Introducing Debugging
Unveiling the Power of Code Inspection

Center for Music in the Brain, Aarhus University

May 23, 2024

Slides handouts

1 Clean code

Definition

Clean code is code that is easy to read, understand, and maintain. It focuses on simplicity and clarity. When we write clean code, we ensure that anyone else who reads our code, including our future selves, can easily comprehend its logic and purpose.

- **Readable:** Clean code uses descriptive names for variables, functions, and classes, making it easy to follow the program's flow.
- **Simple and Clear:** It avoids unnecessary complexity and keeps things as simple as possible while still solving the problem.
- **Well-Structured:** Organizing code into small, manageable functions or classes with a single responsibility helps in understanding and maintaining it.
- **Consistent Naming Conventions:** Using consistent naming conventions across the codebase reduces confusion and errors.
- **Well-Commented:** Comments should explain the 'why' behind complex or non-obvious code. However, clean code strives to be self-explanatory where possible.
- **Modular:** Breaking down functionality into discrete, reusable modules or functions promotes reusability and easier testing.
- **Testable:** Clean code is designed to be easily testable, which helps in catching bugs early.
- **Adheres to Best Practices:** Following principles like DRY (Don't Repeat Yourself) and SOLID principles ensures that our code remains maintainable and scalable.

---

2 Good coding practices

- **Meaningful Variable and Function Names:** Choose names that clearly describe the purpose or usage of the variable or function.

✗ What Not to Do

```
def f(x):
    return x * x
a = 10
result = f(a)
```

Here we have vague names like `f`, `x`, and `a` that do not convey any meaning about their purpose.

✓ What to Do

```
def square(number):
    return number * number
side_length = 10
area        = square(side_length)
```

This example uses descriptive names like `square`, `number`, `side_length`, and `area` that make the code's intent clear.

- **Consistent Formatting and Style:** Adopt a consistent style guide for formatting code, including indentation, spacing, and bracket placement.

✗ What Not to Do

```
def my_function():print("Hello");print("World")
```

Inconsistent formatting makes the code hard to read and understand.

✓ What to Do

```
def my_function():
    print("Hello")
    print("World")
```

Consistent indentation and spacing improve readability and maintainability.

- **Commenting and Documentation:** Write clear comments and documentation to explain the purpose and functionality of the code, especially for complex logic.

✗ What Not to Do

```
def calculate():
    a = 3.14 * r * r # calculation
```

This comment is not helpful because it simply restates what the code does without providing additional context.

✓ What to Do

```
def calculate_area_of_circle(radius):
    """
    Calculate the area of a circle given its radius.

    Args:
    radius (float): The radius of the circle

    Returns:
    float: The area of the circle
    """
    area = 3.14 * radius * radius
    return area
```

Here we have a clear docstring that explains the function's purpose, arguments, and return value.

- **Modular Design:** Break down your code into small, reusable modules or functions to make it easier to manage and test. This makes the code more manageable, testable, and reusable.

  Benefits of Modular Design:

  → **Improves Readability:** Smaller, well-defined modules are easier to understand than a large monolithic codebase.

- → **Facilitates Testing:** Each module can be tested independently, which helps in identifying and fixing bugs more efficiently.
- → **Enhances Reusability:** Modules can be reused across different projects or parts of the same project, reducing redundancy.
- → **Simplifies Maintenance:** Updates and modifications can be made to individual modules without affecting the entire system.

