Software and Code Quality

Week 6

Simple is better than complex.

— Tim Peters

Objectives

- Understand the software and code quality
- Understand the code style and why it matters

Contents

- Software and code quality
- Clean code and code smells
- Coding style

Software Quality

- The degree to which software meets specified requirements and user needs, and is free of defects

- Attributes of software quality
  - Functionality: Does the software do what it is supposed to?
  - Reliability: How often does the software fail?
  - Usability: How user-friendly is the software?
  - Efficiency: Does the software make optimal use of resources?
  - Maintainability: How easy is it to change the software?
  - Portability: Can the software run on different platforms?

Source: Jung, Ho-Won, Seung-Gweon Kim, and Chang-Shin Chung. "Measuring software product quality: A survey of ISO/IEC 9126." IEEE software 21.5 (2004): 88-92.

Why Quality Important?

- User satisfaction
  - High-quality software meets or exceeds user expectations
- Competitive advantage
  - High-quality software differentiates a product in the market
- Cost efficiency
  - Defects are costly to fix, especially if detected late in development
- Reduced risks
  - Ensuring software quality reduces the risk of software failures that can lead to financial loss or harm

How to Achieve High Software Quality?

- Version control
  - Manage changes to codebase for collaboration and traceability
  - Covered in Week 4
- Continuous testing
  - Regularly test the software to catch defects early
  - To be covered in Week 7-8
- Apply good design patterns
  - E.g., SOLID design principles[1] for object oriented programming
  - To be covered in Week 10-11

1. http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Code Quality vs. Software Quality

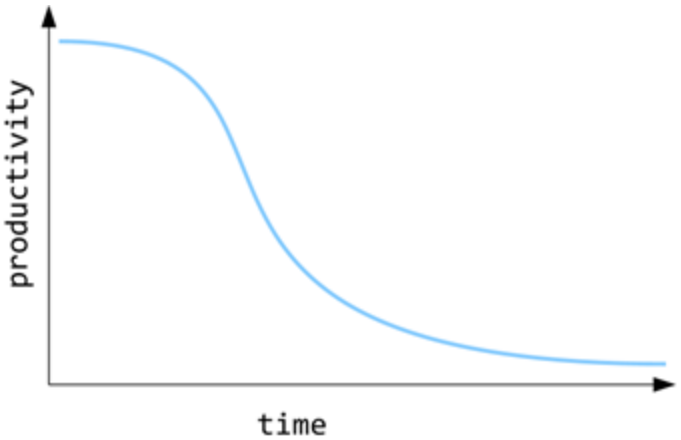
- Code quality
  - The internal representation
  - The maintainability, readability, and scalability of the code
- Software quality
  - The external representation
  - How well software meets user needs and is free of defects
- Code quality determines the long-term robustness, flexibility, and reliability of software

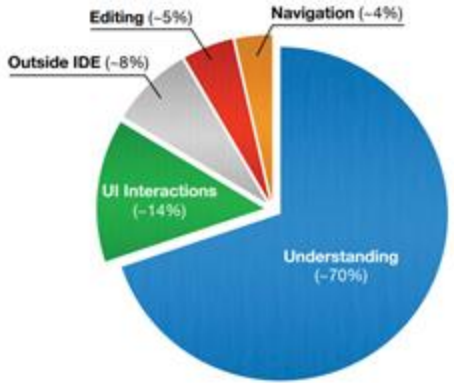
Why Code Quality Matters?

- Foundation
  - Software quality relies on the quality of its code just as a building's durability depends on the quality of its foundation
- Future proofing
  - High code quality ensures that the software can evolve, adapt, and scale with changing needs
- Cost efficiency
  - Poor code quality leads to *technical debt* – issues that will be more expensive to fix later than addressing them now
  - This impacts the long-term maintenance and reliability of software

Clean Code

- Studies have shown that developers spend approximately 50-80% of their time understanding existing code[1,2]
- Writing clean code is essential for productivity





How do developers spend their time?

1. Lammers, Susan M. Programmers at work: Interviews with 19 programmers who shaped the computer industry. Microsoft Press, 1989.
2. Minelli, Roberto, Andrea Mocci, and Michele Lanza. "I know what you did last summer-an investigation of how developers spend their time." 2015 IEEE 23rd International conference on program comprehension. IEEE, 2015.

