|  | **Negative Examples** | **Positive Examples** |
| --- | --- | --- |
| **Modularity** | * "Spaghetti code" *The project contains spaghetti code, e.g. it lacks a clear organization.* * "Unrelated tasks in functions" *Most of the classes and functions perform many unrelated tasks and/or their bodies are large.* * "High degree of coupling" *The degree of coupling is high, e.g. one object depends on each of the other objects.* | * "Clear structure" *Project structure is clear since the code is organized in coherent packages, folders, files, etc.* * "Limited scope of classes and functions" *Most of the classes and functions perform a limited set of tasks and their bodies are limited in length.* |
| **Data Types** | * "Wrong Datatypes" *The choice of some data types is wrong, e.g. an integer is used when a boolean is enough.* * "Too complex data structures" *Complex data structures are used when are not needed, e.g. primitive-data array vs Object-data array.* | * "Variables and attributes" *Appropriate data type selection for variables and attributes.* * "Function/Method return" *Appropriate data type selection for function/method return.* |
| **Readability** | * "Wrong formatting" *Formatting is usually missing, poor or it is used wrongly.* * "Bad layout" *The layout of the code is not easy to read.* * "Poor naming" *Some names appear unreadable, meaningless, misleading and/or do not meet naming conventions.* * "Bad comments" *Comments are generally missing or explain obvious issues, such as what the code statement is doing.* | * "Well formated code" *Indentation, line breaks, spacing and brackets fully clarify program structure.* * "Good names" *Meaningful identifiers which meet naming conventions are used as variables, functions and class names.* * "Comments explain decisions" *Comments do not explain what the code is doing, instead explain tricky or important decisions.* * "Comments enhance understanding" *Comments are present where strictly needed and enhance understanding of the code.* |
| **DRY principle** | * "Repeated code" *Repeat snippets of code quite often.* * "Magic numbers" *Use magic (hard-coding) numbers or string literals.* | * "Helper functions are used" *Helper functions are used in order to reuse code.* * "Use of constants" *Constants are used and they are kept in a common place.* |
| **Flow** | * "Spaghetti code" *The project contains spaghetti code, e.g. it lacks a clear organization.* * "Unrelated tasks in functions" *Most of the classes and functions perform many unrelated tasks and/or their bodies are large.* * "High degree of coupling" *The degree of coupling is high, e.g. one object depends on each of the other objects.* | * "Simple flow" *Flow is simple so that the most common path through the code is clearly visible.* * "Traceability" *Traceability: it is easy to verify know which code line corresponds to which program requirement/s.* |
| **API Documentation** | * "Missing information" *Information is generally missing, redundant, incomplete or misspelled at the top of the file.* * "Authors are not documented" *Documentation about the author is missing.* * "Classes are not documented" *Documentation about the class/module is missing or incomplete.* * "Fields are not documented" *Documentation about the fields is missing or incomplete.* * "Methods are not documented" *Documentation about the methods is missing or incomplete.* | * "Authors are documented" *At the top of the file, there is a block comment in which the programmer provides author’s names.* * "Version and goal are documented" *The summary of the goal of the file and its version.* * "Attributes documented" *Documentation about attributes is correct.* * "Methods documented" *Documentation about methods is correct.* * "Information is given" *Information is generally present and provides a brief description.* * "Conditions are documented" *It contains pre and post conditions.* * "Parameters are documented" *The meaning/role of each parameter is clear.* |
| **Correctness** | * "Not compiling" *The code does not compile and run cleanly.* * "Specifications not met" *The program does not meet some of the specifications.* | * "Functions properly" *Program conforms to the specifications provided by the assignment.* * "Correct results" *It produces correct results for correct inputs.* |
| **Robustness** | * "No error handling" *Errors or abnormal conditions are not all handled.* | * "Errors handled properly" *The program reacts properly to abnormal conditions and erroneous inputs.* |
| **Traceability** | * "Tests not verifiable" *It is hard to verify that the tests cover all the program requirements/specification.* | * "Clear tests" *Tests are clear so that it is easy to detect if any requirement is left out of the tests.* * "Tests understandable" *It is easy to know what requirements are evaluated by each test case/suite.* |
| **Test Completeness** | * "Not thoroughly tested" *Some specification/requirement is not thoroughly tested, e.g. a test case checks correct input but it does not check the behavior of the program with anomalous/exceptional inputs.* | * "Enough tests" *For each specification/requirement that a test suite covers, there are enough tests cases to validate it.* |
| **PG - Extern - Analysis** |  | * "Definitions and signatures" *Being able to answer questions about the analysis considerations in terms of context description/examples of I/O, definitions, description and parameter and return types.* * "Understanding requirements" *Understanding a description in natural language of the program requirements.* * "Analysis considerations." *If asked for: describing the analysis considerations.* |
| **PG - Extern - Design** |  | * "Suitable entities and types." *Proposing suitable classes, methods, parameter and return types for the API.* |
| **PG - Extern - Specification** |  | * "Informal pre- and postconditions" *Giving informal preconditions and postconditions for the methods in the API in terms of the application domain.* * "Specification for robustness" *Providing robustness by specifying the desired behavior in case satisfying the postcondition is not possible (e.g. throwing exceptions, asking for user input, using default values).* * "Well-named subspecifications" *Organizing the case analysis in the specification into well-named subspecifications.* * "Completeness of subcontracts" *Completeness of subcontracts* |
| **PG - Extern - Tests** |  | * "Test coverage" *Proposing a sufficient number of test cases to cover all the cases in the specification, including both typical and edge values.* |
| **PG - Intern - Analaysis** |  | * "Data types" *Being able to answer questions about the analysis considerations in terms of the choices of data types and the problem decomposition.* * "Analysis considerations" *If asked for: a description of the analysis considerations.* |
| **PG - Intern - Design** |  | * "Hhigh cohesion/low coupling, separation" *Decomposing the problem into classes and methods with high cohesion/low coupling and clear responsibilities.* * "Types of the class attributes" *Deciding on type of the class attributes necessary for realizing the class’s responsibilities.* * "Names reflect purposes" *Coming up with names for classes, attributes, (private) methods, parameters (name and type of) that reflect their purpose.* |
| **PG - Intern - Specification** |  | * "Invariants" *Providing a representation invariant that describes the relationship between application domain concepts and data types.* * "Less on domain knowledge" *Translating the external method specifications, using the representation invariant, into versions that rely less on domain knowledge.* * "Specifications for internal methods/classes" *Providing specifications for private methods and methods of helper-objects introduced in the internal design.* |
| **PG - Intern - Tests** |  | * "Grey box tests" *Extending the tests of public methods using their internal specification (grey box tests).* * "Tests for internal methods" *Adding tests for the methods introduced in the internal design, i.e. private methods and helper-objects (grey box) .* |
| **PG - Impl - Analysis** |  | * "Considering re-use" *Consideration of the advantages of using library components or to design build own software.* |
| **PG - Impl - Design** |  | * "Algorithm selection" *Deciding on an algorithm that satisfies the specification and exhibits reasonable efficiency.* * "Maintainability" *Providing for maintainability through replacing switches by subclassing, avoiding loop exit jumps, avoiding external calls from temporarily corrupt objects, etc.* |
| **PG - Impl - Coding** |  | * "Correct algorithm implementation" *Providing code that implements the chosen algorithms and does not generate compiler errors or warnings.* |
| **PG - Impl - Tests** |  | * "Test coverage" *Adding tests to ensure that all code is covered.* * "Tests for code risks" *Adding tests to check for risks in the code (e.g. division by zero, overflow, file I/O).* * "Running all tests." *Running all tests.* |