|  | **Negative Examples** | **Positive Examples** |
| --- | --- | --- |
| **Modularity** | * Unclear structure Project structure is unclear since the code is not organized in coherent packages, folders, files, etc. * Unrelated tasks in functions/low cohesion Most of the classes and functions perform many unrelated tasks and/or their bodies are large. * High degree of coupling Modules (classes, objects) depend heavily on each other to function properly. Modules are not very well encapsulated. * No information hiding The visibility of methods/fields is not well thought out (fields should usually be private). | * Clear structure Project structure is clear since the code is organized in coherent packages, folders, files, etc. * Clear-cut scope of classes and functions/high cohesion Most of the classes and functions perform a limited set of tasks and their bodies are limited in length. * Low degree of coupling Modules are well encapsulated. They only depend on each other if necessary. * Information hiding The modifiers for visibility are very well thought out. Each module (class, method, package) only has access to intended methods. |
| **Data Types** | * Wrong data types The choice of some data types is wrong, e.g., an integer is used when a boolean is more suitable. * Unnecessary complex types Unnecessary use of more complex types. E.g., the result of an integer computation is stored in a double variable. * Unnecessary type casts Manually casting types unnecessarily. E.g., (Object) "Hello World!". * Confusing implicit casting Implicitly using toString() when not intended. | * Appropriate data types The choice of the data types is correct, e.g., an integer is used when an integer is necessary * Appropriate data structures Complex data structures are only used when needed, e.g., ArrayList instead of array when size varies dynamically. No unnecessary coercion or casts necessary. |
| **Readability** | * Wrong indentation Indentation of code is not up to code conventions. * Broken lines of code Code lines are broken badly. They either contain too much code or line-breaks are unnecessarily placed. * Parentheses wrongly placed Parentheses are placed in ways contrary to code conventions. * Poor naming Some names appear unreadable, meaningless, misleading and/or do not meet naming conventions. * Bad comments Comments are usually missing or only explain obvious issues, such as what the code statement is doing. | * Correct indentation Indentation conforms to the code conventions. * Considered lines of code  Complex lines of code are broken up in smaller, easier to understand and coherent chunks. * Correctly placed parenthesis Placement of parenthesis conforms to the code conventions. * Good names Meaningful identifiers which meet naming conventions are used as variables, functions and class names. * Useful comments Comments explain tricky or important decisions. |
| **DRY principle** | * Repeated code Identical code snippets are repeated. * Magic numbers Frequently used numbers or Strings with a specific meaning and/or derived values are hard-coded (magic numbers). | * Helper functions are used Helper functions are used in order to reuse code. * Use of constants Symbolic (named) constants are used and they are kept in a common place. |
| **Flow** | * Obscured flow Code order of blocks obscures intuition of assignment. * Unnecessary branching A conditional statement is used when instead the if-predicate should be used in an expression. For example: if(x) return true else return false; instead of return x; * Unnecessary nesting Nested if-statements are used when not strictly necessary or useful for readability. E.g., if(a) if(b) instead of if(a && b). * Switch/if often mixed up Multiple if-blocks are frequently used instead of using switch or a switch statement is used for binary decisions. | * Simple flow Code order of blocks corresponds to intuition of assignment. * Only necessary branches Code only branches when necessary, not when possible. * Only necessary nesting Nested conditionals are only used when necessary. * Correct use of Switch/if Switch is used instead of multiple if-blocks. If-statements are used for binary decisions. |
| **API Documentation** | * Class Documentation is missing Author, version and goal of the class are missing. * Attributes not well documented Incorrect or insufficient documentation of attributes. * Missing method documentation Documentation of methods is flawed. Either pre- and post-conditions are missing or the method is not well explained (including a summarizing first sentence). * Incorrect method signature documentation The documentation for the signature of a method is flawed. Either parameters are not well documented (wrong position, bad explanation) or exceptions are not documented. | * Classes are well documented One or more of author, version and the goal of the class are mentioned within the documentation. * Attributes documented Documentation about attributes is correct. * Method functionality documented Documentation about methods is correct. It contains a meaningful first sentence and documentation of pre- post conditions. * Documentation of a methods signature is correct. Parameters as well as possible exceptions are mentioned and well documented. |
