**LOUIE JAY A. CENIZA APRIL 2, 2024**

**SMELLS AND HEURISTICS (CHAPTER 17)**

**Comments.** Comments are usually the description of your variables or functions to understand it more andin this lesson, there are 5 sub lessons that are describing what you should avoid in code comments:

**Inappropriate Information:** Comments should focus on technical aspects of the code and design, rather than storing information better suited for other systems like source code control or issue tracking. Avoid cluttering source files with irrelevant metadata such as authors or modification dates.

**Obsolete Comment:** Comments become obsolete quickly. It's best to avoid writing comments that will soon be outdated. If you encounter obsolete comments, update or remove them promptly to prevent confusion and maintain code clarity.

**Redundant Comment:** A comment is considered redundant if it merely restates what is already evident in the code. Ensure comments provide additional insight that enhances understanding beyond what the code itself conveys.

**Poorly Written Comment:** Well-crafted comments are essential. Take the time to choose words carefully, use correct grammar and punctuation, and avoid unnecessary repetition or verbosity. Keep comments concise and focused on relevant technical details.

**Commented-out Code:** Commented-out code is discouraged as it quickly becomes outdated, cluttering the codebase and potentially causing confusion. Instead of leaving code commented out, it's better to delete it. Source code control systems retain previous versions if needed, making it unnecessary to keep commented-out code in the active codebase.

**Environment.** It refers to the set of tools, resources, configurations and conditions and in this lesson, it has a sub lesson which are:

**Build Requires More Than One Step:** Building a project should be a straightforward process, requiring minimal effort and complexity. Ideally, you should be able to check out the system from source code control with a single command and then build it with another simple command. Avoid the need for multiple commands, arcane scripts, or manual gathering of dependencies like JARs or XML files. A seamless build process enhances productivity and reduces errors.

**Tests Require More Than One Step:** Running all unit tests should be a simple, one-step operation. Whether it's clicking a button in your IDE or executing a single command in the shell, the process should be quick, easy, and intuitive. Testing is a critical aspect of software development, and ensuring that all tests can be run efficiently is essential for maintaining code quality and reliability.

**Functions.** Functions are essential building blocks in programming, but there are some common pitfalls to avoid when designing them:

**Too Many Arguments:** Functions should ideally have a small number of arguments, with zero being the best scenario. One or two arguments are generally acceptable, but having more than three can make the function difficult to use and understand. It's best to keep the number of arguments minimal to enhance readability and maintainability.

**Output Arguments:** Functions should primarily accept inputs and produce outputs based on those inputs. Using output arguments, where the function modifies the state of one or more of its arguments, can be confusing for readers. If a function needs to change the state of something, it's better to have it modify the state of the object it's called on rather than using output arguments.

**Flag Arguments:** Boolean arguments, also known as flag arguments, indicate that a function performs more than one action based on the value of the flag. This approach is confusing and should be avoided. Instead of using flag arguments, consider splitting the function into multiple smaller functions, each with a single responsibility.

**Dead Functions:** Functions that are never called are essentially dead code and should be removed from the codebase. Keeping unused functions around only adds unnecessary clutter and can confuse developers who come across them later. It's important to regularly clean up unused functions to maintain a clean and maintainable codebase.

**General.** addresses fundamental aspects of code quality, such as modularity, readability, simplicity, consistency, and maintainability, providing developers with a comprehensive framework for writing high-quality software, and here are the sub lessons:

**Multiple Languages in One Source File:** Avoid mixing multiple programming languages or dialects within a single source file, as it can lead to confusion and decreased readability.

**Obvious Behavior Is Unimplemented:** Ensure that expected behavior is implemented and not left unimplemented, as it can lead to unexpected outcomes and errors.

**Incorrect Behavior at the Boundaries:** Pay special attention to handling boundary cases (e.g., minimum and maximum values, edge cases) correctly to prevent bugs and unexpected behavior.

**Overridden Safeties:** Avoid overriding safety mechanisms or safeguards, as it can compromise the integrity and reliability of the code.

**Duplication:** Minimize code duplication to improve maintainability and reduce the risk of inconsistency and errors.

**Code at Wrong Level of Abstraction:** Ensure that code is written at the appropriate level of abstraction, avoiding unnecessary details or excessive abstraction that obscures the code's intent.

