from the user's perspective:

```
// Function without clear behavior specification
function calculateDiscount(user) {
  if (user.membership) return 0.1;
  if (user.vip) return 0.2;
  return 0;
}

console.log(calculateDiscount({membership: true, vip: true})); // 0.1 -> unexpected
```

Problem:

- Ambiguous rules: Which discount applies?
- Confusing results for users.
- Harder to communicate intent to stakeholders.

Good Example (using BDD approach)

Write behavior specifications first in human-readable form, then implement code:

```
// Behavior specification
// "VIP users get 20%, members get 10%, VIP+Member gets 20%"

// Implement code to satisfy behavior
function calculateDiscount(user) {
  if (user.vip) return 0.2;
  if (user.membership) return 0.1;
  return 0;
}

// Test scenarios
function testDiscount() {
  const tests = [
    {input: {vip: true, membership: true}, expected: 0.2},
    {input: {vip: false, membership: true}, expected: 0.1},
    {input: {vip: false, membership: false}, expected: 0}
  ];

  tests.forEach(({input, expected}) => {
    const result = calculateDiscount(input);
    if (result !== expected) throw new Error(`Failed: ${JSON.stringify(input)}`);
  });
  console.log("All BDD tests passed!");
}

// Run tests
testDiscount();
```

Now:

- Rules are clear and behavior-driven.
- Everyone (developers, testers, stakeholders) can understand what is expected.
- Reduces ambiguity and unexpected results.
- Encourages modular and maintainable code.

Illustration