Clean Code

- Features of clean code
  - Readability: Readable code is easy to maintain and debug
  - Simplicity: Simple code has fewer bugs and is easier to modify
  - Self-documenting: Code should explain its purpose
- Other factors
  - Consistency: Following consistent coding standards and practices
  - Testability: Designing code that can be easily tested ensures it performs as expected
  - Modularity: Decomposing complex systems into simpler modules promotes reusability and easier maintenance

Source: Martin, Robert C. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009.

Clean Code Principles

- Use clear names
- Use one word for one concept
- Make functions smaller
- Minimize side effects
- Keep your code DRY
- Use clear comment
- And many more …

Source: Martin, Robert C. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009.

Use Clear Names

- The name of a class, variable, or method should be clear to describe its purpose
- We should avoid using meaningless names
- Do not abbreviate a variable name too much

Bad Names vs. Good Names

Bad names

```
var d: Int // elapsed time in days
var ds: Int
var dsm: Int
var faid: Int
```

Good names

```
var elapsedTimeInDays: Int
var daysSinceCreation: Int
var daysSinceModification: Int
var fileAgeInDays: Int
```

MineSweeper Example

- Assume that you are making a minesweeper game
- Do you understand the code below at a glance?

```
class MineSweeper {
    private val theArray = Array(10) { IntArray(10) } // 10x10 array

    fun changeArray(i1: Int, i2: Int) {
        theArray[i1][i2] = 1
    }
}
```

MineSweeper Example

- This is a much readable code

```
class MineSweeper {
    private val UNCOVERED = 0
    private val COVERED = 1
    private val FLAGGED = 2

    private val cellClickStates = Array(10) { IntArray(10) } // 10x10 array

    fun flagCell(x: Int, y: Int) {
        // Sets the cell's state to FLAGGED.
        cellClickStates[x][y] = FLAGGED
    }
}
```

Bad Name: More Examples

- Avoid disinformation

```
val customerArray: Array<Customer>
val theTable: Table
```

- Don't use too long/short name length

```
val theCustomersListWithAllCustomersIncludedWithoutFilter: ArrayList<Customer>
val list: ArrayList<Customer>
```

More Bad Naming Examples

- Follow naming convention

```
const val maxcount = 1 // ⇒ maxCount
var change = true // ⇒ isChanged
private var Name: String // ⇒ name
class personaddress // ⇒ PersonAddress
fun getallorders(): Array<Order> // ⇒ getAllOrders
```

- Use names in self contexts

```
var addressCity: String // ⇒ city
var addressHomeNumber: String // ⇒ homeNumber
var addressPostCode: String // ⇒ postCode
```

Use One Word for One Concept

- Using multiple words for one concept makes your code confusing and hard-to-use
- For instance, it is not good to mix “get”, “fetch”, and “retrieve” to write getter methods

```
class Student {
  fun getStudentId(): String { ... }
  fun fetchAge(): Int { ... }                  // => getAge
  fun retrieveDepartment(): String { ... } // => getDepartment
}
```

Make Functions Smaller

- Split big functions into smaller functions
  - A function should do only one thing
  - A function should abstract only one level
- You will feel tempted to write a long code in one function for convenience. This maybe okay for the time being, but think the future use (especially by other programmers)
- If your function becomes longer than 15 lines, think if you can break it into two

Big Function

```
fun withdrawUI(id: Int, password: String, amount: Int) {
    for (i in 0 until numAccounts) {
        val account = accounts[i]
        if (account.getId() == id) {
            if (account.password.equals(password)) {
                balance -= amount
                println("Withdraw success!")
            } else {
                println("Authentication fail")
            }
        } else {
            println("No such account")
        }
    }
}
```

Split into Small Functions

```
fun findAccount(id: Int): BankAccount? {
    for (i in 0 until numAccounts) {
        val account = accounts[i]
        if (account.getId() == id) {return account}
    }
    return null
}
fun withdrawUI(id: Int, password: String, amount: Int) {
    val account = findAccount(id)
    if (account == null) {
        println("No such account")
    } else if (account.password == password) {
        balance -= amount
        println("Withdraw success!")
    } else {
        println("Authentication fail")
    }
}
```

Minimize Side Effects

- If a function modifies a variable value outside its scope, the function called to have a "side effect"
- You need to be careful when you change the value of a variable that may be used by other functions

Side Effect Example

```
class School {
    private var totalStudent = 0
    val studentList = arrayListOf<Student>()

    fun getNewID(): Int {
        return ++totalStudent
    }

    fun registerStudent(name: String) {
        val newStudent = Student(name, getNewID())
        studentList.add(newStudent)
        ++totalStudent
    }
}
```

Keep Your Code DRY!