MATLAB

Example 1: Function for Data Normalization

```
function normalizedData = normalizeData(data)
    % normalizeData - Normalize the input data to [0, 1] range
    minVal = min(data);
    maxVal = max(data);
    normalizedData = (data - minVal) / (maxVal - minVal);
end
```

Usage in Main Script:

```
data = [5, 10, 15, 20];
normalizedData = normalizeData(data);
disp(normalizedData);
```

Example 2: Separate Function for Plotting

```
function plotData(data)
    % plotData - Plot the given data
    figure;
    plot(data, '-o');
    title('Data Plot');
    xlabel('Index');
    ylabel('Value');
end
```

Usage in Main Script:

```
data = [5, 10, 15, 20];
plotData(data);
```

Python

Example 1: Function for Data Normalization

```
def normalize_data(data):
    """Normalize the input data to [0, 1] range."""
    min_val = min(data)
    max_val = max(data)
    normalized_data = [(x - min_val) / (max_val - min_val) for x in data]
    return normalized_data
```

Usage in Main Script:

```
data = [5, 10, 15, 20]
normalized_data = normalize_data(data)
print(normalized_data)
```

Example 2: Separate Function for Plotting

```
import matplotlib.pyplot as plt

def plot_data(data):
    """Plot the given data."""
    plt.figure()
    plt.plot(data, '-o')
    plt.title('Data Plot')
    plt.xlabel('Index')
    plt.ylabel('Value')
    plt.show()
```

Usage in Main Script:

```
data = [5, 10, 15, 20]
plot_data(data)
```

R

Example 1: Function for Data Normalization

```
normalize_data <- function(data) {
    # Normalize the input data to [0, 1] range
    min_val <- min(data)
    max_val <- max(data)
    normalized_data <- (data - min_val) / (max_val - min_val)
    return(normalized_data)
}
```

Usage in Main Script:

```
data <- c(5, 10, 15, 20)
normalized_data <- normalize_data(data)
print(normalized_data)
```

Example 2: Separate Function for Plotting

```
plot_data <- function(data) {
    # Plot the given data
    plot(data, type = 'o', main = 'Data Plot', xlab = 'Index', ylab = 'Value')
}
```

Usage in Main Script:

```
data <- c(5, 10, 15, 20)
plot_data(data)
```

- **Regular Refactoring:** Regularly refactor your code to improve its structure and readability without changing its behavior.

✗ What Not to Do

```
def calculate_total_price(price, tax):
    total = price + (price * tax)
    return total
```

This function works, but let's assume it gets cluttered with more logic over time.

✓ What to Do

```
def calculate_total_price(price, tax):
    """
    Calculate the total price including tax.

    Args:
    price (float): The initial price
    tax (float): The tax rate

    Returns:
    float: The total price including tax
    """
    return price + (price * tax)
```

```python
def apply_discount(total, discount):
    """
    Apply discount to the total price.

    Args:
    total (float): The total price before discount
    discount (float): The discount rate

    Returns:
    float: The total price after discount
    """
    return total - (total * discount)
```

Regular refactoring can help split complex functions into smaller, more manageable ones, improving the code's structure.

- **Writing Tests:** Develop unit tests, integration tests, and other forms of testing to ensure that your code works as expected and to catch issues early.

✗ What Not to Do

```python
def add(a, b):
    return a + b

print(add(2, 3))  # Manual testing
```

Relying on manual testing is error-prone and not scalable.

✓ What to Do

```python
import unittest

def add(a, b):
    return a + b

class TestAddFunction(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
        self.assertEqual(add(-1, 1), 0)
        self.assertEqual(add(0, 0), 0)

if __name__ == '__main__':
    unittest.main()
```

Using a unit testing framework to automate tests, ensuring that the function behaves as expected in different scenarios.

---

3 Importance of clean code

Writing clean code is crucial because it prevents many bugs from arising in the first place and makes the debugging process much smoother.

- **Prevents Bugs:** Clear and simple code reduces the likelihood of introducing bugs.
- **Facilitates Maintenance:** Clean code is easier to update and extend, making future maintenance less error-prone.
- **Enhances Collaboration:** When multiple developers work on the same codebase, clean code ensures that everyone can understand and contribute effectively.

- **Improves Efficiency:** ell-organized and understandable code speeds up development and debugging processes.
- **Encourages Best Practices:** Striving for clean code encourages the adoption of best practices, leading to overall better software quality.

