Acta Univ. Sapientiae, Informatica, **5,** 1 (2013) 77–102

DOI: 10.2478/ausi-2014-0006

Test software quality issues and connections to international standards

Attila KOVÁCS
Eötvös Loránd University
Faculty of Informatics
email:
`attila.kovacs@compalg.inf.elte.hu`

Kristóf SZABADOS[1]
Eötvös Loránd University
Faculty of Informatics
email:
`kristof.szabados@ericsson.com`

Abstract. This paper examines how ISO/IEC 9126-1 and ISO/IEC 25010 quality models can be applied to software testing products in industrial environment. We present a set of code smells for test systems written in TTCN-3 and their categorization according to quality model standards. We demonstrate our measurements on industrial and ETSI projects, and provide a method for estimating their effects on product risks in current projects.

1 Introduction

In our fast changing world the usage of electrical devices belongs to the everyday life of the society. These devices contain software helping the navigation to destinations, supporting the communication with other people, driving the production, distribution and consumption of energy resources. Software drives companies, trades on the markets, takes care of people's health.

All of these systems must fulfill very strict (but different) quality restrictions. In the telecommunication area "five nines" (99.999%) availability allows only 5.26 minutes downtime per year — often including planned upgrades and maintenance. Companies producing these systems perform strategically several

Computing Classification System 1998: D.2.9
Mathematics Subject Classification 2010: 68Q60
Key words and phrases: ISO/IEC 9126, ISO/IEC 25010, TTCN-3, test system architecture, software quality

[1]Corresponding author

activities to ensure the required level of quality. In 1986 the software engineer Barry Boehm observed that the cost of detecting, reporting and correcting defects increases exponentially by the time they are found in the software development process [2]. At that time the overall magnitude of software costs was estimated to be roughly \$140 billion per year, worldwide. Since then the size and complexity of software systems have been growing constantly together with the quality expectations against these systems.

Clearly, as the size and complexity of software systems grows, so does the size and complexity of their tests. Before Y2K tests were mostly designed and executed manually. Nowadays every corporation aims at automating their tests which produces large scale test architectures. In the telecom area this pressure facilitated the ETSI[2] to develop a scripting language used in conformance testing of communicating systems and a specification of test infrastructure interfaces that glue abstract test scripts with concrete communication environments. This programming language standard is called TTCN-3[3] and offers potentials for reducing test maintenance costs significantly.

Companies developing complex software systems require quality standards, models and methods to define, perform and institutionalize their quality management processes. The ISO[4] and IEC[5] published the multiple standards 9126 [11] and 25000 [16] which approach software *product* quality. Other standards like ISO/IEC 9001 or CMMI[6] [4] focus on the quality of the software *processes.* GQM[7] [25] describes *measurement* techniques used in software development, while PSP[8] [27] and TSP[9] [10] aims at the human resources and personal processes used during software development.

Regarding the software testing area applying TMMi[10] organizations can improve their test processes. ISTQB[11] offers certifications of competences in software testing. A wide range of software testing tools supports the testing processes, including test management, functional and non-functional testing, etc. Yet, the theoretical research and graphical tool support lacks behind these quality trends in testing.

[2]European Telecommunications Standards Institute
[3]Testing and Test Control Notation – 3
[4]International Organization for Standardization
[5]International Electrotechnical Commission
[6]Capability Maturity Model Integration
[7]Goal Question Metric
[8]Personal Software Process
[9]Team Software Process
[10]Test Maturity Model Integration
[11]International Software Testing Qualifications Board

The paper is organized as follows. In Section 2 we present the earlier results related to our subject. In Section 3 we give a bird's-eye view on ISO/IEC 9126 and ISO/IEC 25010 models. Section 4 contains the list of code smells we have analyzed and the categorization of them. We implemented and measured the quality of the tests via these code smells on actual industrial projects and we present the findings in Section 5. Section 6 summarizes the results of this paper.

