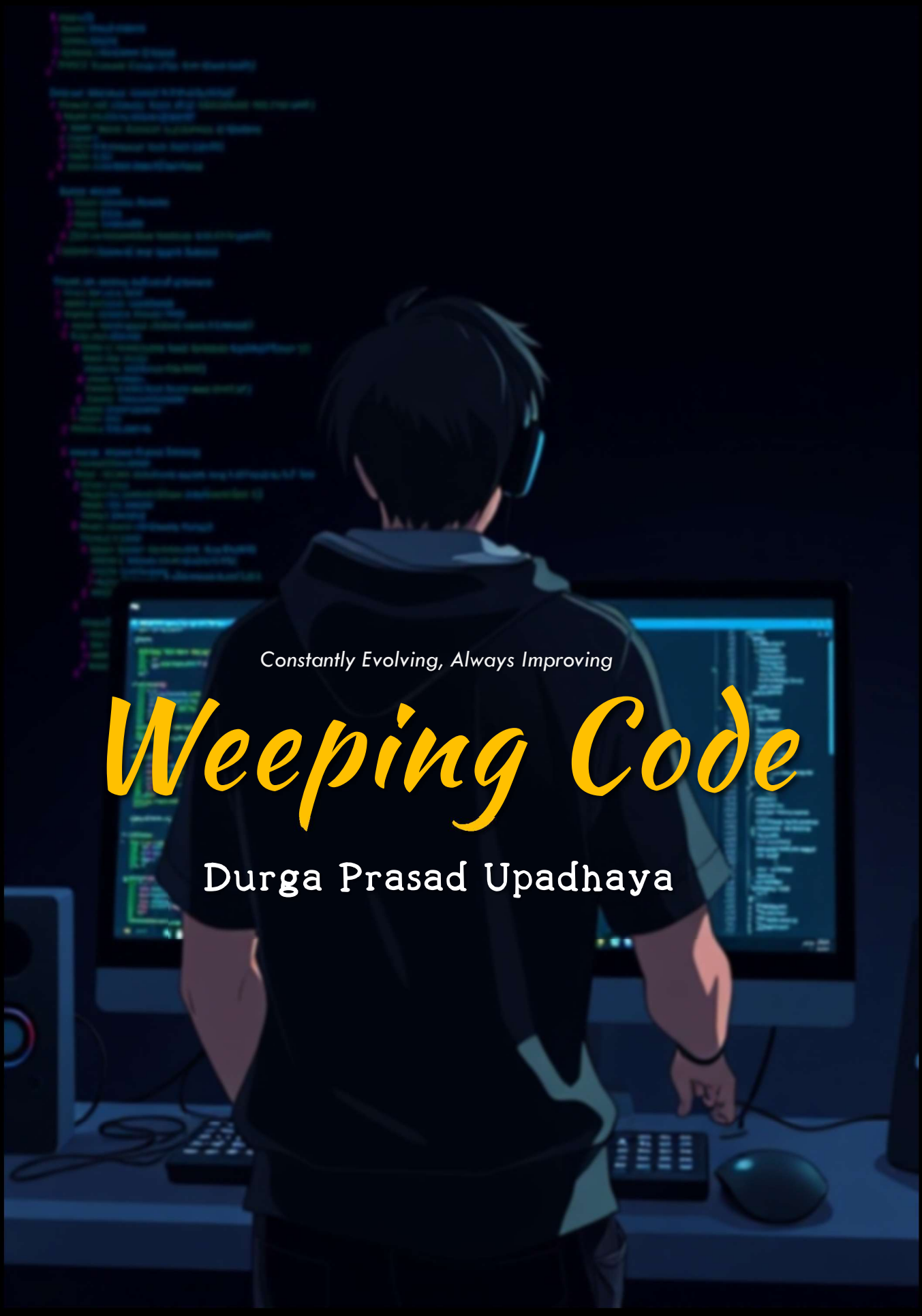


Constantly Evolving, Always Improving

Weeping Code

Durga Prasad Upadhaya

Copyright Information

Weeping Code

ISBN: 978-93-341-7293-5

For inquiries, to report an issue, or to request reproduction permissions, please contact:

dpupadhaya2000@gmail.com

Acknowledgements

Writing a book on Weeping Code has been great, on a journey of learning and sharing.
This book would not have been possible without the support of many.

My Teams
Working with these experienced teams has been a great experience.
From diverse experiences in various fields, I learned the power of teamwork and strength.
Thanks for challenging my ideas, engaging in meaningful debates, and helping me grow.

My Students
My instruction has reflected my educational journey.
The students' curiosity and perceptive inquiries have driven me to investigate the topic more effectively.

Open Source Community
The generosity of the open-source community in freely sharing knowledge has been a valuable guide on my journey.

Let us transform weeping code, which causes issues and inefficiencies,
*into '**happy code**'—clean, efficient, and a joy to work with.*

This book will guide you on this inspiring transformation journey.

PREFACE

What is the Main Idea of This Book?

Weeping Code is designed to help you identify the hidden issues in your code.

What might seem like minor problems can quickly escalate, impacting team morale, slowing down productivity, and even threatening the stability of your project.

I will walk you through common mistakes and share practical tips for avoiding them so your projects do not just survive—they thrive.

Who Should Read This Book?

This book is designed for anyone who aspires to write cleaner, more effective code:

- ***New Developers***
  Lay a strong foundation by cultivating solid coding habits right from the start.
- ***Learners and Students***
  Unlock the secrets to crafting precise, efficient, and maintainable code while mastering the art of software development.
- ***Tech Enthusiasts***
  Discover what distinguishes high-quality code, learn to identify errors, and ensure seamless project execution.

Weeping Code provides valuable tools such as code analysis techniques, best practices, and practical tips for ***writing better, more efficient code****.*

These tools will help you recognize and fix common errors, whether working alone or as part of a team.

Table of Contents

What Is Weeping Code?

'**Weeping code**' refers to code that is hard to read, maintain, and improve—leading to slow performance, hidden bugs, and fragile features.

Left unchecked, it only gets worse.

The good news? **You can turn it around.**

Addressing these common mistakes and **applying best practices** can enhance performance, eliminate bugs, and ensure reliability.

In this guide, I will cover 64 standard '**Weeping codes**' across seven categories,

with tips on code reviews, leveraging AI-generated coding tools, and keeping your code clean.

All examples are in **TypeScript**, ensuring clarity and relevance.

Let us transform "**Weeping Code**" into solid, maintainable, and efficient solutions.

It is worth the effort!

Foundational Structural Challenges

These minor problems in the codebase architecture can be fixed early to prevent more significant issues later.

- **Monolithic Classes** : Monolithic Class with Multiple Responsibilities.
- **Lengthy Functions** : Overly Long Functions.
- **Inconsistent Naming Conventions** : Identifier Naming,
- **Duplicate Code Segments** : Code Redundancy
- **Overly Nested If Statements** : Deeply Nested Logic.
- **Extensive Switch-Case Statements** : Violates the Open/Closed Principle.
- **Feature Envy** : Functionality might be misplaced.
- **Widespread Modifications** : Minor adjustments and widespread alterations.
- **Insufficient Abstraction** : Exposing Implementation details.
- **Excessive Static Members** : Avoiding over-reliance on Static Members.
- **Improper Inheritance** : Inheritance should be used judiciously.
- **Interface Segregation Violation :** Clients must implement methods they don't need.

Monolithic Class

Classes with excessive responsibilities become challenging to manage and maintain. Refactoring them into smaller, specialized units with clear roles can enhance modularity and clarity.

Weeping Code : Monolithic Class with Multiple Responsibilities

```
class UserManager {
  createUser(): void {
    // Logic to create user
  }

  deleteUser(): void {
    // Logic to delete user
  }

  sendEmail(): void {
    // Logic to send email
  }
}
```

This code violates the Single Responsibility Principle (SRP) because it manages multiple unrelated tasks.

Happy Code : Refactor into Smaller, Focused Classes

```
class UserCreator {
  createUser(): void {
    // Logic to create user
  }
}

class UserDeleter {
  deleteUser(): void {
    // Logic to delete user
  }
}

class EmailSender {
  sendEmail(): void {
    // Logic to send email
  }
}
```

Each class in this code has a single responsibility, which adheres to the Single Responsibility Principle (SRP).

Happy Code II : Dependency Injection or Service Layer

```
class UserCreator {
  createUser(userData: any): void {
    // Logic to create user
  }
}

class UserDeleter {
  deleteUser(userId: string): void {
    // Logic to delete user
  }
}

class EmailSender {
  sendEmail(email: string, message: string): void {
    // Logic to send email
  }
}

class UserService {
  private userCreator: UserCreator;
  private userDeleter: UserDeleter;
  private emailSender: EmailSender;

  constructor(
    userCreator: UserCreator,
    userDeleter: UserDeleter,
    emailSender: EmailSender
  ) {
    this.userCreator = userCreator;
    this.userDeleter = userDeleter;
    this.emailSender = emailSender;
  }

  createUserAndSendEmail(userData: any, email: string, message: string): void {
    this.userCreator.createUser(userData);
    this.emailSender.sendEmail(email, message);
  }

  deleteUserAndSendEmail(userId: string, email: string, message: string): void {
    this.userDeleter.deleteUser(userId);
    this.emailSender.sendEmail(email, message);
  }
}
```

Added service layer or dependency injection to decouple the logic further, enhancing flexibility and testability.

Lengthy Functions

Functions that perform multiple tasks can quickly become hard to test, maintain, and extend.

Weeping Code : Function with Multiple Responsibilities

```
function calculateRectangleProperties(length: number, width: number): string {
  if (length <= 0 || width <= 0) return "Length and width must be positive numbers.";
  const area = length * width;
  const perimeter = 2 * (length + width);
  return `Area: ${area}, Perimeter: ${perimeter}`;
}
```

This function violates the Single Responsibility Principle by validating input, calculating the area and perimeter, and preparing the result.

Happy Code : Focused Functions

```
function calculateRectangleProperties(length: number, width: number): string {
  try {
    const validationError = validateDimensions(length, width);
    if (validationError) throw new Error(validationError);
    const { area, perimeter } = getRectangleProperties(length, width);
    return formatProperties(area, perimeter);
  } catch (error) { return `Error: ${error.message}`; }
}

function validateDimensions(length: number, width: number): string | null {
  if (length <= 0 || width <= 0)
    return "Length and width must be positive numbers.";
  return null;
}

function getRectangleProperties(
  length: number,
  width: number
): { area: number; perimeter: number } {
  if (length <= 0 || width <= 0) throw new Error("Invalid dimensions.");
  return {
    area: length * width,
    perimeter: 2 * (length + width),
  };
}

function formatProperties(area: number, perimeter: number): string {
  return `Area: ${area}, Perimeter: ${perimeter}`;
}
```

Each function addresses a specific task, making the code more straightforward for testing and debugging. This enhances readability, maintainability, and testability.

Inconsistent Naming

Inconsistent naming conventions for variables, functions, and classes can confuse.

Weeping Code : Inconsistent Naming

```
function c(n: number): number {
  return n * n;
}

let a = 5;
let b = c(a);
console.log("The result is: " + b);
```

This may result in maintenance challenges and mistakes, as other developers might find it difficult to understand the code's purpose.

Happy Code : Descriptive Naming

```
function calculate square(number: number): number {
    return number * number;
}

const sideLength = 5;
const areaOfSquare = calculateSquare(sideLength);
console.log("The area of the square is: " + areaOfSquare);
```

Descriptive names such as calculateSquare, inputNumber, and squaredResult make the code more straightforward.

Duplicated Code Segment

Repetition of code throughout the codebase leads to inconsistencies and complicates maintenance.

Weeping Code

```
function printUserGreeting(name: string, age: number): void {
  const greeting = "Hello, " + name + "! You are " + age + " years old.";
  console.log(greeting);
}

function printUserFarewell(name: string, age: number): void {
  const farewell = "Goodbye, " + name + "! You were " + age + " years old.";
  console.log(farewell);
}
```

Code duplication exists in the greeting and farewell functions, which share similar logic and violate the DRY (Don't Repeat Yourself) principle.

Happy Code : Refactored for DRY

```
enum MessageType {
  Greeting = "greeting",
  Farewell = "farewell",
}

function formatUserMessage(
  name: string,
  age: number,
  type: MessageType
): string {
  const templates = {
    [MessageType.Greeting]: `Hello, ${name}! You are ${age} years old.`,
    [MessageType.Farewell]: `Goodbye, ${name}! You were ${age} years old.`,
  };
  return templates[type];
}

function printUserMessage(name: string, age: number, type: MessageType): void {
  console.log(formatUserMessage(name, age, type));
}
```

The message formatting logic is now centralised in the formatUserMessage function, which is used by printUserGreeting and printUserFarewell. This reduces duplication and improves code maintenance.

Deeply Nested If

Excessively nested conditionals or loops can make your code hard to follow and debug.

Weeping Code

```
function processOrder(order: any): void {
  if (order) {
    if (order.items && order.items.length > 0) {
      if (order.paymentProcessed) {
        console.log("Order processed:", order);
      } else {
        console.log("Payment not processed for the order.");
      }
    } else {
      console.log("No items in the order.");
    }
  } else {
    console.log("Invalid order.");
  }
}
```

This function has excessive nesting of conditionals, making the logic difficult to follow and prone to errors.

Happy Code : Flattened Logic with Early Return

```
function processOrder(order: any): void {
  if (!isValidOrder(order)) return;
  if (!hasValidItems(order)) return;
  if (!isPaymentProcessed(order)) return;

  console.log("Order processed:", order);
}

function isValidOrder(order: any): boolean {
  if (!order) {
    console.error("Invalid order received.");
    return false;
  }
  return true;
}

function hasValidItems(order: any): boolean {
  if (!order.items || order.items.length === 0) {
    console.error("Order does not contain any items.");
    return false;
  }
  return true;
}

function isPaymentProcessed(order: any): boolean {
  if (!order.paymentProcessed) {
    console.error("Payment for the order has not been processed.");
    return false;
  }
  return true;
}
```

Early returns are used to flatten the logic. If any condition fails, the function returns immediately, Avoiding deep nesting.