---

4 What is Debugging?

Definition

- Debugging is the process of identifying and resolving errors, bugs, or unexpected behavior in software code.
- It involves systematically analyzing code to locate the source of the problem and implementing solutions to fix it.

Importance

- Debugging is an essential skill for software developers, enabling them to ensure the correctness, reliability, and efficiency of their programs.

Techniques

- Debugging techniques vary depending on the programming language, development environment, and nature of the problem.

---

5 Why is it called “debugging”?

Origin

- The term ”debugging” originated in the early days of computing, particularly during the development of early electronic computers and mechanical computing devices.
- Grace Hopper, an American computer scientist and Navy rear admiral, is often credited with coining the term.

Historical Anecdote

- In 1947, while working on the Harvard Mark II computer, Hopper and her team found a moth trapped in one of the relays, causing a malfunction.
- They removed the moth and noted the event in the logbook, using the term ”debugging” to describe the process of identifying and fixing the problem.

Terminology

- The term ”bug” had already been used in engineering to describe a technical glitch or defect.
- The incident with the moth is often cited as the origin of ”debugging” in the context of computer programming.

---

6 Why is Debugging Useful?

Ensures Code Reliability

- Debugging helps identify and fix errors, ensuring that software functions as intended and produces accurate results.

Saves Time and Resources

- By pinpointing and resolving issues promptly, debugging minimizes downtime and prevents costly delays in software development cycles.

Enhances Code Quality

- Debugging promotes cleaner, more efficient code by uncovering and addressing logic flaws, syntax errors, and other issues that can compromise performance and maintainability.

Facilitates Learning

- Through the process of debugging, developers gain insights into how code behaves and learn valuable problem-solving skills that can be applied to future projects.

Builds Confidence

- Successfully debugging code instills confidence in developers, validating their skills and empowering them to tackle more complex challenges with assurance.

---

7 Types of errors

- **Syntax Errors:** These occur when the code violates the rules of the programming language. They are usually detected by the compiler or interpreter during the code compilation or execution phase. Examples include missing parentheses, invalid variable names, or missing semicolons.

```
# Syntax error due to missing closing parenthesis
print("Hello, World!"
```

```
# Syntax error due to invalid variable name
my-variable = 10
```

- **Logical Errors:** Logical errors occur when there is a flaw in the algorithm or logic of the code, causing it to produce incorrect results. Unlike syntax errors, logical errors do not result in immediate error messages but instead lead to unexpected behavior or incorrect output. They are often more challenging to detect and debug.

```
# Incorrect calculation of average
def calculate_average(numbers):
    total = sum(numbers)
    # Incorrect calculation: dividing by total count instead of number of elements
    average = total / len(numbers)
    return average

# Example usage
numbers = [10, 20, 30, 40, 50]
print("Average:", calculate_average(numbers))  # Output: Average: 30.0 (incorrect)
```

```
# Incorrect logic in conditional statement
def is_positive(number):
    if number > 0:
        return True
    else:
        return False

# Example usage
print(is_positive(0))  # Output: False (incorrect)
```

- **Runtime Errors:** Runtime errors occur during the execution of the program and typically result from issues such as invalid input, division by zero, or accessing elements outside the bounds of an array. These errors can cause the program to crash or terminate abruptly if not handled properly.

```
# Runtime error due to division by zero
result = 10 / 0
```

```
# Runtime error due to accessing an index out of range
numbers = [1, 2, 3]
print(numbers[3])
```