| **Correctness** | * Wrong format The file format of the submitted code is not in the correct format. E.g., a PDF is used instead of a Java file. * Not compiling/running The code does not compile or does not run without errors. * Specifications not met The program does not meet some of the specifications in the assignment. | * Functions properly The program conforms to the specifications provided by the assignment. * Correct results The program produces correct results for correct inputs. |
| **Robustness** | * No error handling Errors or abnormal conditions are not all handled. * Edge cases not handled Edge cases are mostly ignored within the implementation. E.g., empty strings, empty lists, and so on. * No parameter validation Validation of parameters is mostly missing. E.g., it is possible to pass negative numbers to a function over natural numbers. | * Errors handled properly The program reacts properly to abnormal conditions and erroneous inputs. * Edge cases handled Edge cases are handled within the method. E.g., empty strings, empty lists, etc. are handled within the code. * Parameters are validated Parameters are validated. It is not possible to enter forbidden input into a method. |
| **Test Traceability** | * Test scenario clear The test scenario is not prepared clearly or preparation is mixed with the test execution. * Tests multiple units A test asserts properties of multiple methods (e.g., method calls and assertions alternate a lot). * Not traceable It is not easy to see a relation between the specifications in the assignment and tests. * Bad test naming The names of the tests do not clearly describe the test case. | * Test scenario clear The creation of the context in which the tested functionality is performed is meaningful. * Tests one unit One test asserts properties of just one method. * Traceable For each test the underlying requirements are easily recognizable. * Good test naming The names of tests clearly describe the test cases. |
| **Test Completeness** | * Insufficient specification coverage Some specifications are not tested at all or are tested insufficiently. * Insufficient coverage for the happy path The most common path through a program is not sufficiently tested. E.g., there are common branches that get ignored in testing. * Insufficient coverage of edge cases Some or all edge cases get ignored in testing. E.g., 0 or empty Strings are ignored in testing. * Insufficient coverage of exceptional path Less frequent branches are not sufficiently tested. * Thrown exceptions are not tested A method that is supposed to throw an exception in a certain case is not tested to do so. * Incorrect assertions The expected values in assertions are wrong or conditions are too weak, such that incorrect programs pass the tests. | * Sufficient specification coverage All specifications of the program are tested properly and sufficiently. * Sufficient coverage happy path The most common path through a program is sufficiently tested. * Sufficient coverage of edge cases All edge cases are tested for every method within the program. * Sufficient coverage of exceptional path Infrequent or exceptional paths through the program are tested sufficiently. * Thrown exceptions are tested It is tested if every method that throws exceptions does indeed throw the exception in question in the correct situations. * Correct Assertions The expected values in assertions are correct and strong enough. Only correct programs can pass the tests. |
| **PG - Extern - Analysis** | * Not proper definitions No or not all domain/model variables are defined. * Not proper context The context of operation of the class or method is not or not properly defined. * Not proper examples of I/O for happy path No or not enough examples of I/O for happy path are provided. * Not proper examples of I/O for robustness No or not enough examples of I/O of non happy path are provided. | * Definitions All domain/model variables are defined. * Context The context of operation of the class or method is defined. * Examples of I/O for happy path Enough examples of I/O for happy path are provided. * Examples of I/O for robustness Enough examples of I/O of non happy path are provided. |
| **PG - Extern - Design** | * Not proper class definition A suitable class definition (name) is not given. * Method signatures   Suitable method signatures (method name, parameters, and return type) are not given. | * Class definition A suitable class definition (name) is given. * Method signatures   Suitable method signatures (method name, parameters, and return type) are given. |