Happy Code II : with Helper Functions

```
class OrderValidator {
  constructor(private logger: Logger) {}

  isValidOrder(order: any): boolean {
    if (!order) {
      this.logger.logError("Invalid order received.", order);
      return false;
    }
    return true;
  }

  hasValidItems(order: any): boolean {
    if (!order.items || order.items.length === 0) {
      this.logger.logError("Order does not contain any items.", order);
      return false;
    }
    return true;
  }

  isPaymentProcessed(order: any): boolean {
    if (!order.paymentProcessed) {
      this.logger.logError(
        "Payment for the order has not been processed.",
        order
      );
      return false;
    }
    return true;
  }
}

class Logger {
  logError(message: string, order: any): void {
    console.error(`${message} Order details: ${JSON.stringify(order)}`);
  }
}
```

With an OrderValidator class, centralizing validation logic and allowing for easier maintenance and extension, with better error handling and logging.

Large Switch-Case Statements

A switch or case statement with too many branches **violates the Open/Closed Principle.**

Weeping Code

```
function sendNotification(notificationType: string, message: string): void {
  switch (notificationType) {
    case "EMAIL":
      console.log(`Sending email notification: ${message}`);
      break;
    case "SMS":
      console.log(`Sending SMS notification: ${message}`);
      break;
    case "PUSH":
      console.log(`Sending push notification: ${message}`);
      break;
    case "IN_APP":
      console.log(`Sending in-app notification: ${message}`);
      break;
    default:
      console.log("Unknown notification type.");
  }
}
```

The significant switch statement for notification types violates the Open/Closed Principle.

Happy Code

```
interface NotificationHandler {
  send(message: string): void;
}

class EmailNotificationHandler implements NotificationHandler {
  send(message: string): void {
    console.log(`Sending email notification: ${message}`);
  }
}

class SmsNotificationHandler implements NotificationHandler {
  send(message: string): void {
    console.log(`Sending SMS notification: ${message}`);
  }
}
```

```
class PushNotificationHandler implements NotificationHandler {
  send(message: string): void {
    console.log(`Sending push notification: ${message}`);
  }
}

class InAppNotificationHandler implements NotificationHandler {
  send(message: string): void {
    console.log(`Sending in-app notification: ${message}`);
  }
}

function getNotificationHandler(notificationType: string): NotificationHandler {
  switch (notificationType) {
    case "EMAIL":
      return new EmailNotificationHandler();
    case "SMS":
      return new SmsNotificationHandler();
    case "PUSH":
      return new PushNotificationHandler();
    case "IN_APP":
      return new InAppNotificationHandler();
    default:
      throw new Error("Unknown notification type");
  }
}

const notificationType = "EMAIL";
const handler = getNotificationHandler(notificationType);
handler.send("Welcome to our service!");
```

By applying polymorphism, each notification type has its handler class.
This allows the system to be easily extended with new notification types without modifying the existing code.

Happy Code II : Factory Pattern

```
class NotificationHandlerFactory {
  private static handlers: { [key: string]: NotificationHandler } = {};

  static registerHandler(type: string, handler: NotificationHandler): void {
    this.handlers[type] = handler;
  }

  static getHandler(type: string): NotificationHandler {
    const handler = this.handlers[type];
    if (!handler) {
      throw new Error(`Unknown notification type: ${type}`);
    }
    return handler;
  }
}

// Register handlers once during the initialization of your app
NotificationHandlerFactory.registerHandler("EMAIL", new EmailNotificationHandler());
NotificationHandlerFactory.registerHandler("SMS", new SmsNotificationHandler());
NotificationHandlerFactory.registerHandler("PUSH", new PushNotificationHandler());
NotificationHandlerFactory.registerHandler("IN_APP", new InAppNotificationHandler());

const handler = NotificationHandlerFactory.getHandler("EMAIL");
handler.send("Welcome to our service!");
```

Instead of a large switch block, a Factory or DI container can dynamically manage the instantiation of notification handlers.

Feature Envy

Feature Envy occurs when one class frequently accesses the methods or fields of another class to perform its operations, indicating that functionality might be misplaced.

Weeping Code

```
class Order {
  constructor(private totalAmount: number) {}
}

class DiscountService {
  // Feature Envy: Directly accessing the internal details of the Order class
  calculateDiscount(order: Order): number {
    if (order["totalAmount"] > 100) {
      // Accessing private property directly
      return 10; // 10% discount
    }
    return 0;
  }
}
```

The DiscountService class depends too much on the internal details of the Order class, which reduces cohesion.

Happy Code

```
class Order {
  constructor(private totalAmount: number) {}

  // Encapsulation: Provide a method to access the total amount safely
  getTotalAmount(): number {
    return this.totalAmount;
  }
}

class DiscountService {
    calculateDiscount(order: Order): number {
    const totalAmount = order.getTotalAmount(); // Encapsulated access
    if (totalAmount > 100) {
      return 10; // 10% discount
    }
    return 0;
  }
}
```

We preserve encapsulation and improve cohesion by moving the discount calculation logic into the Order class.

Widespread Modifications

Widespread Modifications happen when a slight change in one part of the system triggers changes in many other places. This indicates high coupling and poor cohesion.

Weeping Code : violates the Open/Closed Principle (OCP)

```
class Order {
  private weight: number;

  constructor(weight: number) {
    this.weight = weight;
  }

  calculateShippingCost(isExpress: boolean) {
    if (isExpress) return this.weight * 5; // Express shipping
    else return this.weight * 2; // Standard shipping
  }

  getWeight() {
    return this.weight;
  }
}

const order = new Order(10);
console.log(order.calculateShippingCost(true)); // Express shipping (50)
console.log(order.calculateShippingCost(false)); // Standard shipping (20)
```

Shipping logic is hardcoded in the Order class, which must be changed when new shipping methods are added.

Happy Code : Strategy Design Pattern

```
// ShippingStrategy.ts
interface ShippingStrategy {
  calculateShippingCost(weight: number): number;
}

class StandardShipping implements ShippingStrategy {
  calculateShippingCost(weight: number): number {
    return weight * 3;
  }
}

class ExpressShipping implements ShippingStrategy {
  calculateShippingCost(weight: number): number {
    return weight * 6;
  }
}

// Order.ts
class Order {
  constructor(private weight: number) {}

  calculateShipping(strategy: ShippingStrategy): number {
    return strategy.calculateShippingCost(this.weight);
  }

  getWeight() {
    return this.weight;
  }
}

const order = new Order(10);

const standardShipping = new StandardShipping();
console.log(standardShipping.calculateShippingCost(order.getWeight()));

const expressShipping = new ExpressShipping();
console.log(expressShipping.calculateShippingCost(order.getWeight()));
```

Shipping logic is separated into strategy classes, making extending or changing the shipping logic easier without modifying the Order class.

Insufficient Abstraction

When a system lacks abstraction layers, it exposes its implementation details, which leads to tight coupling and rigidity. Abstractions like interfaces or abstract classes are needed to decouple the system components.

Weeping Code

```
class Payment {
  constructor(private amount: number) {}

  processCreditCardPayment(cardNumber: string) {
    // Process payment using card number
  }

  processPaypalPayment(email: string) {
    // Process payment using PayPal
  }
}
```

This code directly couples payment processing logic to the Payment class, making it rigid and difficult to extend when new payment methods are introduced. Every time a new method is added, the Payment class must be modified, violating the Open/Closed Principle.

Happy Code : with Abstraction

```
// PaymentMethod.ts
interface PaymentMethod {
  processPayment(): void;
}

// CreditCardPayment.ts
class CreditCardPayment implements PaymentMethod {
  constructor(private cardNumber: string, private amount: number) {}

  processPayment() {
    // Process payment using credit card
    console.log(`Processing credit card payment of $${this.amount}`);
  }
}

// PaypalPayment.ts
class PaypalPayment implements PaymentMethod {
  constructor(private email: string, private amount: number) {}

  processPayment() {
    // Process payment using PayPal
    console.log(`Processing PayPal payment of $${this.amount}`);
  }
}

// PaymentService.ts
class PaymentService {
  process(paymentMethod: PaymentMethod) {
    paymentMethod.processPayment();
  }
}
```

By introducing the PaymentMethod interface, the PaymentService class can handle different types of payments without modifying its core logic. New payment methods can be added without altering existing code, adhering to the Open/Closed Principle.

Happy Code II : with Factory Method

```
// // PaymentFactory.ts
abstract class PaymentFactory {
  abstract createPaymentMethod(amount: number): PaymentMethod;
}

class CreditCardPaymentFactory extends PaymentFactory {
  constructor(private cardNumber: string) {
    super();
  }

  createPaymentMethod(amount: number): PaymentMethod {
    return new CreditCardPayment(this.cardNumber, amount);
  }
}

class PaypalPaymentFactory extends PaymentFactory {
  constructor(private email: string) {
    super();
  }

  createPaymentMethod(amount: number): PaymentMethod {
    return new PaypalPayment(this.email, amount);
  }
}

// PaymentService.ts (using the factory)
class PaymentService {
  process(paymentFactory: PaymentFactory, amount: number) {
    const paymentMethod = paymentFactory.createPaymentMethod(amount);
    paymentMethod.processPayment();
  }
}
```

Factory Method Pattern lets PaymentFactory handle payment method creation, enabling dynamic factory selection and easy swapping or adding. It decouples payment method instantiation from service logic, with PaymentService using the factory to generate them. Abstracting creation logic enhances flexibility, scalability, and future adaptability.

Excessive Static Members

Static methods can limit flexibility, complicate dependency injection, and make testing and mocking more difficult.

Weeping Code

```
class Logger {
  private static logLevel: string = 'INFO';

  static log(message: string): void {
    if (Logger.logLevel === 'INFO') {
      console.log(`INFO: ${message}`);
    } else if (Logger.logLevel === 'ERROR') {
      console.error(`ERROR: ${message}`);
    } else {
      console.log(`UNKNOWN LEVEL: ${message}`);
    }
  }

  static setLogLevel(level: string): void {
    Logger.logLevel = level;
  }
}
```

Over-reliance on static methods reduces flexibility and testing and violates dependency injection practices.

Happy Code I : Dependency Injection

```
// Logger.ts
interface Logger {
  log(message: string): void;
}

class ConsoleLogger implements Logger {
  log(message: string): void {
    console.log(message);
  }
}

class ErrorLogger implements Logger {
  log(message: string): void {
    console.error(message);
  }
}

// LoggingService.ts
class LoggingService {
  constructor(private logger: Logger) {}
  log(message: string): void {
    this.logger.log(message);
  }
}
```

```
const consoleLogger = new ConsoleLogger();
const loggingService = new LoggingService(consoleLogger);
loggingService.log('This is an info message'); // INFO: This is an info message const
errorLogger = new ErrorLogger();
loggingService.log('This is an error message'); // ERROR: This is an error message
```

Happy Code II : with Factory Pattern

```
//LoggerFactory.ts
interface LoggerFactory {
  createLogger(): Logger;
}

class ConsoleLoggerFactory implements LoggerFactory {
  createLogger(): Logger {
    return new ConsoleLogger();
  }
}

class ErrorLoggerFactory implements LoggerFactory {
  createLogger(): Logger {
    return new ErrorLogger();
  }
}

// LoggingService.ts (with LoggerFactory)
class LoggingService {
  constructor(private loggerFactory: LoggerFactory) {}

  log(message: string): void {
    const logger = this.loggerFactory.createLogger();
    logger.log(message);
  }
}

const consoleLoggerFactory = new ConsoleLoggerFactory();
const loggingService1 = new LoggingService(consoleLoggerFactory);
loggingService1.log('This is an info message');

const errorLoggerFactory = new ErrorLoggerFactory();
const loggingService2 = new LoggingService(errorLoggerFactory);
loggingService2.log('This is an error message');
```

Improper Inheritance

Improper inheritance use leads to tight coupling, poor modularity, and difficulty making changes or extending functionality. Inheritance should be used judiciously, and composition should be preferred for greater flexibility.

Weeping Code

```
class Animal {
    protected name: string;

    constructor(name: string) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a sound.`);
    }
}

class Dog extends Animal {
    constructor(name: string) {
        super(name);
    }

    speak() {
        console.log(`${this.name} barks.`);
    }
}

class Cat extends Animal {
    constructor(name: string) {
        super(name);
    }

    speak() {
        console.log(`${this.name} meows.`);
    }
}

const dog = new Dog("Rex");
dog.speak();  // Rex barks.

const cat = new Cat("Whiskers");
cat.speak();  // Whiskers meows.
```

While inheritance is used here, it's not the best animal representation approach.
It leads to a deep inheritance chain, which can make modifications harder.

Happy Code : Composition and Interfaces

```
interface Animal {
    speak(): void;
}

class Dog implements Animal {
    constructor(private name: string) {}

    speak() {
        console.log(`${this.name} barks.`);
    }
}

class Cat implements Animal {
    constructor(private name: string) {}

    speak() {
        console.log(`${this.name} meows.`);
    }
}

const dog = new Dog("Rex");
dog.speak();  // Rex barks.

const cat = new Cat("Whiskers");
cat.speak();  // Whiskers meows.
```

By using interfaces and composition, we simplify the design. Each animal implements the Animal interface, making the system more flexible and easier to maintain. We can now easily add more animals without modifying the existing classes.

Interface Segregation Violation

The Interface Segregation Principle means that clients should rely only on the methods they need. When interfaces become too large or clients must implement methods they don't need, it leads to unnecessary complexity.

Weeping Code

```
// Interface with too many methods
interface PaymentProcessor {
  processCreditCardPayment(cardNumber: string): void;
  processPaypalPayment(email: string): void;
  processBankTransfer(amount: number): void;
}

// OnlinePayment class implementing all methods
class OnlinePayment implements PaymentProcessor {
  public processCreditCardPayment(cardNumber: string): void {
      // Process credit card payment
  }

  public processPaypalPayment(email: string): void {
      // Process PayPal payment
  }

  public processBankTransfer(amount: number): void {
      // Process bank transfer payment
  }
}
```

The PaymentProcessor interface forces OnlinePayment to implement methods it doesn’t use (e.g., processBankTransfer), leading to unnecessary code and potential errors.