**Base Classes Depending on Their Derivatives:** Avoid situations where base classes depend on the behavior or implementation details of their subclasses, as it can lead to tight coupling and fragile designs.

**Too Much Information:** Avoid overwhelming the reader with excessive information or details in code, striving for clarity and simplicity instead.

**Dead Code:** Remove unused or dead code from the codebase to reduce clutter and confusion, and to maintain a clean and maintainable codebase.

**Vertical Separation:** Organize code vertically to group related functionality together, improving readability and understanding of the code. Inconsistency (G11): Strive for consistency in naming, formatting, and style throughout the codebase to enhance readability and maintainability.

**Clutter:** Remove unnecessary clutter, such as commented-out code or debug statements, to keep the codebase clean and focused.

**Artificial Coupling:** Avoid creating unnecessary dependencies between modules or components, as it can increase complexity and hinder flexibility.

**Feature Envy:** Be mindful of methods that excessively interact with or depend on the data of other classes, as it may indicate a design issue.

**Selector Arguments:** Avoid methods or functions that use selector arguments (e.g., Boolean flags) to determine behavior, as it can lead to complex and hard-to-understand code.

**Obscured Intent:** Write code that clearly communicates its intent, avoiding ambiguity or obscure constructs that make it difficult to understand.

**Misplaced Responsibility:** Ensure that each component or module is responsible for a single concern or functionality, avoiding situations where responsibilities are misplaced or unclear.

**Inappropriate Static:** Use static methods or variables judiciously, considering their impact on maintainability and testability, and favoring instance-based solutions when appropriate.

**Use Explanatory Variables:** Use descriptive and explanatory variable names to improve readability and understanding of the code.

**Function Names Should Say What They Do:** Choose function names that clearly convey their purpose or behavior, aiding comprehension and maintainability.

**Understand the Algorithm**: Ensure that algorithms are well understood and properly implemented to avoid errors and inefficiencies.

**Make Logical Dependencies Physical:** Explicitly declare dependencies between components or modules to make the codebase more manageable and understandable.

**Prefer Polymorphism to If/Else or Switch/Case:** Favor polymorphic solutions over conditional logic (e.g., if/else or switch/case statements) to improve flexibility and maintainability.

**Follow Standard Conventions:** Adhere to established coding conventions and standards to promote consistency and readability across the codebase.

**Replace Magic Numbers with Named Constants:** Replace magic numbers or hardcoded values with named constants to improve code clarity and maintainability.

**Be Precise:** Write code that is precise and unambiguous, avoiding vague or ambiguous language or constructs.

**Structure over Convention:** Prefer explicit structure and organization over relying on implicit conventions or assumptions.

**Encapsulate Conditionals:** Encapsulate conditional logic within well-named methods or functions to improve readability and maintainability.

**Avoid Negative Conditionals:** Minimize the use of negative conditionals (e.g., if not) to enhance code clarity and readability. Functions Should Do One Thing (G30): Ensure that functions or methods have a single responsibility and do one thing well, avoiding bloated or overly complex implementations.

**Hidden Temporal Couplings:** Be aware of hidden temporal couplings (dependencies based on timing or order of operations) and address them to improve code reliability and maintainability.

**Don't Be Arbitrary:** Avoid arbitrary decisions or choices in code design or implementation, striving for clarity and consistency instead.

**Encapsulate Boundary Conditions:** Encapsulate handling of boundary conditions within well-defined methods or functions to improve code readability and reliability.

**Functions Should Descend Only One Level of Abstraction:** Maintain a consistent level of abstraction within functions or methods, avoiding excessive nesting or complexity.

**Keep Configurable Data at High Levels:** Keep configurable or parameterized data at higher levels of the codebase to promote reusability and flexibility.

**Avoid Transitive Navigation:** Minimize transitive navigation (e.g., chaining method calls across multiple objects) to reduce complexity and improve code clarity and maintainability.

**Java.** It refers to a programming language and, in this lesson, it has a sub lesson to provide us guidelines and best practices specifically tailored to developers working with Java codebases:

**Avoid Long Import Lists by Using Wildcards:** Instead of listing each individual class to import, consider using wildcard imports (e.g., import java.util.\*) to import all classes within a package. However, be cautious not to import entire packages if only a few classes are needed to prevent namespace pollution and potential conflicts.