- Don’t Repeat Yourself (DRY)
- Repeated code makes you hard to modify your code
- If you miss one of the repeated parts, bugs can appear

Repeated Code

```
class Printer {
    companion object {
        fun printInt(i: Int) {println("Type: Integer, Value: $i)}

        fun printString(str: String) {println("Type: String, Value: $str")}

        fun printDouble(d: Double) {println("Type: Double, Value: $d")}
    }
}

fun main() {
    Printer.printInt(2)
    Printer.printString("Hello World!")
    Printer.printDouble(1.23)
}
```

Non-Repeated Code

```
object Main {
    fun printVariable(obj: Any) {
        println("Type: ${obj::class.simpleName}, Value: $obj")
    }
}

fun main() {
    Main.printVariable(2)
    Main.printVariable("Hello World!")
    Main.printVariable(1.23)
}
```

Repeated vs Non-repeated Code

- In the previous examples,
  - What if you want to print 100 types of other variables?
  - If you want to change the printing message format, for example “This is a <type> variable <value>”?

Use Clear Comments

- Before writing comments, think if you can make your code self-explanatory
- It is usually not a good idea to explain what your code does in detail
  - Needs for detailed comments indicate that your code may not be intuitive and clean
- In comments, explain your high-level intention or other information that can’t be easily captured by reading code

Use Clear Comments

Bad Comment

```
/*
 * Find an account with findAccount,
 * and then return true if account is not null,
 * and the account is authorized with the password
 */
```

Good Comment

```
/*
 * BankAccount authorization api for external libraries
 */
fun authorize(accountId: Int, password: String): Boolean {
    val account = findAccount(accountId)
    return account != null && account.authorize(password)
}
```

More Complicated Example

- Why Comments Are Stupid, a Real Example

https://simpleprogrammer.com/why-comments-are-stupid-a-real-example/

Code Small Bits and Test

- Test a small part of your code before you write too much
- This may sound annoying, but it’s much better than testing after you write a hundred lines of buggy code when you don’t know where the bug comes from

Other Useful Guides

- Consistent indentation
  - Properly indent your code to enhance readability
- Use exceptions, not return codes
  - Using exceptions for error handling makes the code cleaner and less cluttered with error-handling logic
- Define and control boundaries
  - Know where to draw the line between different parts of the system
  - Use interfaces and encapsulation to separate concerns
- Keep configurations external
  - Avoid hardcoding configuration values; instead, externalize them
- Many more…

Code Smells: Indicator of not being ‘Clean’

- Code smells are indicators of potential problem in the code, even if the code works
- Code smells are not necessarily bugs, but they indicate weaknesses in design, which might affect performance, maintenance, or scalability
- While clean code embodies best practices, code smells signal departures from these practices

Source: Fowler, Martin. Refactoring. Addison-Wesley Professional, 2018.

Common Code Smells

- Duplicated code
  - The same code structure in multiple places
- Long method
  - A method that tries to do too much, making it hard to understand
- Large class
  - A class that has taken on too many responsibilities
- Feature envy
  - A method more interested in a class other than the one it is in
- Data clumps
  - The same group of variables is passed around in multiple places

What to do when Code Smells? Refactoring!

- Refactoring is the process of restructuring code to improve its internal structure without changing its external behavior
- Refactoring converts the just working code to the well-managed clean code
- More details will be covered in Week 13

Source: Fowler, Martin. Refactoring. Addison-Wesley Professional, 2018.