Happy Code

```
// Specific Interfaces
interface CreditCardPaymentProcessor {
  processCreditCardPayment(cardNumber: string): void;
}

interface PaypalPaymentProcessor {
  processPaypalPayment(email: string): void;
}

interface BankTransferPaymentProcessor {
  processBankTransfer(amount: number, accountNumber: string): void;
}

// Implementations
class CreditCardPayment implements CreditCardPaymentProcessor {
  public processCreditCardPayment(cardNumber: string): void {
    console.log(`Processing credit card payment for card: ${cardNumber}`);
  }
}

class PaypalPayment implements PaypalPaymentProcessor {
  public processPaypalPayment(email: string): void {
    console.log(`Processing PayPal payment for email: ${email}`);
  }
}

class BankTransferPayment implements BankTransferPaymentProcessor {
  public processBankTransfer(amount: number, accountNumber: string): void {
    console.log(`Processing bank transfer of $${amount} to account: ${accountNumber}`);
  }
}
```

By splitting the payment processing logic into smaller, more specific interfaces, each implementation must only deal with the required methods. This reduces unnecessary complexity and makes the system more flexible.

Happy Code II : Modularity, Dependency Injection, and Factory Pattern

```
// Specific Interfaces
interface CreditCardPaymentProcessor {
  processCreditCardPayment(cardNumber: string): void;
}

interface PaypalPaymentProcessor {
  processPaypalPayment(email: string): void;
}

interface BankTransferPaymentProcessor {
  processBankTransfer(amount: number, accountNumber: string): void;
}

// Individual Payment Implementations
class CreditCardPayment implements CreditCardPaymentProcessor {
  processCreditCardPayment(cardNumber: string): void {
    console.log(`Processing credit card payment for card: ${cardNumber}`);
  }
}

class PaypalPayment implements PaypalPaymentProcessor {
  processPaypalPayment(email: string): void {
    console.log(`Processing PayPal payment for email: ${email}`);
  }
}

class BankTransferPayment implements BankTransferPaymentProcessor {
  processBankTransfer(amount: number, accountNumber: string): void {
    console.log(`Processing bank transfer of $${amount} to account: ${accountNumber}`);
  }
}

// PaymentService with Dependency Injection
class PaymentService {
  constructor(
    private creditCardProcessor?: CreditCardPaymentProcessor,
    private paypalProcessor?: PaypalPaymentProcessor,
    private bankTransferProcessor?: BankTransferPaymentProcessor
  ) {}

  processCreditCard(cardNumber: string): void {
    if (!this.creditCardProcessor) {
      throw new Error("Credit card processor not available");
    }
    this.creditCardProcessor.processCreditCardPayment(cardNumber);
  }

  processPaypal(email: string): void {
    if (!this.paypalProcessor) {
      throw new Error("PayPal processor not available");
    }
    this.paypalProcessor.processPaypalPayment(email);
  }

  processBankTransfer(amount: number, accountNumber: string): void {
    if (!this.bankTransferProcessor) {
      throw new Error("Bank transfer processor not available");
    }
```

```
    this.bankTransferProcessor.processBankTransfer(amount, accountNumber);
  }
}

// Factory for Dynamic Configuration
class PaymentServiceFactory {
  static createPaymentService(paymentType: string): PaymentService {
    switch (paymentType) {
      case "CreditCard":
        return new PaymentService(new CreditCardPayment());
      case "PayPal":
        return new PaymentService(undefined, new PaypalPayment());
      case "BankTransfer":
        return new PaymentService(undefined, undefined, new BankTransferPayment());
      default:
        throw new Error("Invalid payment type");
    }
  }
}

// Example Usage with Dynamic Configuration
const userPaymentChoice = "PayPal";
const paymentService = PaymentServiceFactory.createPaymentService(userPaymentChoice);
paymentService.processPaypal("user@example.com"); // Processing PayPal payment
```

This code improves modularity, promotes dependency injection, and allows for runtime configuration of payment processors, making the system even more flexible and testable.

Complexity and Architectural Barriers

These hurdles make the codebase hard to maintain or scale effectively.

- **Excessive Null Checks –** Unnecessary null checks.
- **Weak Encapsulation –** Allowing external classes to access internal details.
- **Complex Dependency Injection –** Convoluted & hard-to-trace dependency chains.
- **Yoyo Problem** – Frequent back-and-forth navigation between files or components.
- **Overengineering –** Building overly complex solutions for simple problems.
- **Inconsistent Abstraction Levels –** Varying detail levels within the same context.
- **Nested Loops –** Layers of loops.
- **Numerous Parameters –** The method has too many parameters.
- **Rigid Design Patterns –** Overuse of patterns in inappropriate scenarios.

Excessive Null Checks

Constantly checking for null values can clutter code or reduce readability.

Weeping Code

```
interface Product {
  name?: string | null;
  price?: number | null;
  category?: string | null;
}

function displayProductInfo(product: Product | null): void {
  if (product === null || product.name === null || product.name === undefined)
    console.log("Product Name: Not available");
  else console.log(`Product Name: ${product.name}`);

  if (product === null || product.price === null || product.price === undefined)
    console.log("Price: Not available");
  else console.log(`Price: $${product.price}`);

  if (
    product === null ||
    product.category === null ||
    product.category === undefined
  )
    console.log("Category: Not available");
  else console.log(`Category: ${product.category}`);
}

displayProductInfo(null);
displayProductInfo({ name: "Laptop" });
```

The code checks if the product is null and if each field (name, price, category) is null or undefined in multiple places. It also introduces repetitive logic, which a more elegant solution could have avoided.

Happy Code

```
interface Product {
  name?: string | null;
  price?: number | null;
  category?: string | null;
}

function displayProductInfo(product: Product | null): void {
  const name = product?.name ?? "Not available";
  const price = product?.price ?? "Not available";
  const category = product?.category ?? "Not available";

  console.log(`Product Name: ${name}`);
  console.log(`Price: ${price}`);
  console.log(`Category: ${category}`);
}

displayProductInfo(null);
displayProductInfo({ name: "Laptop" });
```

By using optional chaining (?.) and nullish coalescing (??), the code becomes more concise and avoids repeated null checks.

Weak Encapsulation

Allowing external classes to access internal details directly weakens modularity and increases coupling, making the system harder to maintain and refactor.

Weeping Code

```
class BankAccount {
    public balance: number;

    constructor(balance: number) {
        this.balance = balance;
    }
}

const account = new BankAccount(1000);
console.log(account.balance); // Directly accessing internal state
```

The balance field is public, allowing external code to directly access and modify the internal state of the BankAccount class. This weakens the encapsulation.

Happy Code

```
class BankAccount {
    private balance: number;

    constructor(balance: number) {
        this.balance = balance;
    }

    public getBalance(): number {
        return this.balance;
    }

    public deposit(amount: number): void {
        if (amount > 0) this.balance += amount;
    }
}

const account = new BankAccount(1000);
console.log(account.getBalance()); // Accessing through a method
```

By making the balance field private and providing controlled access through getBalance() and deposit(), we enforce encapsulation and reduce direct external manipulation of the class's internal state.

Complex Dependency Injection

Excessive use of dependency injection can lead to convoluted and hard-to-trace dependency chains.

Weeping Code

```
class Logger {
    log(message: string): void {
        console.log(`Log: ${message}`);
    }
}

class EmailService {
    sendEmail(to: string, subject: string, content: string): void {
        console.log(`Email successfully sent to ${to} with subject "${subject}":
${content}`);
    }
}

class SmsService {
    sendSms(to: string, content: string): void {
        console.log(`SMS successfully sent to ${to}: ${content}`);
    }
}

class PaymentService {
    constructor(
        private logger: Logger,
        private emailService: EmailService,
        private smsService: SmsService
    ) {}

    processPayment(amount: number): void {
        this.logger.log(`Processing payment of $${amount}`);
        this.emailService.sendEmail("a@example.com", "Payment Confirmation", "Your payment
was successful.");
        this.smsService.sendSms("0123456789", "Payment processed successfully.");
    }
}
```

In this code, the PaymentService constructor injects multiple dependencies (Logger, EmailService, and SmsService). This makes the system harder to maintain and debug, as it becomes difficult to understand how components are related or what dependencies are involved.

Happy Code

```
interface Logger {
    log(message: string): void;
}

interface NotificationService {
    sendNotification(to: string, content: string): void;
}

class ConsoleLogger implements Logger {
    log(message: string): void {
        console.log(`Log: ${message}`);
    }
}

class EmailNotificationService implements NotificationService {
    sendNotification(to: string, content: string): void {
        console.log(`Email sent to ${to}: ${content}`);
    }
}

// PaymentService depends on abstractions
class PaymentService {
    constructor(
        private logger: Logger,
        private notificationService: NotificationService
    ) {}

    processPayment(amount: number): void {
        this.logger.log(`Processing payment of $${amount}`);
        this.notificationService.sendNotification("customer@example.com", "Your payment was
successful.");
    }
}

const logger = new ConsoleLogger();
const notificationService = new EmailNotificationService();
const paymentService = new PaymentService(logger, notificationService);
paymentService.processPayment(150);
```

Use dependency injection in moderation, and ensure the dependencies are clearly defined and well-scoped. Keep the injection chain as short and straightforward as possible. Consider using interfaces or abstract classes to define dependency contracts to reduce tight coupling.

Yoyo Problem (Frequent Back-and-Forth Navigation)

This occurs when reading and understanding code requires switching between different classes, modules, or files due to fragmented logic.

Weeping Code : Unnecessary Delegation

```
class Order {
    private item: Item;

    constructor(item: Item) {
        this.item = item;
    }

    getItemPrice(): number {
        return this.item.getPrice(); // Delegates to Item
    }
}

class Item {
    private pricing: Pricing;

    constructor(pricing: Pricing) {
        this.pricing = pricing;
    }

    getPrice(): number {
        return this.pricing.getPrice(); // Delegates to Pricing
    }
}

class Pricing {
    private price: number;

    constructor(price: number) {
        this.price = price;
    }

    getPrice(): number {
        return this.price;
    }
}

const pricing = new Pricing(1000);
const item = new Item(pricing);
const order = new Order(item);
console.log(order.getItemPrice()); // Retrieves price via multiple layers
```

This code delegates the responsibility of fetching item details from Order to the Item class, which introduces unnecessary interaction between the classes.

Happy Code : Direct Access to Properties

```
class Order {
    constructor(private item: Item) {}

    getItemPrice(): number {
        return this.item.price; // Direct access to price
    }
}

class Item {
    constructor(public price: number) {} // Price directly accessible
}

const item = new Item(1000);
const order = new Order(item);
console.log(order.getItemPrice()); // Direct retrieval
```

The Item class now directly holds the item details, and the Order class retrieves them without unnecessary delegation. This minimises the number of interactions between classes and simplifies the flow.

Happy Code II : Using Composition

```
class Order {
    constructor(private price: number) {}

    getItemPrice(): number {
        return this.price; // Direct price retrieval without Item or Pricing class
    }
}

const order = new Order(1000);
console.log(order.getItemPrice()); // Direct price retrieval
```

Keeps the code as flat and direct as possible, improving performance and understanding for small applications or MVPs.

Overengineering

Overengineering involves adding unnecessary features or complexity that do not solve real problems. This leads to bloated codebases that are harder to maintain and more difficult to scale.

Weeping Code : Overcomplicated Strategy Pattern

```
interface UserStrategy {
    getUserInfo(): string;
}

class NameOnlyUser implements UserStrategy {
    constructor(private userName: string) {}
    getUserInfo(): string {
        return `Name: ${this.name}`;
    }
}

class FullUserDetails implements UserStrategy {
    constructor(private userName: string, private useAge: number) {}
    getUserInfo(): string {
        return `Name: ${this. userName}, Age: ${this.userAge}`;
    }
}

class User {
    private strategy: UserStrategy;

    constructor(strategy: UserStrategy) {
        this.strategy = strategy;
    }

    getUserInfo(): string {
        return this.strategy.getUserInfo();
    }
}

const nameOnlyUser = new User(new NameOnlyUser("A"));
console.log(nameOnlyUser.getUserInfo());

const fullUserDetails = new User(new FullUserDetails("B", 30));
console.log(fullUserDetails.getUserInfo()); // Name: Bob, Age: 35
```

The method getAgeInMonths() adds unnecessary complexity to displaying the user's age. The code could have been simpler without this complexity.

Happy Code II : Minimalist and Direct

```
class User {
    constructor(private userName: string, private userAge?: number) {}
    getUserInfo(): string {
        return `Name: ${this.userName}${this.userAge ? `, Age: ${this.userAge}` : ''}`;
    }
}

const user1 = new User("A");
console.log(user1.getUserInfo());  // Name: A
const user2 = new User("B", 25);
console.log(user2.getUserInfo());  // Name: B, Age: 25
```

Unnecessary features are removed, and the code is simplified to focus on the immediate problem of displaying user information. This improves clarity and maintainability.

Inconsistent Abstraction Levels

When different parts of the codebase operate at various levels of abstraction within the same context, it becomes harder to understand and maintain.

Weeping Code : Mixing High-Level Logic with Low-Level Access

```
class UserService {
    fetchUser(userId: string) {
        // Directly calls a low-level API method
        return database.getConnection().query(`SELECT * FROM users WHERE id = ${userId}`);
    }

    updateUserEmail(userId: string, email: string) {
        // High-level logic combined with direct database access
        if (email.includes("@")) {
            database.getConnection().query(`UPDATE users SET email = '${email}' WHERE id =
${userId}`);
        }
    }
}
```

The UserService class mixes high-level business logic with low-level database access, making it harder to follow and maintain.

Happy Code : Clear Separation Between Business Logic and Data Access

```
// Data access layer
class UserRepository {
    getUserById(userId: string): Promise<User> {
        return database.query(`SELECT email FROM users WHERE id = ${userId}`);
    }

    updateUserEmail(userId: string, email: string): Promise<void> {
        return database.query(`UPDATE users SET email = '${email}' WHERE id = ${userId}`);
    }
}

// Business logic layer
class UserService {
    constructor(private userRepository: UserRepository) {}

    async updateEmail(userId: string, email: string): Promise<void> {
        if (email.includes("@")) {
            await this.userRepository.updateUserEmail(userId, email);
        }
    }
}

const userRepository = new UserRepository();
const userService = new UserService(userRepository);
```

The UserService class focuses only on high-level business logic, while UserRepository is responsible for database interactions. This separation of concerns ensures consistent abstraction levels.

Nested Loops

Nested loops can create performance bottlenecks and make code harder to maintain.