**Don't Inherit Constants:** Avoid inheriting constants from interfaces or classes, as it can lead to tight coupling and inheritance-related issues. Instead, consider using static imports or placing constants in utility classes to make them easily accessible without inheritance.

**Constants versus Enums:** Choose between using constants or enums based on the context and requirements of your application. Constants are suitable for simple, fixed values, while enums are preferred for representing a fixed set of related values with associated behaviors. Consider the readability, maintainability, and flexibility of each approach when deciding between constants and enums in your codebase.

**Names.** Refers to the identifiers used for variables, functions, classes, methods, parameters, and other elements within a codebase and in this lesson, it gives us guidance on selecting appropriate and descriptive names for variables, functions, classes, and other elements within your codebase:

**Choose Descriptive Names:** Select names that clearly and accurately convey the purpose or intent of the variable, function, class, etc. Choose names that make the code easier to understand and maintain.

**Choose Names at the Appropriate Level of Abstraction:** Select names that reflect the level of abstraction at which the entity operates. Names should be neither too specific nor too general, aligning with the level of detail appropriate for the context.

**Use Standard Nomenclature Where Possible:** Utilize common naming conventions and patterns within the programming language or domain to enhance consistency and readability. Follow established conventions to make your code more familiar and understandable to others.

**Unambiguous Names:** Ensure that names are clear and unambiguous, avoiding vague or confusing terminology. Names should accurately represent the purpose or behavior of the entity they refer to, leaving no room for misinterpretation.

**Use Long Names for Long Scopes:** Choose longer, more descriptive names for variables or functions with longer scopes, such as global variables or functions used across multiple modules or files. Longer names can provide additional context and clarity within larger scopes.

**Avoid Encodings:** Refrain from using encoding schemes, such as Hungarian notation or prefixing variable names with data types, as they can reduce code readability and maintainability. Instead, rely on descriptive names to convey relevant information about variables and functions.

**Names Should Describe Side-Effects:** When naming functions or methods that have side-effects (i.e., modify state or perform actions beyond returning a value), choose names that clearly indicate their behavior to prevent unexpected consequences and improve code understanding.

**Tests.** Refers to a set of procedures, scripts, or code segments written to assess the behavior, functionality, and correctness of software applications or components and in this lesson, it gives us guidelines what are the best practices and guidelines for writing effective and comprehensive tests:

**Insufficient Tests:** Ensure that the test suite provides sufficient coverage to verify the behavior and functionality of the code adequately. Insufficient tests may lead to undiscovered bugs and decrease confidence in the codebase.

**Use a Coverage Tool!:** Utilize coverage tools to assess the effectiveness of the test suite and identify areas of code that are not adequately covered by tests. Coverage tools help ensure that critical parts of the code are thoroughly tested.

**Don't Skip Trivial Tests:** Even seemingly trivial or simple functionality should be tested to ensure correctness and prevent regressions. Skipping trivial tests may lead to unexpected issues and undermine the reliability of the codebase.

**An Ignored Test Is a Question about an Ambiguity:** Treat ignored tests as indicators of ambiguity or uncertainty in the codebase. Ignored tests should be addressed promptly to clarify requirements and improve the understanding of the system.

**Test Boundary Conditions:** Test boundary conditions and edge cases to validate the behavior of the code under extreme or exceptional circumstances. Boundary condition testing helps identify potential vulnerabilities and ensures robustness.

**Exhaustively Test Near Bugs:** After discovering bugs or issues in the code, write tests to reproduce and validate the behavior near those bugs. Testing near bugs helps ensure that fixes are effective and prevent similar issues from occurring in the future.

**Patterns of Failure Are Revealing:** Analyze patterns of test failures to identify recurring issues or areas of weakness in the codebase. Patterns of failure can provide valuable insights into underlying problems and guide improvements in the test suite and code quality.

**Test Coverage Patterns Can Be Revealing:** Similarly, analyze patterns of test coverage to identify gaps or deficiencies in the test suite. Test coverage patterns can highlight areas of the code that require additional testing and help prioritize testing efforts.

**Tests Should Be Fast:** Ensure that tests execute quickly to support rapid feedback during development and continuous integration processes. Slow tests can impede developer productivity and delay the feedback loop, slowing down the development cycle.