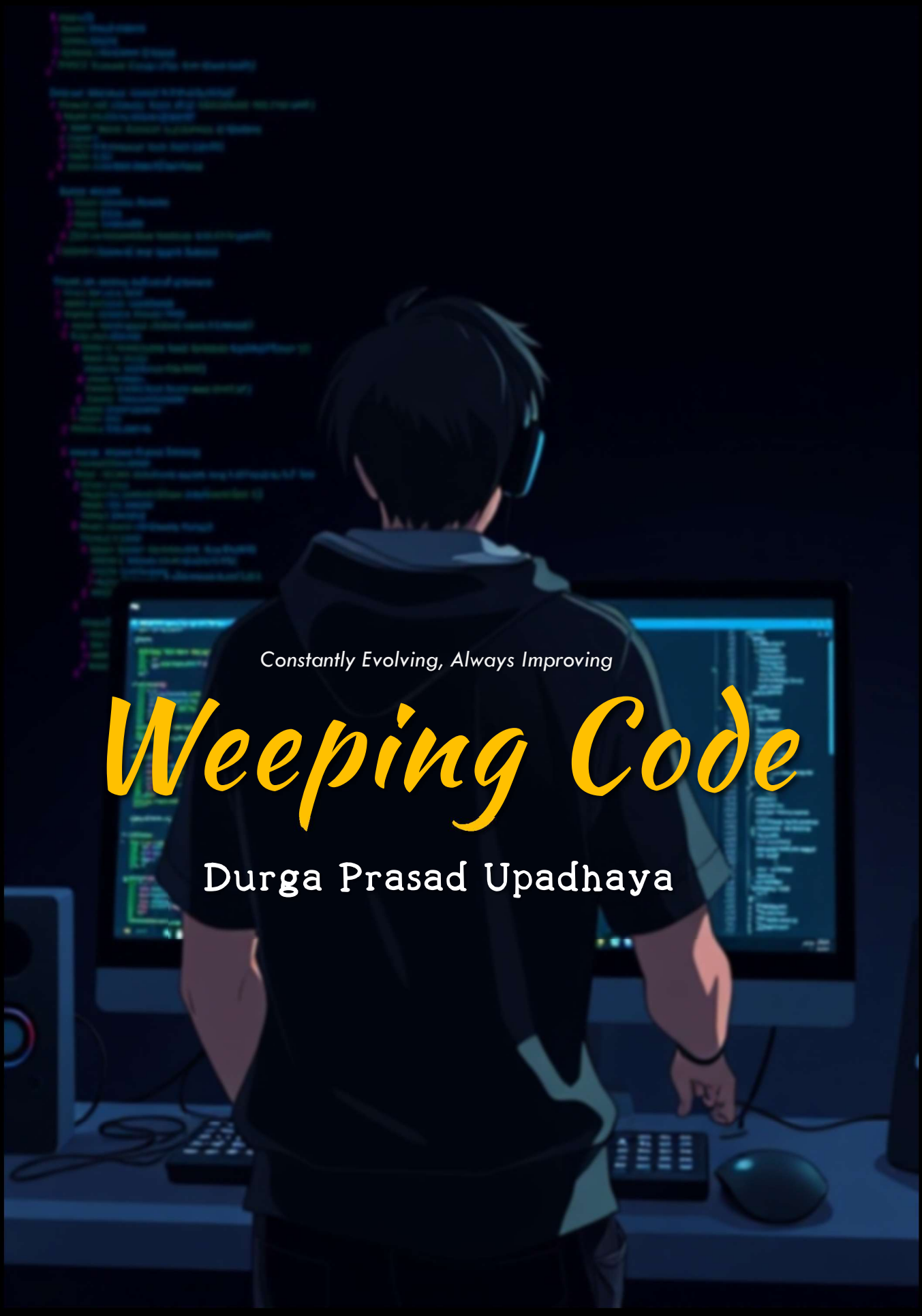

Constantly Evolving, Always Improving

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.