Capstone Project 2

CMU-SE 451

Code Standard Document

Version 2.0

Date: 01/03/2023

Craft Village Pollution Monitor System

Submitted by

Ca, Van Cong Le

Huy, Bui Duc

Phuc, Hua Hoang

Trung, Nguyen Thanh

Nhan, Huynh Ba

Approved by

Ph.D. Nguyen Thanh Binh

Proposal Review Panel Representative:

Name Signature Date

Capstone Project 2 - Mentor:

Name Signature Date

PROJECT INFORMATION

Project acronym	CVPMS		
Project Title	Craft Village Pollution Monitor System		
Start Date	01/03/2023	**End Date**	15/05/2023
Lead Institution	International School, Duy Tan University		
Project Mentor	Ph.D. Thanh Binh, Nguyen		
Scrum master / Project Leader & contact details	Ca, Van Cong Le *Email:* cascabusiness@gmail.com *Tel:* 0352707895		
Partner Organization			
Project Web URL	*https://github.com/Casca113s2/craft-village-pollution-monitor-system*		
Team members	**Name**	**Email**	**Tel**
25211207666	Ca, Van Cong Le	cascabusiness@gmail.com	0352707895
25211215894	Huy, Bui Duc	duchuyltt122@gmail.com	0818648090
25211204084	Phuc, Hua Hoang	phuchuho0402@gmail.com	0905639682
25211215133	Trung, Nguyen Thanh	nguyenttrung2601@gmail.com	0774496838
25211203702	Nhan, Huynh Ba	huynhbanhan1491999@gmail.com	0935430785

DOCUMENT NAME

Document Title	Code Standard Document		
Author(s)	Van Cong Le Ca Hua Hoang Phuc Huynh Ba Nhan		
Date	01/03/2023	**File Name**	C2SE.01_CVPMS_Code-Standard-Document_v2.0.docx

REVISION HISTORY

Version	Date	Comments	Author	Approval
2.0	01/03/2023	Initial Release	All members	

Approve Document: Sign in to approve the document

Mentor	Binh, Nguyen Thanh	Date	01/03/2023
		Sign	
Scrum Master	Ca, Van Cong Le	Date	01/03/2023
		Sign	
Scrum Member	Huy, Bui Duc	Date	01/03/2023
		Sign	
Scrum Member	Phuc, Hua Hoang	Date	01/03/2023
		Sign	
Scrum Member	Trung, Nguyen Thanh	Date	01/03/2023
		Sign	
Scrum Member	Nhan, Huynh Ba	Date	01/03/2023
		Sign	

Table Of Contents

1. Introduction

1.1. Purpose

This Coding Standard requires certain practices for developing programs in the Java, Dart language. The objective of this coding standard is to have a positive effect on:

- Avoidance of errors/bugs, especially the hard-to-find ones.
- Maintainability, by promoting some proven design principles

1.2. Scope

This standard pertains to the use of the Java, Dart language.

2. Code Standards

2.1. Dart Language Code Standard

2.1.1. Identifiers

- DO name types using UpperCamelCase.
- DO name extensions using UpperCamelCase.
- DO name libraries, packages, directories, and source files using lowercase_with_underscores.
- DO name import prefixes using lowercase_with_underscores.
- DO name other identifiers using lowerCamelCase.
- PREFER using lowerCamelCase for constant names.
- DO capitalize acronyms and abbreviations longer than two letters like words.
- PREFER using _, __, etc. for unused callback parameters.
- DON’T use a leading underscore for identifiers that aren’t private.
- DON’T use prefix letters.

2.1.2. Ordering

- DO place “dart:” imports before other imports.
- DO place “package:” imports before relative imports.
- DO specify exports in a separate section after all imports.
- DO sort sections alphabetically.

2.1.3. Formatting

- DO format your code using dart format.
- CONSIDER changing your code to make it more formatter-friendly.

- AVOID lines longer than 80 characters.
- DO use curly braces for all flow control statements.

2.1.4. Comments

- DO format comments like sentences.
- DON’T use block comments for documentation.

2.1.5. Doc comments

- DO use /// doc comments to document members and types.
- PREFER writing doc comments for public APIs.
- CONSIDER writing a library-level doc comment.
- CONSIDER writing doc comments for private APIs.
- DO start doc comments with a single-sentence summary.
- DO separate the first sentence of a doc comment into its own paragraph.
- AVOID redundancy with the surrounding context.
- PREFER starting function or method comments with third-person verbs.
- PREFER starting a non-boolean variable or property comment with a noun phrase.
- PREFER starting a boolean variable or property comment with “Whether” followed by a noun or gerund phrase.
- DON’T write documentation for both the getter and setter of a property.
- PREFER starting library or type comments with noun phrases.
- CONSIDER including code samples in doc comments.
- DO use square brackets in doc comments to refer to in-scope identifiers.
- DO use prose to explain parameters, return values, and exceptions.
- DO put doc comments before metadata annotations.