Weeping Code : Inefficient Nested Loops

```
function commonItems(arr1: number[], arr2: number[]): number[] {
    const items: number[] = [];
    for (const item1 of arr1) {
        for (const item2 of arr2) {
            if (item1 === item2) {
                commonItems.push(item1);
            }
        }
    }
    return items;
}

console.log(commonItems([1, 2, 3], [2, 3, 4])); // Output: [2, 3]
```

The code uses nested loops, leading to inefficient performance, especially with large arrays.

Happy Code : Optimized with a HashSet

```
function commonItems(arr1: number[], arr2: number[]): number[] {
    const set = new Set(arr1);
    return arr2.filter(item => set.has(item));
}

console.log(commonItems([1, 2, 3], [2, 3, 4])); // Output: [2, 3]
```

Using a HashSet for efficient lookups replaces the nested loop, improving performance and making the code more readable.

Happy Code II : Further Optimization with Early Exit

```
function commonItems(arr1: number[], arr2: number[]): number[] {
    const set = new Set(arr1);
    const common: number[] = [];

    for (const item of arr2) {
        if (set.has(item)) common.push(item);
    }

    return common;
}

console.log(commonItems([1, 2, 3], [2, 3, 4])); // Output: [2, 3]
```

Numerous Parameters

When a method has too many parameters, it can become hard to comprehend, test, and keep up with over time.

Weeping Code

```
class Order {
    constructor(
        private userId: string,
        private productId: string,
        private quantity: number,
        private shippingAddress: string,
        private paymentMethod: string
    ) {}

    processOrder() {
        // Order processing logic
    }
}
```

The Order class has too many parameters, making it hard to maintain. The constructor could be refactored.

Happy Code

```
interface OrderDetails {
    userId: string;
    productId: string;
    quantity: number;
    shippingAddress: string;
    paymentMethod: string;
}

class Order {
    constructor(private details: OrderDetails) {}

    processOrder() {
        // Order processing logic using this.details
    }
}
```

Encapsulating the parameters within a single OrderDetails object improves code maintainability and readability

Rigid Design Patterns

While design patterns can offer significant benefits, they should be used where appropriate. Rigid adherence to a pattern, just for the sake of using it, can introduce unnecessary complexity. Patterns should be applied when they solve specific problems and enhance clarity, not to fulfil an arbitrary rule.

Weeping Code

```
class Database {
    private static instance: Database;

    private constructor() { }

    public static getInstance() {
        if (!Database.instance) {
            Database.instance = new Database();
        }
        return Database.instance;
    }
}
```

The Singleton pattern is used without a real need, adding unnecessary complexity.

Happy Code

```
class Database {
    constructor() { }
}

const db = new Database();
```

In this case, a simple class with direct instantiation suffices without needing the Singleton pattern.

Performance and Scalability Issues

Problems that limit the system's ability to grow or operate efficiently.

- **Memory Leaks** – Poor memory management.
- **Suboptimal Algorithms** – Inefficient algorithms with high time/space complexity.
- **Inefficient Resource Use** – Overloading computational or hardware resources.
- **Concurrency Issues** – Leading to race conditions.
- **Caching Problems** – Ineffective caching strategies.
- **Database Bottlenecks** – Unoptimized database queries.
- **Non-Scalable Models** – Store everything in a single object.
- **Cold Start Problems**: Service initializes without optimization.

Memory Leaks

A memory leak occurs when memory that is no longer needed by the application is not released, causing the application to consume more memory over time.
If not addressed, this can lead to system crashes or significant degradation in performance.

Weeping Code

```
let users: string[] = [];

function addUser(user: string): void {
    users.push(user); // Memory grows indefinitely
}
```

The users array grows indefinitely as new users are added without removing old ones, leading to a memory leak. This causes the application to use more and more memory over time.

Happy Code

```
let users: Set<string> = new Set();
const MAX_USERS = 1000;

function addUser(user: string): void {
    users.add(user);
    if (users.size > MAX_USERS) {
        // Remove a random user or use a custom eviction strategy if needed
        const iterator = users.values();
        users.delete(iterator.next().value); // Evict one user
    }
}

function cleanupMemory(): void {
    users.clear(); // Explicitly clear all users when no longer needed
}
```

Using efficient data structures like Set, ensuring that memory is explicitly cleaned up when it's no longer needed, and implementing more advanced eviction strategies when necessary.

Suboptimal Algorithms

Inefficient algorithms with high time or space complexity can result in slow execution times and excessive resource consumption, particularly as data volumes increase. Enhancing algorithms is essential for boosting performance

Weeping Code

```
function bubbleSort(arr: number[]): number[] {
    let sorted = false;
    while (!sorted) {
        sorted = true;
        for (let i = 0; i < arr.length - 1; i++) {
            if (arr[i] > arr[i + 1]) {
                // Swap adjacent elements
                [arr[i], arr[i + 1]] = [arr[i + 1], arr[i]];
                sorted = false; // Mark as not sorted
            }
        }
    }
    return arr;
}
```

The bubbleSort algorithm is inefficient for large datasets due to its O(n^2) time complexity, which makes it slow as the array size increases.

Happy Code

```
function mergeSort(arr: number[]): number[] {
    if (arr.length <= 1) return arr; // Base case
    const mid = Math.floor(arr.length / 2);
    return merge(mergeSort(arr.slice(0, mid)), mergeSort(arr.slice(mid)));
}

function merge(left: number[], right: number[]): number[] {
    const result: number[] = [];
    while (left.length && right.length) {
        result.push(left[0] < right[0] ? left.shift()! : right.shift()!);
    }
    return result.concat(left, right); // Combine the sorted parts
}
```

The mergeSort algorithm is a more efficient sorting method with O(n log n) time complexity, providing faster performance for large datasets.

Happy Code II : Optimized Merge Sort

```
function mergeSort(arr: number[]): number[] {
    if (arr.length <= 1) return arr;
    const mid = Math.floor(arr.length / 2);
    const left = arr.slice(0, mid);
    const right = arr.slice(mid);

    return merge(mergeSort(left), mergeSort(right));
}

function merge(left: number[], right: number[]): number[] {
    const result: number[] = [];
    let leftIndex = 0;
    let rightIndex = 0;

    // Avoid using .shift() to prevent O(n) complexity
    while (leftIndex < left.length && rightIndex < right.length) {
        if (left[leftIndex] < right[rightIndex]) {
            result.push(left[leftIndex]);
            leftIndex++;
        } else {
            result.push(right[rightIndex]);
            rightIndex++;
        }
    }

    // Add remaining elements from both arrays
    return result.concat(left.slice(leftIndex), right.slice(rightIndex));
}

const arr = [5, 3, 8, 2, 1, 4];
console.log(mergeSort(arr));
```

Inefficient Resource Use

Excessive CPU, memory, or network resource consumption can slow down the system, resulting in poor user experience and server performance, especially under high-traffic conditions.

Weeping Code

```
function isPrime(num: number): boolean {
    if (num < 2) return false;
    for (let i = 2; i < num; i++)
        if (num % i === 0) return false;
    return true;
}
```

The isPrime function checks divisibility for every number from 2 to num-1, making it inefficient for large numbers.

Happy Code

```
function isPrime(num: number): boolean {
    if (num < 2) return false;
    for (let i = 2; i <= Math.sqrt(num); i++)
        if (num % i === 0) return false;
    return true;
}
```

This optimised version of the isPrime function significantly improves performance by iterating only up to the square root of the number, reducing the number of divisibility checks.

Concurrency Issues

Concurrency problems arise when multiple threads or processes simultaneously access shared resources, leading to race conditions, data corruption, or deadlocks.

Weeping Code : Race Condition

```
let counter = 0;

function incrementCounter(): void {
    counter += 1; // Potential race condition if accessed concurrently
}
```

In this case, if increment counter is called concurrently from multiple threads, it could lead to inconsistent values because multiple threads may overwrite each other's updates to counter.

Happy Code : Safe Handling

```
let counter = 0;

function incrementCounter(): void {
    const newCounter = counter + 1; // Calculate new counter value
    counter = newCounter;  // Update counter atomically
}
```

This solution uses atomic-like operations to ensure the counter value is safely updated even in a concurrent environment, preventing race conditions.

Happy Code II : Atomic Operations

```
import AsyncLock from 'async-lock';


const lock = new AsyncLock();
let counter = 0;

function incrementCounter(): Promise<void> {
    return new Promise((resolve, reject) => {
        lock.acquire('counterLock', () => {
            counter += 1;
            resolve();
        }, (err) => {
            reject(err);
        });
    });
}
```

The async-lock library ensures that only one thread can modify a counter at a time, locking the shared resource during the update to prevent race conditions.

Caching Problems

Caching failures occur when data that should be cached is not stored or is cached inefficiently. This leads to repeated database queries, increased latency, and unnecessary resource consumption.

Weeping Code

```
class UserDataCache {
  private cache: Map<string, string>;

  constructor() {
    this.cache = new Map();
  }

  getUserData(userId: string): string {
    if (this.cache.has(userId)) {
      return this.cache.get(userId)!;
    }
    const data = `User data for ${userId}`;
    this.cache.set(userId, data);
    return data;
  }
}

const weepingCache = new UserDataCache();
console.log(weepingCache.getUserData("123")); // Fetches and caches data
console.log(weepingCache.getUserData("123")); // Returns stale data from cache
```

In this example, the data is never updated once it is cached. If the data changes, the cache will not reflect the new information, leading to potential data staleness.

Happy Code

```
class UserDataCache {
  private cache: Map<string, { data: string; lastUpdated: number }>;
  private refreshInterval: number;

  constructor(refreshInterval = 60000) { // Refresh interval in milliseconds
    this.cache = new Map();
    this.refreshInterval = refreshInterval;
  }

  getUserData(userId: string): string {
    const currentTime = Date.now();

    // Check if data exists in cache and is fresh
    const cached = this.cache.get(userId);
    if (cached && currentTime - cached.lastUpdated < this.refreshInterval) {
      return cached.data;
    }

    // Fetch and update cache
    const data = `User data for ${userId}`;
    this.cache.set(userId, { data, lastUpdated: currentTime });
    return data;
  }
}

const userDataCache = new UserDataCache(30000); // Refresh every 30 seconds
console.log(userDataCache.getUserData("123")); // Fetches and caches data
console.log(userDataCache.getUserData("123")); // Returns cached data if within interval
setTimeout(() => {
  console.log(userDataCache.getUserData("123")); // Refreshes after interval
}, 31000);
```

This solution efficiently checks and updates the cache, reducing redundant database queries and improving system performance.

Database Bottlenecks

Unoptimized database queries, especially those that fetch unnecessary data or lack indexing, can slow response times and increase resource consumption.

Weeping Code

```
async queryDatabase(): Promise<User[]> {
  return await this.dataSource.getRepository(User).find({
    where: { active: true },
  });
}
```

This query fetches all columns from the users table, even though only a subset of the data is required, resulting in unnecessary resource usage and slower performance.

Happy Code

```
async queryDatabase(): Promise<{ userId: number; username: string }[]> {
  return await this.dataSource.getRepository("User")
    .createQueryBuilder("user")
    .select(["user.id AS userId", "user.name AS username"])
    .where("user.accountStatus = :accountStatus", { accountStatus: "active" })
    .getRawMany();
}
```

This optimized query fetches only the necessary columns (id, name), improving query performance by reducing data transfer and resource consumption.

Non-Scalable Models

A non-scalable data model might store everything in a single object or structure, leading to slow performance when the dataset grows.

Weeping Code

```
type User = {
  id: string;
  name: string;
  posts: {
    id: string;
    title: string;
    content: string;
    comments: {
      id: string;
      userId: string;
      comment: string;
    }[];
  }[];
};

const users: User[] = [
  {
    id: "1",
    name: "Mark",
    posts: [
      {
        id: "101",
        title: "My First Post",
        content: "Hello, world!",
        comments: [
          { id: "501", userId: "10", comment: "Great Post!" },
          { id: "502", userId: "11", comment: "Awesome Post" },
        ],
      },
    ],
  },
];

// Query Example: Find all comments for Mark's posts
const markComments = users
  .find((user) => user.id === "1")?.posts.flatMap((post) => post.comments) ?? [];
console.log(markComments);
```

This Code uses deeply nested structures, causing data duplication and making queries and updates inefficient as the dataset grows. Traversing large, coupled objects is slow and error-prone, leading to inconsistency and poor scalability.

Happy Code

```
// Data Model
type User = {
  id: string;
  name: string;
};

type Post = {
  id: string;
  userId: string;
  title: string;
  content: string;
};

type Comment = {
  id: string;
  postId: string;
  userId: string;
  comment: string;
};

// Example Data
const users: User[] = [
  { id: "10", name: "Elon" }, { id: "20", name: "Musk" },
];

const posts: Post[] = [
  { id: "101", userId: "1", title: "My First Post", content: "Hello, world!" },
];

const comments: Comment[] = [
  { id: "501", postId: "101", userId: "10", comment: "Great Post!" },
  { id: "502", postId: "101", userId: "11", comment: "Awesome Post" },
];

//Query: Fetch All Posts with Comments for a User
function getPostsWithComments(userId: string) {
  const userPosts = posts.filter((post) => post.userId === userId);

  return userPosts.map((post) => ({
    ...post,
    comments: comments.filter((comment) => comment.postId === post.id),
  }));
}

console.log(getPostsWithComments("1"));
```

This Code normalizes data, separating entities like users, posts, and comments into independent structures linked by IDs. This eliminates redundancy, simplifies queries, and ensures scalability, consistency, and maintainability as the dataset grows.

Cold Start Problems

When a function or service initializes without optimization, it delays response times, especially for serverless architectures.

Weeping Code

```
async fetchData(): Promise<User[]> {
  // Initializing a heavy service instance for every invocation
  const service = new HeavyService();
  return await service.getUsers();
}
```

In this code, every invocation incurs a delay due to the repeated initialization of the HeavyService, impacting performance.

Happy Code

```
const heavyServiceInstance = new HeavyService(); // Initialize once at startup

async fetchData(): Promise<User[]> {
  return await heavyServiceInstance.getUsers(); // Reuse the instance for faster
execution
}
```

This approach minimizes the latency caused by repeated initialization, improving the responsiveness of the service.