- **Semantic Errors:** Semantic errors occur when the code runs without generating any error messages but does not produce the intended outcome due to incorrect logic or misunderstanding of the problem requirements. These errors are subtle and may go unnoticed until they manifest as unexpected behavior in the program.

```
# Incorrect interpretation of sorting requirements
def sort_numbers(numbers):
    # Incorrect assumption: sorting in descending order
    sorted_numbers = sorted(numbers, reverse=True)
    return sorted_numbers

# Example usage
numbers = [3, 1, 5, 2, 4]
print("Sorted Numbers:", sort_numbers(numbers))
# Output: Sorted Numbers: [5, 4, 3, 2, 1] (incorrect)
```

```
# Incorrect assumption about input format
def process_data(data):
    # Incorrect assumption: expecting data in a specific format
    processed_data = data * 2
    return processed_data

# Example usage
data = "10"
print("Processed Data:", process_data(data))
# Output: Processed Data: 1010 (incorrect)
```

- **Concurrency Errors:** Concurrency errors occur in multi-threaded or parallel programs when multiple threads access shared resources concurrently, leading to issues such as race conditions, deadlocks, or data corruption. These errors are often difficult to reproduce and debug due to their non-deterministic nature.

```
import threading

counter = 0

def increment():
    global counter
    for _ in range(1000000):
        counter += 1

thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Counter:", counter)  # Output may vary due to race condition
```

```python
import threading

lock1 = threading.Lock()
lock2 = threading.Lock()

def process1():
    lock1.acquire()
    lock2.acquire()
    # Process 1 logic
    lock1.release()
    lock2.release()

def process2():
    lock2.acquire()
    lock1.acquire()
    # Process 2 logic
    lock2.release()
    lock1.release()

thread1 = threading.Thread(target=process1)
thread2 = threading.Thread(target=process2)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

- **Integration Errors:** Integration errors arise when combining different modules or components of a system, leading to inconsistencies, compatibility issues, or unexpected interactions between subsystems. These errors can occur during the integration phase of software development and may require thorough testing to identify and resolve.

```python
# Integration error due to incompatible library versions
import pandas as pd
import numpy as np

data = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6]]))
```

```python
# Integration error due to mismatched data formats
import json

data = '{"name": "John", "age": 30}'
processed_data = json.loads(data)
```

- **Environment Errors:** Environment errors occur due to differences in the execution environment, such as operating system dependencies, hardware configurations, or external dependencies. These errors can lead to platform-specific issues that manifest when running the code in different environments.

```python
import os

# Environment error due to operating system dependency
file_path = "/path/to/nonexistent/file.txt"
if os.path.exists(file_path):
    with open(file_path, "r") as file:
        content = file.read()
else:
    print("File does not exist:", file_path)
```

```python
import requests

# Environment error due to external dependency issues
response = requests.get("https://api.example.com/data")
if response.status_code == 200:
    data = response.json()
    # Process data
else:
    print("Failed to retrieve data:", response.status_code)
```

8 Debugging Techniques (Detailed Overview)

Techniques

- **Print Statements**: Inserting print statements in code to output variable values and program state at specific points.
- **Logging**: Using logging libraries to record program execution details, errors, and warnings to a file or console.
- **Interactive Debuggers**: Utilizing tools like gdb, pdb, or IDE-integrated debuggers to step through code, inspect variables, and set breakpoints.
- **Automated Testing**: Writing and running test cases to automatically check for expected behavior and identify failing scenarios.
- **Code Review**: Collaborating with peers to review code for potential issues and gain different perspectives on problem-solving.
- **Static Analysis Tools**: Employing tools that analyze code for common errors, code smells, and potential bugs without executing the program.
- **Rubber Duck Debugging**: Explaining code and logic to an inanimate object (or another person) to gain a better understanding and spot errors.

Best Practices

- **Stay Organized**: Keep code clean and well-documented to make debugging easier.
- **Understand the Problem**: Take time to understand the bug before diving into fixing it.
- **Isolate the Issue**: Narrow down the code section where the problem occurs to avoid unnecessary debugging.
- **Test Solutions**: Verify that the implemented fixes resolve the issue without introducing new bugs.
- **Reflect and Learn**: Analyze what caused the bug and how to prevent similar issues in the future.