2.1.6. Markdown

- AVOID using markdown excessively.
- AVOID using HTML for formatting.
- PREFER backtick fences for code blocks.

2.1.7. Writing

- PREFER brevity.
- AVOID abbreviations and acronyms unless they are obvious.
- PREFER using “this” instead of “the” to refer to a member’s instance.

2.1.8. Libraries

- DO use strings in part of directives.
- DON’T import libraries that are inside the src directory of another package.
- DON’T allow an import path to reach into or out of lib.
- PREFER relative import paths.

2.1.9. Null

- DON’T explicitly initialize variables to null.
- DON’T use an explicit default value of null.
- PREFER using ?? to convert null to a boolean value.
- AVOID late variables if you need to check whether they are initialized.
- CONSIDER assigning a nullable field to a local variable to enable type promotion.

2.1.10. Strings

- DO use adjacent strings to concatenate string literals.
- PREFER using interpolation to compose strings and values.
- AVOID using curly braces in interpolation when not needed.
- Collections
- DO use collection literals when possible.
- DON’T use .length to see if a collection is empty.
- AVOID using Iterable.forEach() with a function literal.
- DON’T use List.from() unless you intend to change the type of the result.
- DO use whereType() to filter a collection by type.
- DON’T use cast() when a nearby operation will do.
- AVOID using cast().

2.1.11. Functions

- DO use a function declaration to bind a function to a name.
- DON’T create a lambda when a tear-off will do.
- DO use = to separate a named parameter from its default value.
- Variables
- DO follow a consistent rule for var and final on local variables.
- AVOID storing what you can calculate.

2.1.12. Members

- DON’T wrap a field in a getter and setter unnecessarily.
- PREFER using a final field to make a read-only property.
- CONSIDER using => for simple members.
- DON’T use this. except to redirect to a named constructor or to avoid shadowing.
- DO initialize fields at their declaration when possible.

2.1.13. Constructors

- DO use initializing formals when possible.
- DON’T use late when a constructor initializer list will do.
- DO use ; instead of {} for empty constructor bodies.
- DON’T use new.
- DON’T use const redundantly.

2.1.14. Error handling

- AVOID catches without on clauses.
- DON’T discard errors from catches without on clauses.
- DO throw objects that implement Error only for programmatic errors.
- DON’T explicitly catch Error or types that implement it.
- DO use rethrow to rethrow a caught exception.

2.1.15. Asynchrony

- PREFER async/await over using raw futures.
- DON’T use async when it has no useful effect.
- CONSIDER using higher-order methods to transform a stream.
- AVOID using Completer directly.
- DO test for Future<T> when disambiguating a FutureOr<T> whose type argument could be Object.

2.1.16. Names

- DO use terms consistently.
- AVOID abbreviations.
- PREFER putting the most descriptive noun last.
- CONSIDER making the code read like a sentence.
- PREFER a noun phrase for a non-boolean property or variable.

- PREFER a non-imperative verb phrase for a boolean property or variable.
- CONSIDER omitting the verb for a named boolean parameter.
- PREFER the “positive” name for a boolean property or variable.
- PREFER an imperative verb phrase for a function or method whose main purpose is a side effect.
- PREFER a noun phrase or non-imperative verb phrase for a function or method if returning a value is its primary purpose.
- CONSIDER an imperative verb phrase for a function or method if you want to draw attention to the work it performs.
- AVOID starting a method name with get.
- PREFER naming a method to___() if it copies the object’s state to a new object.
- PREFER naming a method as___() if it returns a different representation backed by the original object.
- AVOID describing the parameters in the function’s or method’s name.
- DO follow existing mnemonic conventions when naming type parameters.

2.1.17. Libraries

- PREFER making declarations private.
- CONSIDER declaring multiple classes in the same library.

2.1.18. Classes and mixins

- AVOID defining a one-member abstract class when a simple function will do.
- AVOID defining a class that contains only static members.
- AVOID extending a class that isn’t intended to be subclassed.
- DO document if your class supports being extended.
- AVOID implementing a class that isn’t intended to be an interface.
- DO document if your class supports being used as an interface.
- DO use mixin to define a mixin type.
- AVOID mixing in a type that isn’t intended to be a mixin.

2.1.19. Constructors

- CONSIDER making your constructor const if the class supports it.
- Members
- PREFER making fields and top-level variables final.

- DO use getters for operations that conceptually access properties.
- DO use setters for operations that conceptually change properties.
- DON’T define a setter without a corresponding getter.
- AVOID using runtime type tests to fake overloading.
- AVOID public late final fields without initializers.
- AVOID returning nullable Future, Stream, and collection types.
- AVOID returning this from methods just to enable a fluent interface.