Happy Code II : Lazy Initialization with Singleton Pattern

```
class ServiceManager {
  private static instance: HeavyService;

  private constructor() {} // Private constructor to prevent direct instantiation

  static getInstance(): HeavyService {
    if (!ServiceManager.instance) {
      ServiceManager.instance = new HeavyService(); // Initialize once when required
    }
    return ServiceManager.instance;
  }
}

async fetchData(): Promise<User[]> {
  const service = ServiceManager.getInstance(); // Get the existing or new instance
  return await service.getUsers();
}
```

Security Weaknesses

Common vulnerabilities that expose applications to threats.

- **SQL Injection** – Failure to sanitise database inputs.
- **Weak Access Controls** – Improper role-based permissions.
- **Weak Cryptography** – Outdated cryptographic algorithms.
- **Unsafe Deserialization** – Remote code execution.
- **DoS/DDoS Risks** – Overload the server with requests.
- **XSS Attacks** – execute malicious JavaScript.
- **CSRF Attacks** - Actions on behalf of an authenticated user.
- **Clickjacking Threats** – Invisible iframe on a malicious site.
- **Insecure Redirects** – Allowing malicious redirections.
- **Weak Authentication** - without encryption or hashing.
- **Session Flaws** – Sessions remain active indefinitely.
- **Vulnerable Dependencies** – use of a deprecated library.

SQL Injection

SQL injection occurs when user inputs are improperly sanitised and directly included in SQL queries, allowing attackers to inject malicious SQL code.
This can lead to unauthorized access to or manipulation of a database.

Weeping Code

```
const accountId = req.query.accountId;
// Insecure: directly injecting user input into SQL query
const sql = `SELECT * FROM accounts WHERE account_id = ${accountId}`;
db.query(sql, (error, rows) => {
    if (error) throw error;
    console.log(rows);
});
```

Directly injecting user input (userId) into the SQL query without validation or sanitization exposes the application to SQL injection attacks. Malicious users can manipulate the query by inputting dangerous SQL code, potentially giving them unauthorized access to the database or corrupting its data.

Happy Code

```
const accountId = req.query.accountId;
const sql = 'SELECT * FROM accounts WHERE account_id = ?';
db.query(sql, [accountId], (error, rows) => {
    if (error) throw error;
    console.log(rows);
});
```

Parameterized queries use placeholders (e.g., ?) to incorporate user input into SQL statements safely. This approach ensures that input is handled strictly as data, eliminating the risk of executing it as SQL code. The database engine automatically escapes special characters in the input, which prevents attackers from injecting malicious SQL code. This approach ensures safe querying and mitigates the risk of SQL injection attacks.

Weak Access Controls

Broken access control occurs when applications fail to properly enforce user roles and permissions, allowing unauthorized users to access resources or perform actions they should not be allowed to.

Weeping Code

```
// No access control enforcement
if (user.isSuperAdmin) {
    db.query("SELECT * FROM confidentialInfo", (error, data) => {
        if (error) throw error;
        console.log(data);
    });
}
```

This Code directly checks user.isAdmin to grant access to sensitive data without a proper access control system, making it easy for unauthorized users to bypass the check.

Happy Code

```
// Middleware to enforce user roles and permissions
function authorizeRole(requiredRole) {
    return (req, res, next) => {
        const { role } = req.user;
        if (role !== requiredRole) {
            return res.status(403).json({ message: 'Access denied' });
        }
        next();
    };
}

function fetchConfidentialInfo(userId) {
    const query = 'SELECT * FROM confidentialInfo WHERE ownerId = ?';
    db.query(query, [userId], (error, records) => {
        if (error) throw error;
        console.log(records);
    });
}

// Route demonstrating secure access control with role validation and safe queries
app.get('/confidential', authorizeRole('superadmin'), (req, res) => {
    fetchConfidentialInfo(req.user.id);
    res.send('Confidential information retrieved successfully.');
});
```

This code uses a middleware function to check for specific permissions, ensuring only authorized users can access sensitive data. It also employs parameterized queries, preventing SQL injection by treating user input as data, not executable code.

Weak Cryptography

Broken cryptography occurs when weak or outdated cryptographic algorithms encrypt sensitive data, which attackers can easily crack.

Weeping Code

```
import * as crypto from 'crypto';

const hash = crypto.createHash('md5').update(userPassword).digest('hex');
```

MD5 is an outdated, weak hashing algorithm prone to collision and rainbow table attacks, which makes it insecure for password storage.

Happy Code

```
import * as crypto from 'crypto';

const salt = crypto.randomBytes(16).toString('hex');
const hash = crypto.createHmac('sha256', salt).update(userPassword).digest('hex');
```

Using SHA-256 with a salt, the code ensures that password hashes are unique, secure, and resistant to common cryptographic attacks, improving overall security.

Unsafe Deserialization

Deserializing data from untrusted sources without thorough validation can expose applications to security risks, such as remote code execution, where attackers inject malicious payloads. Malicious actors can tamper with serialized data to inject harmful payloads, taking advantage of vulnerabilities in the deserialization process to compromise the application's security

Weeping Code

```
// Deserializing untrusted data without validation
const serializedData = req.body.data;

// Direct deserialization of untrusted input
const user = JSON.parse(serializedData);
console.log(user);
```

Deserializing untrusted data directly exposes the application to various attacks, such as remote code execution, if the data contains crafted objects that exploit the deserialization process.

Happy Code

```
// Strict validation before deserialization
const serializedData = req.body.data;

// Validate the data before deserialization to ensure it's safe
if (!isValidData(serializedData)) {
    throw new Error('Invalid data');
}

// Safely deserialize trusted data
const user = JSON.parse(serializedData);
console.log(user);
```

By validating the incoming data before deserialization, we ensure that the system only processes safe, expected structures. This prevents attackers from embedding harmful code in the serialized data, protecting the system from remote code execution and other deserialization vulnerabilities.

Denial of Service (DoS/DDoS) Risks

Poor backend design can leave systems vulnerable to resource exhaustion through malicious attacks or inefficient resource handling, disrupting the service and degrading user experience.

Weeping Code

```
app.post('/processData', (req, res) => {
    // Processing without rate limiting or throttling
    performHeavyComputation(req.body);
    res.send('Data processed');
});
```

The lack of rate limiting makes the system vulnerable to DoS/DDoS attacks, where attackers can overload the server with requests, leading to degraded performance or service outages..

Happy Code

```
import * as rateLimit from 'express-rate-limit';

// Define rate limit for each IP
const time = 10 * 60 * 1000;
const limiter = rateLimit({
    windowMs: time, // Set a 10-minute window
    max: 100, // Restrict each IP to 100 requests per time window
});

app.use(limiter);

app.post('/processData', (req, res) => {
    performHeavyComputation(req.body);
    res.send('Data processed');
});
```

The implementation of rate limiting ensures that users cannot overload the system, preventing DoS and DDoS attacks by limiting the number of requests per user.

XSS Attacks

XSS vulnerabilities occur when user inputs are not properly sanitized or encoded before being rendered in the frontend, enabling attackers to inject malicious scripts.

Weeping Code

```
const userInput = req.query.userInput;
// Directly injecting user input into HTML
document.getElementById('output').innerHTML = userInput;
```

User input is injected into the DOM without sanitization or escaping, allowing attackers to execute malicious JavaScript in the user's browser.

Happy Code

```
import DOMPurify from 'dompurify';

// Example of input from the user
const userInput = req.query.userInput;

// Sanitize the user input to remove any malicious content
const sanitizedInput = DOMPurify.sanitize(userInput);

// Validate input if necessary (e.g., check for length, format)
if (!isValidInput(sanitizedInput)) {
    throw new Error('Invalid input');
}

// Render the sanitized, validated input as text
document.getElementById('output').textContent = sanitizedInput;
```

Proper sanitization, input validation, secure rendering, and security headers combine to create a robust defense against XSS attacks while maintaining simplicity and performance. By employing multiple layers of defense, the system ensures that if one protective measure fails, other safeguards continue to function and protect the application.

CSRF Attacks

CSRF vulnerabilities occur when attackers trick a user into performing unwanted actions on a web application where the user is authenticated, without validating their intentions.

Weeping Code

```
app.post('/updateProfile', (req, res) => {
    //Acting without CSRF protection
    updateProfile(req.body);
    res.send('Profile updated');
});
```

Lacks any CSRF protection, leaving it vulnerable to attacks where a malicious actor could perform actions on behalf of an authenticated user.

Happy Code

```
import * as csrf from 'csurf';
import * as express from 'express';

const app = express();

// Set up CSRF protection
const csrfProtection = csrf({ cookie: true });
app.use(express.urlencoded({ extended: true })); // For handling form data

// API endpoint to get CSRF token
app.get('/get-csrf-token', csrfProtection, (req, res) => {
  res.json({ csrfToken: req.csrfToken() });
});

// CSRF-protected route
app.post('/updateProfile', csrfProtection, (req, res) => {
  // Perform the profile update logic
  updateProfile(req.body);
  res.send('Profile updated');
});

function updateProfile(data) {
  // Profile update logic here
}
```

Implements CSRF protection using tokens to validate the authenticity of requests, ensuring that only legitimate user requests can trigger sensitive actions.

Clickjacking Threats

Clickjacking occurs when a website is embedded within an invisible iframe on a malicious site, tricking users into clicking on hidden elements, potentially executing unintended actions.

Weeping Code

```
// src/main.ts ( NestJS )
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // No clickjacking protection set
  app.use((req, res, next) => {
    // Missing protection header
    next();
  });

  await app.listen(3000);
}

bootstrap();
```

In this code, no protection is set to prevent your pages from being embedded in an iframe, which makes them vulnerable to clickjacking attacks.

Happy Code

Backend Section

```
// src/main.ts (NestJS)
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Set global headers to prevent clickjacking attacks
  app.use((req, res, next) => {
    // Prevent embedding in iframe for all pages
    res.setHeader('X-Frame-Options', 'DENY');
    // Additional layer of security
    res.setHeader('Content-Security-Policy', "frame-ancestors 'none'");
    next();
  });

  await app.listen(3000);
}

bootstrap();
```

This code sets the X-Frame-Options: DENY header to protect your pages from being embedded in an iframe.

```
// src/dashboard/dashboard.controller.ts (NestJS)
import { Controller, Get } from '@nestjs/common';

@Controller('api/dashboard')
export class DashboardController {
  @Get()
  getDashboardData() {
    return {
      title: 'User Dashboard',
      content: 'Welcome to your dashboard!',
    };
  }
}
```

Frontend Section

```
// pages/dashboard.tsx
import { GetServerSideProps } from 'next';

interface DashboardContent {
  title: string;
  description: string;
}

const DashboardPage = ({ dashboardContent }: { dashboardContent: DashboardContent }) => {
  return (
    <div>
      <h1>{dashboardContent.title}</h1>
      <p>{dashboardContent.description}</p>
    </div>
  );
};

// Custom getServerSideProps function with different naming
export const getDataForDashboard: GetServerSideProps = async () => {
  try {
    const response = await fetch('http://localhost:3000/api/dashboard');
    if (!response.ok) {
      throw new Error('Failed to fetch dashboard content');
    }
    const dashboardContent = await response.json();
    return { props: { dashboardContent } };
  } catch (error) {
    console.error(error);
    return {
      props: { dashboardContent: { title: 'Error', description: 'Unable to load dashboard
data.' } },
    };
  }
};

export default DashboardPage;
```

In Next.js, use getServerSideProps to fetch the dashboard data server-side. This will ensure the data is available when the page loads.

Insecure Redirects

Unvalidated redirects occur when an application redirects users to untrusted or unsafe destinations, potentially leading to phishing or malicious sites.

Weeping Code

```
app.get('/redirect', (req, res) => {
    const redirectUrl = req.query.redirectTo as string;
    // Redirecting without validating the URL, which can lead to unsafe destinations
    res.redirect(redirectUrl);
});
```

In this example, the application takes the redirectTo parameter from the query string and redirects the user to it without any checks. An attacker could manipulate the redirect query parameter, leading the user to an untrusted destination, such as a malicious site, instead of the intended location.

Happy Code

```
// Importing the express library and types for Request and Response in TypeScript
import express from 'express';
import { Request, Response } from 'express';

const app = express();

// List of allowed, trusted URLs
const validUrls = ['https://safe-site.com', 'https://another-safe-site.com'];

// Secure redirect endpoint
app.get('/redirect', (req: Request, res: Response) => {
    const redirectUrl = req.query.redirectTo as string;

    // Check if the redirect URL is in the list of allowed URLs
    if (validUrls.includes(redirectUrl)) {
        res.redirect(redirectUrl);
    } else {
        res.send('Invalid redirect URL');
    }
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

If the URL is not in the trusted list, show an error message or redirect to a default page in this way this code securely handles redirects by ensuring the target URL is in a trusted list. This mitigates risks like phishing, drive-by attacks, and malicious redirection, helping you maintain control over where your users are redirected.

Weak Authentication

Improper authentication occurs when authentication mechanisms are weak or incomplete, allowing attackers to bypass access controls and gain unauthorized access.

Weeping Code

```
app.post('/login', (req, res) => {
    // Destructuring the incoming request body
    const { username, password } = req.body;
    if (username === 'admin' && password === 'password123')
        res.send('Login successful');
    else res.send('Invalid credentials');
});
```

Hardcoding credentials and checking passwords in plain text without encryption or hashing is insecure.

Happy Code

```
import * as bcrypt from 'bcrypt';
import express from 'express';
import { Request, Response } from 'express';

const app = express();
app.use(express.json());

// Function to retrieve a user from the database (simulate with mock data)
const getUserFromDatabase = (username: string) => {
    const users = [
        { username: 'admin', password:
'$2b$10$U1DqZ5QeF2Z/0l7v1sV1bubK2nJGGfmW5hIk4J.TlXzph3tnKjlC6' }
    ];
    return users.find(user => user.username === username);
};

app.post('/login', (req: Request, res: Response) => {
    // Destructuring the incoming request body
    const { username, password } = req.body;
    const user = getUserFromDatabase(username);

    if (user && bcrypt.compareSync(password, user.password)) {
        res.send('Login successful');
    } else res.status(401).send('Invalid credentials');
});

app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

Passwords are hashed using bcrypt, protecting sensitive data. The password comparison is done securely using the hashed value stored in the database.

Session Flaws

Insufficient session expiration occurs when sessions remain active for too long, allowing attackers to hijack them if they are not properly invalidated.