| **PG - Extern - Specification** | * No class invariant   In case of model (class) variables no class invariant is provided.   * No happy path pre- and postconditions No informal preconditions and postconditions for the methods in the API are provided for happy path. * No specification for robustness No robustness is provided by specifying the desired behavior in case of non expected input (e.g., throwing exceptions, asking for user input, using default values). * No well-chosen subcontracts The case analysis in the contract is well-organized into well-named subcontracts. * No completeness of subcontracts The subcontract are not complete. | * Class invariant   In case of model (class) variables a class invariant is provided.   * Happy path pre- and postconditions Informal preconditions and postconditions for the methods in the API are provided for happy path. * Specification for robustness Robustness is provided by specifying the desired behavior in case of non expected input (e.g., throwing exceptions, asking for user input, using default values). * Well-chosen subcontracts The case analysis in the contract is well-organized into well-named subcontracts. * Completeness of subcontracts The subcontracts are complete. |
| **PG - Extern - Tests** | No sufficient coverage A insufficient number of test cases to cover all (sub)contract(s), including typical, edge and exception values are provided. | * Test coverage A sufficient number of test cases to cover all (sub)contract(s), including typical, edge and exception values are provided. |
| **PG - Intern - Analaysis** | * No Consideration of class internals Appropriate attributes and types are not considered. * No Consideration of relation domain versus class attributes The relation between domain variables and class attributes is made clear. * No consideration of extra methods and classes   The need for private methods and extra classes is not considered. | * Consider class internals Appropriate attributes and types are considered. * Consider relation domain versus class attributes The relation between domain variables and class attributes is clear. * Consider extra methods and classes   The need for private methods and extra classes is considered. |
| **PG - Intern - Design** | * No attributes are added Not all attributes necessary for realizing the class’s responsibilities are added. * Types of attributes are not correct The attributes are not of appropriate type necessary for realizing the class’s responsibilities. * Names do not reflect purposes Attributes, private methods and parameters do not have names that reflect their purpose. | * Attributes are added The attributes necessary for realizing the class’s responsibilities are added. * Types of attributes The attributes are of appropriate type necessary for realizing the class’s responsibilities. * Names reflect purposes Attributes, private methods and parameters have names that reflect their purpose. |
| **PG - Intern - Specification** |  | * Representation relation The domain concepts are translated into and related to the attributes. * representation invariants If needed, representation invariants are provided. * Less on domain knowledge The external method specifications, using the representation relation and invariants, are translated into versions that rely less on domain knowledge. * Specifications for internal methods/classes Specifications for private methods and methods of helper-objects introduced in the internal design are provided. |
| **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. |
|  |  |  |
|  |  |  |
| **Design class diagram** | * No class diagram   No class diagram is given.   * No multiplicities exist   Multiplicities are not added.   * Multiplicities are incorrect   Multiplicities are added but are incorrect.   * No associations included   No associations are added.   * Associations has incorrect directions   Associations are included but have incorrect  directions.   * MDD notation (instead of UML notation)   You applied MDD notation instead of UML notation. | * Class diagram given   A class diagram is included.   * Multiplicities included   Multiplicities are included.   * Multiplicities are correct   Multiplicities are correct.   * Association included   Associations are included.   * Associations have correct directions   Associations have correct directions.   * UML notation (no MDD notation)   The (UML) notation is correct applied. |
| **Class hierarchy- /interfaces/abstract classes** | * No class hierarchy used   A class hierarchy misses in your design.   * No interface or abstract class used   No interface or abstract class is used as top class in the class hierarchy. | * Class hierarchy defined   A class hierarchy exists in the design.   * Interface or abstract class applied   The top of the hierarchy is an interface or abstract class.   * Hierarchy is logical   The class hierarchy is logical. |
| **Comments/JDoc** | * No JDoc or comments added   No JDoc is included. | * JDoc added   JDoc is included.   * If needed, inline comments added   When a codefragment is complex, inline comment is added.   * JDoc satisfies the requirements   The JDoc satisfies the requirements. |
|  |  |  |
| **Code** | * Code does not compile   The code does not compile.   * Code readability le   The code is ba |  |
|  |  |  |
|  |  |  |
|  |  |  |