2.1.20. Types

- DO type annotate variables without initializers.
- DO type annotate fields and top-level variables if the type isn’t obvious.
- DON’T redundantly type annotate initialized local variables.
- DO annotate return types on function declarations.
- DO annotate parameter types on function declarations.
- DON’T annotate inferred parameter types on function expressions.
- DON’T type annotate initializing formals.
- DO write type arguments on generic invocations that aren’t inferred.
- DON’T write type arguments on generic invocations that are inferred.
- AVOID writing incomplete generic types.
- DO annotate with dynamic instead of letting inference fail.
- PREFER signatures in function type annotations.
- DON’T specify a return type for a setter.
- DON’T use the legacy typedef syntax.
- PREFER inline function types over typedefs.
- PREFER using function type syntax for parameters.
- AVOID using dynamic unless you want to disable static checking.
- DO use Future<void> as the return type of asynchronous members that do not produce values.
- AVOID using FutureOr<T> as a return type.

2.1.21. Parameters

- AVOID positional boolean parameters.
- AVOID optional positional parameters if the user may want to omit earlier parameters.

- AVOID mandatory parameters that accept a special “no argument” value.
- DO use inclusive start and exclusive end parameters to accept a range.
- Equality
- DO override hashCode if you override ==.
- DO make your == operator obey the mathematical rules of equality.
- AVOID defining custom equality for mutable classes.
- DON’T make the parameter to == nullable.

2.2. Java Language Code Standard

2.2.1. Source file structure

- License or copyright information, if present: If license or copyright information belongs in a file, it belongs here.
- Package statement: The package statement is not line-wrapped. The column limit does not apply to package statements.
- Import statements
  - No wildcard imports
  - No line-wrapping
  - Ordering and spacing
  - No static import for classes
- Class declaration
  - Exactly one top-level class declaration
  - Ordering of class contents

2.2.2. Formatting

- Braces
  - Use of optional braces
  - Nonempty blocks: K & R style
  - Empty blocks: may be concise
- Block indentation: +2 spaces
- One statement per line
- Column limit: 100
- Line-wrapping
  - Prefer to break at a higher syntactic level
  - Indent continuation lines at least +4 spaces

- Whitespace
  - Vertical Whitespace
  - Horizontal whitespace
  - Horizontal alignment: never required
- Grouping parentheses: recommended
- Specific constructs
  - Enum classes
  - Variable declarations
    - One variable per declaration
    - Declared when needed
  - Arrays
    - Array initializers: can be "block-like"
    - No C-style array declarations
  - Switch statements
    - Indentation
    - Fall-through: commented
    - Presence of the *default* label
  - Annotations
    - Type-use annotations
    - Class annotations
    - Method and constructor annotations
    - Field annotations
    - Parameter and local variable annotations
  - Comments
    - Block comment style: Block comments are indented at the same level as the surrounding code.
  - Modifiers: Class and member modifiers, when present, appear in the order recommended by the Java Language Specification
  - Numeric Literals: long-valued integer literals use an uppercase L suffix, never lowercase (to avoid confusion with the digit 1).

2.2.3. Naming

- Rules common to all identifiers: Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores.
- Rules by identifier type
  - Package names use only lowercase letters and digits (no underscores). Consecutive words are simply concatenated together.
  - Class names are written in UpperCamelCase. Class names are typically nouns or noun phrases
  - Method names are written in lowerCamelCase. Method names are typically verbs or verb phrases.
  - Constant names use UPPER_SNAKE_CASE: all uppercase letters, with each word separated from the next by a single underscore.
  - Non-constant field names (static or otherwise) are written in lowerCamelCase. These names are typically nouns or noun phrases.
  - Parameter names are written in lowerCamelCase. One-character parameter names in public methods should be avoided.
  - Local variable names are written in lowerCamelCase. Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.
  - Each type variable is named in one of two styles:
    - A single capital letter, optionally followed by a single numeral (such as E, T, X, T2)
    - A name in the form used for classes (see Section 5.2.2, Class names), followed by the capital letter T (examples: RequestT, FooBarT).
- Camel case: defined
  - Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
  - Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
    - Recommended: if any word already has a conventional camel-case appearance in common usage, split this into its constituent

parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel case per se; it defies any convention, so this recommendation does not apply.

- Now lowercase everything (including acronyms), then uppercase only the first character of:
  - ... each word, to yield upper camel case, or
  - ... each word except the first, to yield lower camel case
- Finally, join all the words into a single identifier.

2.2.4. Programming Practices

- @Override: always used
- Caught exceptions: not ignored
- Static members: qualified using class
- Finalizers: not used

3. References

1. *Google team*, "Effective Dart"
2. *Google team*, "Google Java Style Guide"