Weeping Code

```
// Session remains active even after extended periods of inactivity
app.post('/login', (req, res) => {
    req.session.user = { username: 'john' };
    res.send('Login successful');
});

app.get('/profile', (req, res) => {
    if (req.session.user) res.send(`Welcome ${req.session.user.username}`);
    else res.status(401).send('Unauthorized');
});
```

There is no expiration, meaning sessions can remain active indefinitely, potentially vulnerable to hijacking.

Happy Code

```
// Enabling session timeout after some period of inactivity
app.use(session({
    secret: 'secretKey',
    resave: false,
    saveUninitialized: false,
    cookie: { maxAge: 15 * 60 * 1000 } //Session expires after 15 min of inactivity
}));

app.post('/login', (req, res) => {
    req.session.user = { username: 'john' };
    res.send('Login successful');
});

app.get('/profile', (req, res) => {
    if (req.session.user) res.send(`Welcome ${req.session.user.username}`);
    else res.status(401).send('Unauthorized');
});
```

The session is set to expire after 15 minutes of inactivity, reducing the risk of session hijacking.

Vulnerable Dependencies

Vulnerable dependencies occur when outdated or vulnerable libraries are used, leaving applications open to known exploits.

Weeping Code

```
import express from "express";
import request from "request"; // Insecure, deprecated package

const app = express();
app.use(express.json()); // Using built-in express JSON parsing

app.post("/submitData", (req, res) => {
  // Using a deprecated and insecure package for HTTP requests
  request.post(
    "http://example.com/api",
    { json: req.body },
    (err, response, body) => {
      if (err) {
        res.status(500).send("Error sending data");
      } else {
        res.send("Data submitted successfully");
      }
    }
  );
});

app.listen(3000, () => {
  console.log("Server running on port 3000");
});
```

The use of a deprecated and insecure HTTP request library (request) that may expose the application to vulnerabilities.

Happy Code

```
import express from "express";
import axios from "axios";
import dotenv from "dotenv";

dotenv.config();

const app = express();
app.use(express.json()); // Built-in Express JSON parsing

// Environment variable for external API URL
const API_URL = process.env.API_URL || "http://example.com/api";

app.post("/submitData", async (req, res) => {
  try {
    // Use axios to make a secure and modern HTTP request
    const response = await axios.post(API_URL, req.body);
    res.send("Data submitted successfully");
  } catch (err) {
    // Improved error handling with detailed message
    console.error("Error sending data:", err.message);
    res.status(500).send("Failed to submit data. Please try again later.");
  }
});

app.listen(3000, () => {
  console.log("Server running on port 3000");
});
```

The deprecated request package is replaced with axios, an actively maintained modern and secure HTTP request library. This ensures that the application uses a more secure dependency, reducing the risk of security vulnerabilities.

Error Management and Logging Practices

Proper handling of errors and logs is vital for debugging and compliance.

- **Unclear Error Messages –** Non-descriptive errors that confuse users.
- **Centralized Error Management –** Repeated error-handling logic.
- **Detailed Logging –** Logs without context.
- **Retry Logic –** Failing to handle transient errors.
- **Secure Logging –** Sensitive information in logs.
- **Error Prioritization –** No categorization of the error severity or type.
- **Uncaught Exceptions** - Failures to handle exceptions, leading to crashes.

Unclear Error Messages

Unclear error messages fail to provide useful information to the user or developer, making it difficult to understand what went wrong and how to resolve it.

Weeping Code : Generic and Unhelpful

```
try {
    someFunction();
} catch (e) {
    console.log("Something went wrong");
    // Generic error message without context
}
```

The error message doesn't provide useful context or details about the issue, making debugging harder for developers.

Happy Code : Developer-Friendly Logs

```
try {
    someFunction();
} catch (e) {
    if (e instanceof Error) {
        console.error(`Error in someFunction: ${e.message}`); // Clear and contextual
message
        console.error(e.stack); // Stack trace for debugging (in development mode)

        // User-friendly output
        console.log("An unexpected error occurred. Please try again later.");
    } else {
        console.error("An unexpected error occurred."); // Catch non-Error cases
        console.log("An unexpected error occurred. Please try again later.");
    }
}}
```

It includes contextual information (someFunction) and the error message, making debugging easier.

Centralized Error Management

Duplicating error-handling logic increases maintenance overhead and makes consistency difficult.

Weeping Code : Duplicated Logic

```
try {
    const result = fetchData();
} catch (error) {
    console.error('Error fetching data:', error);
}

try {
    const result = saveData();
} catch (error) {
    console.error('Error saving data:', error);
}
```

Repeated error-handling logic leads to duplication and inconsistency.

Happy Code

```
class AppError extends Error {
    constructor(public message: string, public context: string, public statusCode: number)
{
        super(message);
        this.name = 'AppError';
    }
}

const handleError = (error: AppError) => {
    const errorMessage = `${error.context}: ${error.message}`;
    console.error(errorMessage);

    if (process.env.NODE_ENV !== 'production') {
        console.error(error.stack);
    }

    // Optional: Send error to an external service
    // sendErrorToService(errorMessage, error.stack);
};

try {
    const result = fetchData();
} catch (error) {
    if (error instanceof AppError) handleError(error);
    else
        handleError(new AppError('Unknown error occurred', 'Error fetching data', 500));
}
```

Centralizes error-handling logic in handleError, reducing duplication and improving maintainability.

Detailed Logging

Logs without context make debugging issues difficult.

Weeping Code : Minimal, Unhelpful Logs

```
try {
    const data = processData(input);
} catch (error) {
    console.error('Error processing data');
}
```

No metadata is logged, making it hard to trace the error.

Happy Code : Structured Logs

```
interface LogContext {
    input: any;
    operation: string;
    timestamp: string;
    userId?: string;
}

const logError = (context: LogContext, error: Error) => {
    console.error(JSON.stringify({
        level: 'ERROR',
        timestamp: context.timestamp,
        operation: context.operation,
        userId: context.userId || 'N/A',
        input: context.input,
        message: error.message,
        stack: error.stack,
    }, null, 2));
};

const processDataWithLogging = (input: any) => {
    const context: LogContext = {
        input,
        operation: 'processData',
        timestamp: new Date().toISOString(),
        userId: 'user123',
    };

    try {
        return processData(input);
    } catch (error) {
        logError(context, error);
    }
};

processDataWithLogging({ name: 'Test' });
```

Provides metadata (input) to help trace the cause of the error.

Retry Logic

Blind retries can strain resources and create cascading failures.

Weeping Code

```
const fetchData = () => {
    return apiCall();
};
fetchData(); // No retry logic
```

Lacks retry logic, failing to handle transient errors.

Happy Code

```
const fetchDataWithRetry = async (retries = 3, delay = 1000, maxDelay = 3000) => {
  for (let attempt = 1; attempt <= retries; attempt++) {
    try {
      return await apiCall(); // Try fetching data
    } catch (error) {
      console.warn(`Attempt ${attempt} failed: ${error.message}`);
      if (attempt === retries) throw new Error(`Failed after ${retries} attempts:
${error.message}`);

      // Exponential backoff
      const currentDelay = Math.min(maxDelay, delay * 2 ** (attempt - 1));
      console.log(`Retrying in ${currentDelay}ms...`);
      await new Promise(resolve => setTimeout(resolve, currentDelay));
    }
  }
};

// Example usage
fetchDataWithRetry(5, 1000, 5000)
  .then(response => console.log("Data fetched successfully:", response))
  .catch(error => console.error(error.message));
```

Implements retry logic with limits and warnings for transient failures.

Secure Logging

Sensitive information in logs can expose the system to vulnerabilities.

Weeping Code

```
console.error(`Error occurred: ${error.stack}`); // Exposes stack trace
```

Logs sensitive information that could be exploited.

Happy Code

```
const logError = (error: Error, context: string = "") => {
  const sanitizedMessage = error.message.replace(/(token|password)=[^&]*/g, "[REDACTED]");

  const log = {
    level: "ERROR",
    timestamp: new Date().toISOString(),
    context,
    message: sanitizedMessage,
    stack: process.env.NODE_ENV === "development" ? error.stack : undefined,
  };

  console.error(process.env.NODE_ENV === "production" ? JSON.stringify(log) : log);

  // Optionally send to external service
  // sendToExternalService(log);
};

try {
  throw new Error("Database connection failed. token=12345");
} catch (error) {
  logError(error, "Database Connection");
}
```

Logs only necessary details, ensuring sensitive information is not exposed.

Error Prioritization

Without categorization, critical issues may be overlooked.

Weeping Code

```
console.error('Database error');
```

No categorization of the error severity or type.

Happy Code

```
// Enum for error severity levels
enum Severity {
    Critical = 'critical',
    Warning = 'warning',
    Info = 'info',
}

// Centralized logging function with error severity categorization and improved context
class Logger {
    static log(message: string, severity: Severity, context?: string) {
        const timestamp = new Date().toISOString();
        const formattedMessage = `[${severity.toUpperCase()}] [${timestamp}] ${context ?
`[${context}] ` : ''}${message}`;

        switch (severity) {
            case Severity.Critical:
                console.error(formattedMessage); // Critical issues logged as errors
                break;
            case Severity.Warning:
                console.warn(formattedMessage); // Warnings logged with a warning
                break;
            case Severity.Info:
                console.info(formattedMessage); // Informational messages logged with info
                break;
        }

        // Optionally, send to external logging system
        // sendToExternalSystem(formattedMessage);
    }
}

// Usage of the logging system with severity categorization
Logger.log('Database connection failed', Severity.Critical, 'Database');
Logger.log('Memory usage is high', Severity.Warning, 'Performance');
Logger.log('Application started successfully', Severity.Info, 'Startup');
```

Categorizes errors by severity, making it easier to prioritize critical issues.

Uncaught Exceptions

Unhandled errors are a silent killer of application reliability. They can cause the app to crash or behave unpredictably. Robust error-handling practices ensure that the system stays resilient and predictable.

Weeping Code

```
function processPayment(paymentDetails) {
    const paymentResponse = apiCall(paymentDetails);
    return paymentResponse;  // Unhandled errors if API fails
}
```

This function doesn't handle any potential errors, which can lead to crashes or unexpected behavior if the API call fails.

Happy Code

```
// Centralized error handling with structured logging and retry mechanisms if needed
function processPayment(paymentDetails: any) {
    try {
        const paymentResponse = apiCall(paymentDetails);  // Call to the external API
        return paymentResponse;
    } catch (error) {
        // Log the error with severity and context
        Logger.log(`Payment failed: ${error.message}`, Severity.Critical, 'Payment
Process');

        // Optional: Retry logic or other error recovery steps can be added here if
appropriate

        // Propagate the error after logging
        throw error;
    }
}
```

By adding error handling with a try-catch block, this code prevents the application from crashing and provides useful error logging for debugging.

Testing and Maintainability Best Practices

Improving test effectiveness and codebase longevity.

- **Lack of Unit Tests –** Untested functionality and potential bugs.
- **Long Test Suites –** When the test loses clarity.
- **Poor Test Coverage –** leaves crucial code paths untested.
- **Continuous Integration** – Integration of testing into CI/CD pipelines.
- **Mocking / Stubbing** – Unit Testing to simulate external dependencies
- **Testing Error Scenarios –** Simulating potential failures.

Lack of Unit Tests

One of the primary issues in software development is the absence of unit tests, which leads to untested functionality and potential bugs.

Weeping Code

```
class Calculator {
    add(a: number, b: number): number {
        return a + b;
    }

    subtract(a: number, b: number): number {
        return a - b;
    }
}
```

The above code lacks unit tests, making it prone to unnoticed errors.

Happy Code

```
class Calculator { … }

// Unit Tests using Jest
describe('Calculator', () => {
    let calculator: Calculator;

    beforeEach(() => {
        // Instantiating a fresh Calculator before each test
        calculator = new Calculator();
    });

    it('should add two numbers', () => {
        expect(calculator.add(2, 3)).toBe(5);
        expect(calculator.add(-2, 3)).toBe(1); // Edge case for negative numbers
        expect(calculator.add(0, 0)).toBe(0); // Edge case for zero
    });

    it('should subtract two numbers', () => {
        expect(calculator.subtract(5, 3)).toBe(2);
        expect(calculator.subtract(2, 3)).toBe(-1); // Edge case for negative result
        expect(calculator.subtract(0, 0)).toBe(0); // Edge case for zero
    });

    it('should handle non-numeric inputs gracefully', () => {
        expect(() => calculator.add('a' as any, 3)).toThrow('Invalid input');
        expect(() => calculator.subtract(5, 'b' as any)).toThrow('Invalid input');
    });
});
```

This example includes basic tests that ensure key functionalities are covered.

Long Test Suites

Long, disorganized test suites are difficult to maintain and scale.
Grouping tests logically enhances both readability and maintainability, making them easier to manage.

Weeping Code

```
describe('User Service', () => {
    it('handle user creation', () => {
        // Test creating user
    });

    it('handle user update', () => {
        // Test updating user
    });

    it('handle user delete', () => {
        // Test deleting user
    });

    // More tests go here
});
```

All tests are lumped into a single, large test suite. As more functionality is added, this becomes harder to maintain and scale, and the tests lose clarity.

Happy Code

```
describe('User Service - Create', () => {
    it('It should create a user successfully', () => {
        // Test creating user
    });
});

describe('User Service - Update', () => {
    it('It should update a user successfully', () => {
        // Test updating user
    });
});

describe('User Service - Delete', () => {
    it('It should delete a user successfully', () => {
        // Test deleting user
    });
});
```

Organizing tests into logical, focused groups for better readability and maintenance

Poor Test Coverage

Poor coverage leaves crucial code paths untested.

Weeping Code

```
describe('LoginService', () => {
    let loginService: LoginService;

    beforeEach(() => {
        loginService = new LoginService();
    });

    it('should return true for valid credentials', () => {
        expect(loginService.login('admin', 'password123')).toBe(true);
    });
});
```

The test only checks a single positive scenario (valid credentials).

Happy Code

```
describe('LoginService', () => {
    let loginService: LoginService;

    beforeEach(() => {
        loginService = new LoginService();
    });

    it('should return true for valid credentials', () => {
        expect(loginService.login('admin', 'password123')).toBe(true);
    });

    it('should return false for invalid credentials', () => {
        expect(loginService.login('user', 'wrongpassword')).toBe(false);
    });

    it('should throw an error for missing username', () => {
        expect(() => loginService.login('', 'password123')).toThrowError('Username and
password are required');
    });

    it('should throw an error for missing password', () => {
        expect(() => loginService.login('admin', '')).toThrowError('Username and password
are required');
    });

    it('should throw an error for both username and password missing', () => {
        expect(() => loginService.login('', '')).toThrowError('Username and password are
required');
    });
});
```

This version tests multiple paths, including errors and edge cases.