Coding Style

Coding Style

- A set of guidelines and conventions that developers in a team follow to write clean code
- Following the good coding style is the first step to write clean code
- It ensures that code is consistent, readable, and maintainable across the entire project or organization
- While different styles might be equally valid, consistency is key to prevent confusion and errors

Coding Style Examples

- Python
  - PEP8: the official style guide
- JavaScript
  - Airbnb's JavaScript style
- For Android development,
  - Google's Kotlin Style Guide (https://developer.android.com/kotlin/style-guide)
  - Google's Java Style Guide (https://google.github.io/styleguide/javaguide.html)

Source: https://google.github.io/styleguide/

Adopting and Enforcing a Coding Style

- Choose or define
  - Adopt an existing style guide or create one for your project
- Documentation
  - Clearly document and share the style guide with your team
- Tooling
  - Use linters (e.g., ESLint, Pylint) to check and enforce coding style
- Code reviews
  - Regularly review code to ensure adherence to the style guide

Kotlin Style Guide

Source File Basics

- All source files must be encoded as UTF-8
- If a source file contains only one top-level class
  - Name the file the same as the class name with .kt extension
- If a source file contains multiple top-level declarations
  - Use a file name that describes the contents with .kt extension
  - Use PascalCase, or camelCase if the name is plural

```
// MyClass.kt
class MyClass { }
```

```
// extensions.kt
fun MyClass.process() = // …
fun MyResult.print() = // …
```

Source: https://developer.android.com/kotlin/style-guide#source_files

Source File Structure

- A .kt file includes items below:
  - Copyright/license header
  - Package statement
  - Import statements
  - Top-level declarations
  - One blank line separating each section

```kotlin
// Copyright or licence info

package org.example.project

import kotlin.math.*
import java.util.*

class MyClass {
    // class-level members.
    companion object {
        const val MY_CONSTANT = 10
    }
    // Instance field
    private var myField: Int = 0
    fun myMethod() {
        // method body
    }
}
```

Source: https://developer.android.com/kotlin/style-guide#structure

Naming Rules

- Package: Use lowercase letters. Do not use underscore.
- Class and object: Use UpperCamelCase
- Function: Start with a lowercase letter and use camel case
- Constant: Use all uppercase, underscore-separated names

```
package org.example.project


object EmptyDeclarationProcessor :
DeclarationProcessor() { /*...*/ }
```

```
fun processDeclarations() {
/*...*/ }


const val MAX_COUNT = 8
```

Source: https://kotlinlang.org/docs/coding-conventions.html#naming-rules

Indentation

- Use four spaces for indentation → Do not use tabs(\t)
- For curly braces {}
  - Opening brace at the end of the declaration
  - Closing brace on a new line aligned with the declaration

```
if (elements != null) {
    for (element in elements) {
        // ...
    }
}
```

Source: https://kotlinlang.org/docs/coding-conventions.html#indentation

Horizontal Whitespace

- Add spaces around binary operators (`a + b`)
- No spaces around unary operators (`a++`)
- Add a space after control keywords (`if`,`when`,`for`, and `while`)
- No space before parentheses in declarations or calls

```
class A(val x: Int)

fun foo(x: Int) { ... }

fun bar() {
    foo(1)
}
```

Source: https://kotlinlang.org/docs/coding-conventions.html#horizontal-whitespace

Modifiers Order

- Put modifiers in the following order

```
public / protected / private / internal
expect / actual
final / open / abstract / sealed / const
external
override
lateinit
tailrec
vararg
suspend
inner
enum / annotation / fun // as a modifier
in `fun interface`
companion
inline / value
infix
operator
data
```

Source: https://kotlinlang.org/docs/coding-conventions.html#modifiers-order

Functions

- For multi-line function signatures

```
fun longMethodName(
    argument: ArgumentType = defaultValue,
    argument2: AnotherArgumentType,
): ReturnType {
    // body
}
```

- Use an expression body (=) for single-expression functions

```
fun foo(): Int {     // bad
    return 1
}

fun foo() = 1        // good
```

Source: https://kotlinlang.org/docs/coding-conventions.html#functions

Comments

- For long comments, place /** on a new line and * on the following lines
- Avoid using @param and @return tags. Wrtie descriptions in the text

```
// Avoid doing this:
/**
 * Returns the absolute value of the given number.
 * @param number The number to return the absolute value for.
 * @return The absolute value.
 */
fun abs(number: Int): Int { /*...*/ }

// Do this instead:
/**
 * Returns the absolute value of the given [number].
 */
fun abs(number: Int): Int { /*...*/ }
```

Source: https://kotlinlang.org/docs/coding-conventions.html#documentation-comments

Additional Resources

- "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
- "Refactoring: Improving the Design of Existing Code" by Martin Fowler
- "Software Engineering at Google" by Titus Winters, Tom Manshreck
- "Seriously Good Software: Code that works, survives, and wins" by Marco Faella

Thank You.
Any Questions?