Continuous Integration

Seamless integration of testing into CI/CD pipelines ensures code quality checks occur automatically during each deployment cycle.

Weeping Code

```
npm run test
```

Happy Code

Create a package.json with separate test scripts for unit and integration tests

```
{
  "scripts": {
    "test:unit": "jest --runInBand tests/unit",  // Fast executable unit tests
    "test:integration": "jest --runInBand tests/integration"  // Slower integration tests
  }
}
```

We can use Husky to automatically run tests when certain Git actions are performed (e.g., before committing or pushing changes). This ensures that tests are automatically triggered before code changes are pushed.

Create a pre-commit or pre-push hook

```
npx husky add .husky/pre-commit "npm run test:unit"
npx husky add .husky/pre-push "npm run test:integration"
```

If you don't want to use Git hooks, you can also automate your tests using npm scripts

```
{
  "scripts": {
    "test:unit": "jest --runInBand tests/unit",
    "test:integration": "jest --runInBand tests/integration",
    "test": "npm run test:unit && npm run test:integration"  // Run all tests
  }
}
```

Simply run npm test to run both unit and integration tests sequentially.

CI Pipeline for Local Testing

```
npm run test:unit        # Fast unit tests
npm run test:integration # Slower integration tests
```

Mocking/Stubbing

Mocks and stubs are essential for isolating tests from external dependencies. This allows tests to focus purely on the logic being tested, helping to create reliable, deterministic tests.

Weeping Code

```
describe('API Service', () => {
    let apiService: ApiService;

    beforeEach(() => {
        apiService = new ApiService();
    });

    it('should fetch data and return expected result', async () => {
        const data = await apiService.fetchData();
        expect(data).toEqual({ key: 'value' });  // No mocking, relying on actual API call
    });

    it('should handle API failure', async () => {
        const data = await apiService.fetchData(); // No error handling or mock
        expect(data).toThrow('API Error'); // This will never throw an error
    });

    it('should return undefined when API is not mocked', async () => {
        const data = await apiService.fetchData(); // Relying on real API, which may not
exist
        expect(data).toBeUndefined();  // Default behavior can cause unexpected behavior
    });
});
```

Here in this code error handling, testing practices, and mocking are poorly implemented

Happy Code

```
describe('API Service', () => {
    let apiService: ApiService;
    let mockFetch: jest.Mock;

    beforeEach(() => {
        mockFetch = jest.fn();
        apiService = new ApiService();
        apiService.fetchData = mockFetch;
    });

    it('should fetch data and return expected result', async () => {
        mockFetch.mockResolvedValue({ key: 'value' });
        const data = await apiService.fetchData();
        expect(data).toEqual({ key: 'value' });
        expect(mockFetch).toHaveBeenCalledTimes(1);
    });

    it('should handle API failure', async () => {
        mockFetch.mockRejectedValue(new Error('API Error'));
        await expect(apiService.fetchData()).rejects.toThrow('API Error');
        expect(mockFetch).toHaveBeenCalledTimes(1);
    });

    it('should return default value when API is not mocked', async () => {
        const data = await apiService.fetchData();
        expect(data).toBeUndefined(); // Default behavior if no mock is set
    });
});
```

Mocking helps isolate the logic under test by replacing external dependencies, ensuring that tests are focused on the code's behavior.

Testing Error Scenarios

Error-handling mechanisms are crucial for robust application behavior. Automated tests ensure errors are handled as expected, even in failure situations.

Weeping Code

```
const handleError = (error: Error) => {
    console.error(error.message);  // Simply logs the error message
};

handleError(new Error('Test Error')); // No automated testing or validation
```

Lacks automated testing to validate error-handling scenarios.

Happy Code

```
import { expect } from 'chai';
import sinon from 'sinon';

describe('Error Handling', () => {
    let consoleErrorSpy: sinon.SinonSpy;

    beforeEach(() => {
        // Setup: Create a spy to monitor console.error calls
        consoleErrorSpy = sinon.spy(console, 'error');
    });

    afterEach(() => {
        // Cleanup: Restore the original console.error after each test
        consoleErrorSpy.restore();
    });

    it('should log the error message without throwing exceptions', () => {
        const error = new Error('Test Error');

        // Execute the error-handling function
        handleError(error);

        // Assert: Check that console.error was called with the correct error message
        expect(consoleErrorSpy).to.have.been.calledOnceWithExactly('Test Error');
    });

    it('should log custom messages for different error types', () => {
        const error = new TypeError('TypeError occurred');

        // Execute the error-handling function
        handleError(error);

        // Assert: Check that console.error was called with the TypeError message
        expect(consoleErrorSpy).to.have.been.calledOnceWithExactly('TypeError occurred');
    });
```

```
    it('should log error details including the error name and message', () => {
        const error = new ReferenceError('ReferenceError occurred');

        // Execute the error-handling function
        handleError(error);

        // Assert: Check that console.error was called with the error's name and message
        expect(consoleErrorSpy).to.have.been.calledOnceWithExactly('ReferenceError:
ReferenceError occurred');
    });

    it('should not log anything when no error is provided', () => {
        // Execute error handling with null (no error provided)
        handleError(null);

        // Assert: Ensure console.error was not called
        expect(consoleErrorSpy).to.not.have.been.called;
    });

    it('should handle and log unexpected error formats gracefully', () => {
        const error = { message: 'Unexpected error', toString: () => 'Custom error' };

        // Execute the error-handling function with a non-standard error object
        handleError(error as any);

        // Assert: Check that the custom message is logged correctly
        expect(consoleErrorSpy).to.have.been.calledOnceWithExactly('Custom error');
    });

    it('should correctly handle asynchronous errors', async () => {
        const error = new Error('Async Error');

        // Simulate an async function call that throws an error
        const asyncErrorHandling = async () => {
            throw error;
        };

        try {
            await asyncErrorHandling();
        } catch (e) {
            // Execute the error-handling function on catch
            handleError(e);

            // Assert: Ensure that the error was logged correctly
            expect(consoleErrorSpy).to.have.been.calledOnceWithExactly('Async Error');
        }
    });
});
```

Uses unit testing to validate error-handling logic.

The Power of Precision - Solving the Invisible Issues

It illustrates how minor oversights—trivial code modifications, unfounded assumptions, or mishandling of asynchronous processes—can trigger challenging cascading effects.

- **The Butterfly Effect -** unintended side effects.
- **The Code That Never Sleeps -** Async fun can result in unpredictable behaviour.
- **The Layers of Deception -** Obscure underlying issues.
- **The Ticking Clock -** Validate the event time.
- **The Infinite Loop -** Ensure loops terminate to prevent crashes.
- **Hidden Errors in Complex Logic** - Complex logic often hides bugs.
- **Optimizing Hidden Performance Issues -** Fix performance bottlenecks.
- **Handling Rare Cases -** Handle rare cases to prevent errors.
- **The Tightrope –** Balance speed and accuracy for optimal results.
- **The Tangle of Logic -** Tightly coupled code is hard to maintain.

The Butterfly Effect

Small code changes can have unexpected and far-reaching consequences, especially when dependencies or assumptions are overlooked. This phenomenon highlights the importance of careful testing and validation when modifying systems.

Weeping Code

```
function updateUserData(user) {
    // Directly mutating the user object can lead to unintended side effects
    .age += 1;
}
```

Directly mutating the user object can lead to unintended side effects.

Happy Code

```
function updateUserData(user) {
    // Ensure user object is valid before updating
    if (!user || typeof user.age !== 'number')
      throw new Error('Invalid user data');

    // Avoids mutation by creating a new user object
    const updateUser = { ...user, age: user.age + 1 };
    return updateUser;
}
```

Creates a new user object to prevent mutation, ensuring original data remains intact.

The Code That Never Sleeps

Asynchronous programming brings both flexibility and complexity. While it allows for concurrent operations, it also introduces challenges like race conditions, unpredictable behavior, and error handling issues. Proper management of asynchronous code is essential for ensuring that the system remains stable and predictable, even under heavy loads.

Weeping Code

```
async function fetchData() {
    const data = await apiCall();
    return processData(data); // Potential issues with concurrent calls
}
```

Lack of error handling can result in unpredictable behavior if the API call fails.

Happy Code

```
async function fetchData() {
    const timeout = setTimeout(() => {
        throw new Error('API call timed out');
    }, 5000); // 5-second timeout

    try {
        const data = await apiCall();
        if (!data) throw new Error('No data returned');

        const processedData = await processData(data);
        return processedData;
    } catch (error) {
        console.error('Error during fetch:', error.message);
        throw error;
    } finally {
        clearTimeout(timeout); // Ensure timeout is cleared
    }
}
```

Includes error handling to ensure that the function behaves predictably when no data is returned..

The Layers of Deception

Abstraction can simplify complex systems, but it may also obscure underlying issues. Diagnosing failures becomes increasingly difficult when critical problems are hidden behind layers of abstraction, leading to potential risks.

Weeping Code

```
function getUser () {
    return database.getUser (); // Assumes database always works
}
```

This code assumes the database operation will always succeed. If it fails, it can cause unhandled exceptions or unexpected behavior, leaving the system vulnerable.

Happy Code

```
async function getUser() {
    try {
        const user = await database.getUser();  // Await the asynchronous call
        return user;  // Return the user data if successful
    } catch (error) {
        console.error('Database error:', error.message);  // Logs the error message
        return null;  // Return null for a safe fallback, avoiding system failure
    }
}
```

This code introduces robust error handling, ensuring the system gracefully handles any failure. It logs the specific error, offering better insight into the problem and providing a safe fallback (returning null) to maintain system integrity.

The Ticking Clock

Time-related issues like race conditions or incorrect time calculations can cause critical application failures. Properly managing time in code is essential to avoid such errors and ensure smooth execution.

Weeping Code

```
function scheduleEvent(eventTime) {
    setTimeout(() => {
        triggerEvent(); // Timing issues can lead to unexpected behavior
    }, eventTime);
}
```

This code does not validate the event time, which could allow events to be scheduled in the past, leading to logical errors or unexpected behaviour. Without proper checks, the system may malfunction, causing severe issues.

Happy Code

```
function scheduleEvent(eventTime) {
    const now = Date.now();
    // Validate that eventTime is in the future
    if (eventTime < now) {
        console.error('Error: Event time must be in the future');
        return;
    }

    // Calculate delay and schedule the event
    const delay = eventTime - now;
    console.log(`Event scheduled for ${new Date(eventTime).toLocaleString()}`);
    setTimeout(() => {
        triggerEvent();
        console.log('Event triggered');
    }, delay);
}
```

Validates the event time to ensure it is in the future, preventing scheduling errors.

The Infinite Loop

Even simple code can lead to infinite loops that consume resources and crash the system if not properly managed. Loops must be ensured to terminate correctly to maintain performance and stability.

Weeping Code

```
function calculated(numbers) {
    let sum = 0;
    while (numbers.length > 0) {
        sum += numbers.pop(); // Potential infinite loop if 'numbers' is empty
    }
    return sum;
}
```

In this example, if the number is an empty array, the loop will not terminate, causing the application to hang.

Happy Code

```
function calculateSum(numbers) {
    if (!Array.isArray(numbers)) {
        throw new Error('Input must be an array');
    }

    let sum = 0;
    while (numbers.length > 0) {
        sum += numbers.pop();
    }

    return sum;
}
```

This code ensures the loop terminates safely, preventing infinite loops and unnecessary CPU usage.

Hidden Errors in Complex Logic

Complex logic often hides bugs, especially from intricate conditions or mismatches between expected and actual behavior.

Weeping Code

```
function processOrder(order) {
    if (order.status === 'paid' && order.items.length > 0) {
        // Process payment
    } else if (order.status === 'shipped' && order.items.length === 0) {
        // Mark as complete
    }
    // Bugs may appear here if 'status' or 'items' values are not checked thoroughly
}
```

In this scenario, the logic could fail silently if conditions are not entirely validated, leading to misprocessed orders.

Happy Code

```
function validateOrder(order) {
  if (!order) throw new Error("Order data is required");
  if (!Array.isArray(order.items) || order.items.length === 0) {
    throw new Error("Order must have at least one item");
  }
  if (!["paid", "shipped"].includes(order.status)) {
    throw new Error(`Invalid order status: ${order.status}`);
  }
}

function processOrder(order) {
  try {
    validateOrder(order);
    switch (order.status) {
      case "paid": // Process payment logic
      case "shipped": // Complete order logic
      default: throw new Error(`Unhandled order status: ${order.status}`);
    }
  } catch (error) {
    console.error("Error processing order:", error.message);
    throw error;
  }
}
```

This approach ensures thorough input validation, preventing bugs from hidden logic errors.

Optimizing Hidden Performance Issues

Performance problems often lurk in places developers don't immediately see—such as database queries or inefficient data structures. Identifying and addressing these bottlenecks is crucial for achieving optimal performance.

Weeping Code

```
function searchUsers(query) {
    const users = getAllUsers();  // Fetch all users, then filter them
    return users.filter(user => user.name.includes(query));
}
```

In this code, the getAllUsers() function retrieves all user data and then filters it in memory, leading to inefficiencies, especially with large datasets.

Happy Code

```
function searchUsers(query) {
    if (!query || query.trim() === '') {
        throw new Error('Search query cannot be empty.');
    }

    // Use parameterized queries to prevent SQL injection and improve security
    return database.query('SELECT * FROM users WHERE name LIKE ?', [`%${query.trim()}%`]);
}
```

This solution shifts the filtering to the database level, reducing memory usage and speeding up search operations using indexed columns.

Handling Rare Cases with Grace

Rare cases often break code unexpectedly. To avoid this, developers should design code that accounts for all potential scenarios, even the rare ones, ensuring stability under various inputs.

Weeping Code

```
function getUserAge(userId) {
    const user = getUserFromDatabase(userId);
    return user.age;  // Assumes user. age is always defined
}
```

If the user. age is undefined or null; this function will break, leading to unexpected behavior.

Happy Code

```
function getUserAge(userId) {
    if (!userId) {
        throw new Error('Invalid user ID');
    }

    const user = getUserFromDatabase(userId);

    if (!user) {
        throw new Error(`User with ID ${userId} not found`);
    }

    if (typeof user.age !== 'number' || user.age < 0) {
        throw new Error(`Invalid or missing age for user with ID ${userId}`);
    }

    return user.age;
}
```

This code gracefully handles the rare case where the user's age may be missing, throwing an error to avoid silent failures and provide clear feedback.

The Tightrope

In software development, there's often a trade-off between speed and accuracy. Focusing too heavily on one aspect at the expense of another can create unforeseen challenges. Achieving harmony between both is vital to delivering outstanding outcomes.

Weeping Code

```
function calculateAverageScore(scores) {
    return scores.reduce((acc, score) => acc + score) / scores.length;
}
```

While this function quickly calculates the average, it may produce inaccurate results if the scores array contains invalid or non-numeric values.

Happy Code

```
function calculateAverageScore(scores) {
    if (!Array.isArray(scores) || scores.length === 0) {
        throw new Error('Scores array must be a non-empty array');
    }

    // Filter valid numeric scores
    const validScores = scores.filter(score => typeof score === 'number' && !isNaN(score));

    if (validScores.length === 0) {
        throw new Error('No valid numeric scores to calculate the average');
    }

    // Calculate the average of valid scores
    const total = validScores.reduce((acc, score) => acc + score, 0);
    return total / validScores.length;
}
```

This version first filters out invalid values, ensuring accurate calculation while maintaining performance. It balances speed with accuracy, providing better results.

The Tangle of Logic

Complex interactions between modules can create tightly coupled code, making maintenance and testing difficult.

Weeping Code

```
function userDashboard(userId) {
    const user = getUserData(userId);
    const notifications = getUserNotifications(userId);
    const settings = getUserSettings(userId);
    return renderDashboard(user, notifications, settings);
}
```

This function combines various concerns—retrieving user data, notifications, settings, and rendering the dashboard—into one place, making it difficult to test and extend.

Happy Code

```
async function getUserData(userId) {
    try {
        return await User.findById(userId);
    } catch (error) {
        console.error('Error fetching user data:', error.message);
        throw new Error('Failed to fetch user data');
    }
}

async function getUserNotifications(userId) {
    try {
        return await notificationService.getNotifications(userId);
    } catch (error) {
        console.error('Error fetching notifications:', error.message);
        return []; // Fallback to an empty array
    }
}

async function getUserSettings(userId) {
    try {
        return await settingsService.getSettings(userId);
    } catch (error) {
        console.error('Error fetching settings:', error.message);
        return {}; // Fallback to default settings
    }
}
```

```
async function fetchDashboardData(userId) {
    const [user, notifications, settings] = await Promise.all([
        getUserData(userId),
        getUserNotifications(userId),
        getUserSettings(userId),
    ]);
    return { user, notifications, settings };
}

async function userDashboard(userId) {
    try {
        const dashboardData = await fetchDashboardData(userId);
        return renderDashboard(
            dashboardData.user,
            dashboardData.notifications,
            dashboardData.settings
        );
    } catch (error) {
        console.error('Error rendering dashboard:', error.message);
        return renderErrorPage(); // Render a fallback error page
    }
}
```

Decoupling the logic into independent functions makes the code easier to test, modify, and scale.

Harnessing GPT – A Balanced Approach to AI-Generated Code

AI tools like GPT are changing the game in software development. They make things quicker, smoother, and a lot more efficient. However, like with anything powerful, we have got to be careful about how we use it. Too much reliance on AI-generated code could lead to some severe problems.

The Pitfalls of AI-Generated Code

1. **Intellectual Property (IP) Risks**
   AI grabs information from all over, and sometimes that includes code someone else owns. This could mean you're unknowingly using someone else's copyrighted work, which could lead to legal issues.

2. **Backdoors**
   AI-generated code might look good on the surface, but you can't always see potential security holes or backdoors that could compromise your app.

3. **Legal and Compliance Issues**
   In some cases, the code generated might break privacy laws, like GDPR or HIPAA. A simple oversight could mean big legal problems down the road.

4. **Code Quality and Maintainability**
   Even though AI can generate code that works, it only sometimes aligns with your team's specific needs. If the code does not follow your internal standards, it can quickly become a maintenance nightmare..

5. **Operational Risks**
   Relying too much on AI can make developers less familiar with the code they're working with. This could make troubleshooting harder and make your team less flexible when changes are needed.

6. **Reputational Damage**
   A security flaw or a privacy violation from AI-generated code can hurt your company's reputation. One mistake could cost you the trust of your users.

How to Manage AI-Generated Code Effectively

To make sure you're using AI the right way and not letting it take over, here's what you can do:

1. **Validate and Audit the Code**
   Don't just trust AI-generated code blindly. Always go over it, check for security issues, and make sure it meets your standards.

2. **Follow Industry Best Practices**
   AI can help speed things up, but it still needs to follow the same rules. Make sure it fits in with your coding standards and aligns with what's expected in the industry.

3. **Use AI as a Helper, Not a Replacement**
   AI is great at handling repetitive tasks, but it can't replace human creativity and decision-making. Think of it as a tool that boosts your productivity rather than replacing the need for skilled developers.

4. **Ensure Legal Compliance**
   Before using AI-generated code, double-check that it doesn't violate any laws or privacy rules. Regular audits can help make sure you're not at risk of any legal issues.

The Art of Code Simplification – Enhancing Existing Codebases

Technical debt grows with a project's size, making its management and maintenance even more complex.

Code refactoring, which involves restructuring the code without changing its output, is necessary to maintain code scalability, maintainability, and adaptability.

Why Refactor Code?

Refactoring software goes beyond tidying; it also involves making it do.
This is why such an activity is so important:

1. **Efficiency**
   Best of all, optimization opportunities often arise during refactoring.
   Sometimes, a hash map or more sophisticated data structure can dramatically improve performance by replacing poorly performing nested loops.

2. **Structure and Clarity**
   Eliminating code smells such as extended methods and unnecessary logic can enhance code clarity and comprehensibility.

3. **Saves money and time**
   A clean codebase results in the rapid release of bug fixes and feature enhancements.

4. **A reduction in technological responsibilities**
   Ongoing refactoring pays off the technical debt. It also keeps the code quality high and prevents the repeated occurrence of specific problems.

5. **Staying Up to Date**
   Repeated refactoring removes software staleness and ensures the program is always compatible with evolving industry standards, new libraries, and frameworks.

When and How to Refactor?

Refactoring should be deliberate and informed.
Signs that a codebase needs refactoring and steps to approach it.

1. **Identify Code Smells**
   Identifying these problems is the initial step toward making progress.

2. **Refactor Incrementally**
   Make small, incremental changes to minimize bugs and allow testing and validation.

3. **Ensure Comprehensive Test Coverage**
   Automated tests are a safety net, ensuring changes do not introduce regressions or break functionality.

4. **Document Changes Clearly**
   Maintain clear documentation of refactoring efforts. This helps current and future developers understand the rationale behind changes.

5. **Target Performance Optimization**
   Refactoring is an opportunity to optimize algorithms and streamline logic.
   For example, improving database queries or simplifying complex loops enhances performance.

6. **Remove Dead Code**
   Remove unused variables, functions, or classes. Unused code introduces unnecessary complexity and heightens the likelihood of errors.

Code Review Efficiency – Optimizing for Maximum Impact

Code reviews maintain good source code and allow for timely catching of improvements. A proper review process may improve relationships between team members, facilitate communication of needs, significantly avoid rework, and speed up review cycles.

Reduce Review Cycles

Requirements must be documented before development starts to reduce needless changes during code reviews.
The development remains on track with the early alignment of expectations, requiring fewer adjustments in the pull requests.

Leverage Tools to Streamline Reviews

Automated tools are vital in speeding the process of code reviews.
These technologies enhance efficiency and accuracy by identifying prevailing errors and maintaining code standards, allowing human reviewers to focus on more complex or contextual issues.

Recommended Tools

- ESLint: It detects syntactic errors before review.
- SonarQube: It identifies weaknesses and quality-related issues in the code, improving the maintainability of the code.
- Prettier: Preserves code formatting, minimizing aesthetic adjustments.
- Husky: It identifies problems before they commit by allowing automated pre-committing inspections, including testing and linting.

Swift and Constructive Feedback

- Swift: Respond promptly to suggest simplifying complex functions.
- Constructive- Refactoring for reusability without diminishing the developer's initial efforts.
- Integrated: Recognize commendable performance while providing recommendations for enhancement (e.g., "Excellent work! Consider improving error handling here for greater robustness.")

Limit Review Scope

Small, concentrated commits in code generally create a more efficient review. First, dividing significant changes into smaller feasible pieces is a strategy that encourages fast, focused feedback that can drive progress.

Prioritized important issues First.

From the beginning of its evaluation process, it tackles defects, performance-related issues, security flaws, and the lack of tests.
These focal points enhance the codebase's quality, safety, and consistency.

Major Problems:

- Modifications to the database would be more of data corruption, which takes precedence over alterations deemed non-essential or are done merely for aesthetics.

- Optimization for performance is most important when making algorithm changes or increasing efficiency before making nonessential changes.

- Fix security bugs like SQL injection and XSS before examining code style or structure.

- This would be a vital step when improving the less critical code, including all the edge cases and having a solid methodology for error handling.

- Testing is essential in the workflows that should be done and edge cases when focusing on aesthetic improvements.

git commit -m "Happy Code"

"Weeping Code" reminds us that having **clean and maintainable code** is a journey, not an event.

It demands focus, discipline, and a never-ending commitment to improvement.

Adherence to the principle of **Continuous Improvement** allows coding standards to improve with each iteration continually.

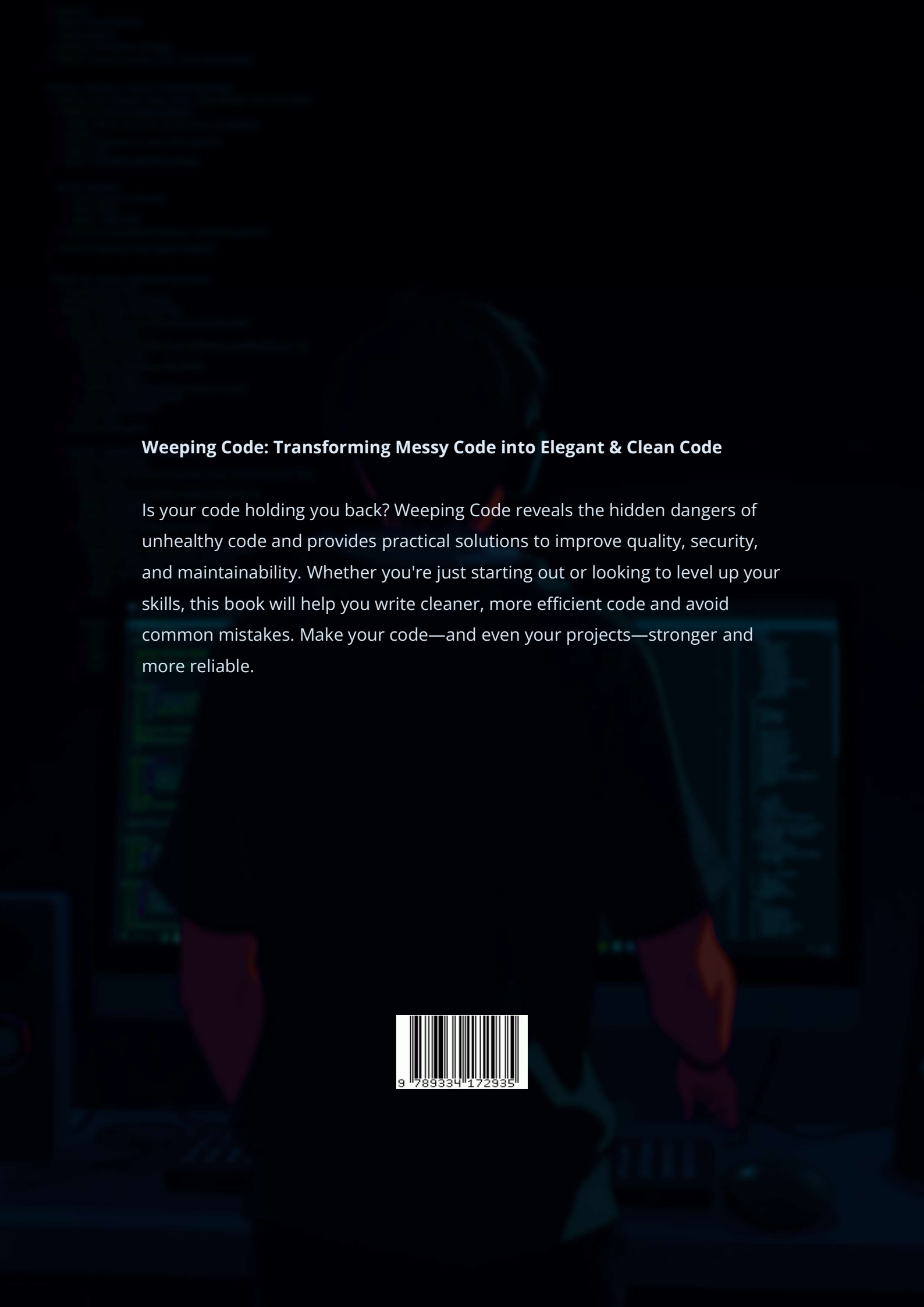
Besides this, **Continuous Discussion** promotes the culture of teams that do not hide knowledge behind any veil but instead share it, exchange ideas, and grow together.

This series provides action items for making **existing code cleaner, more elegant, and maintainable**.

It's a continuous process, and that'll make a difference.

With every such improvement, we get closer to realising our potential and satisfaction in the sheer act of coding to build and maintain.

Let's begin this journey of change together and **commit to writing happy code** for us and our successors.



Weeping Code: Transforming Messy Code into Elegant & Clean Code

Is your code holding you back? Weeping Code reveals the hidden dangers of unhealthy code and provides practical solutions to improve quality, security, and maintainability. Whether you're just starting out or looking to level up your skills, this book will help you write cleaner, more efficient code and avoid common mistakes. Make your code—and even your projects—stronger and more reliable.

9 789334 172935