2 Earlier results and related work

In 2007 Zeiss et al. [29] published a model for test specification derived from ISO 9126 concentrating only on internal quality attributes. This model is divided into 7 main characteristics: *test effectivity*, *reliability*, *usability*, *efficiency*, *maintainability*, *portability* and *re-usability*. Each main characteristic is divided into sub-characteristics. Most of these main and sub-characteristics are re-interpreted and re-named from ISO 9126 in order to be more appropriate in the context of testing. For example the *suitability* sub-characteristic of the *functionality* main characteristic in the ISO 9126 model is renamed to *test effectivity / test coverage*. The reason for this, as they explained, is that "in the context of test specification, the *suitability* aspect is characterized by the *test coverage*. Coverage constitutes a measure for test completeness". Another important part of this work is the definition of the *re-usability* main characteristic. *Re-usability* is an important aspect in case of test specifications. The specifications of functional tests, performance tests and load tests might be different, but the test data could be reused in all of these test suites. Another example could be functional tests: once developed, they might be used for regression testing purposes.

In the present work we take a different way of looking at the tests. Our attention is to look at tests as software products. So, instead of re-interpreting the quality standards for testing, we re-interpret the testing for software product quality. The previously mentioned article chooses TTCN-3 as a *test specification language*. In this work TTCN-3 is viewed as a *programming language*. Software products written in TTCN-3 have to be analyzed in order to fulfill quality requirements by applying quality metrics. As we see the two standpoints is not contradicting but rather complementing each other.

In the paper of Bánsághi et al. [1] one of the cornerstones was the comparison of the models ISO 9126 and ISO 25010. The article comes to the conclusion that even though the new model is broader, both models suffer from the fact that “different parties with different views of software quality can select different definitions”. They state that although both of the standards offer a good frame of reference for software product quality, neither of them offer a practically applicable method for *assessing* quality. The other cornerstone was the fact that there is a wast literature proposing numerous ways of measuring software quality metrics without providing traceable and easily applicable translation to the multi-faceted notation of quality. To bridge this gap the authors analyzed the rules of the PMD [24] and FxCop [8] static source code analyzers and classified them into the quality characteristics of the standards. PMD analyzes Java source code and looks for possible code smells, like duplicate code, unused local variables, etc. FxCop analyzes managed code assemblies in the .NET framework common language runtime. They have classified 203 PMD and 225 FxCop rules into ISO 9126 and ISO 25010 categories. They have found that most rules fall into the *maintainability* characteristic in both standards.

This work tries to extend [1] with quality rules for testing systems.

The following questions arise naturally: are the TTCN-3 language and the test systems written in it complex enough for such an analysis? Is it worth to regard the test language as a programming language? The answer is yes. It was already shown [26] that current testing systems written in TTCN-3 are not only large in size, but are also very complex. The importation graph-structure of these software systems shows scale-free properties. This was one of our main motivations why the TTCN-3 language and systems need to be studied deeper.

3 Bird’s-eye view on ISO/IEC product quality standards

It is well-known that specification and evaluation of software quality attributes are key factors to ensure adequate business quality. Yet, software quality is still considered to be a trait that can not be measured [17]. This attribute is based on the fact that in general quality is a multidimensional concept. The understanding of quality depends on the entity of interest, attributes of that entity and the viewpoint of the observer. In order to achieve high software quality appropriate quality characteristics must be defined and taken into account. These characteristics should drive the significant architectural and design decisions. Software quality management should help to ensure that the required level of quality is reached.

3.1 ISO/IEC 9126

ISO/IEC 9126-1 defines quality as “the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs” [11]. This standard has later been revised and separated into 4 parts: part one defines the characteristics and viewpoints of software product quality [12]; part two provides metrics for the external quality, as the totality of characteristics of the software product from an external point of view [13]; part three assesses a software system’s internal quality, as the totality of characteristics of a software product from an internal point of view [14]. It provides metrics that measure the software itself. Part four evaluates metrics by the quality in use, as the end user’s view of the quality of the software product, as it is used in the specific usage context [15].

ISO/IEC 9126-1 contains a two part model. The first part is applicable for modelling the internal and external quality of a product, while the second part models the quality in use of a software product. Generally, *internal quality* is understood as something attainable on reviews by doing static code analysis. Internal quality can be used early in the development process to predict the quality of the final product. *External quality* describes properties of software as it interacts with its environment. *Quality in use* is perceived by the end user who executes the software product. These product quality views represent different stages of the development, but instead of being completely independent, they influence each other.

ISO/IEC 9126-1 defines the same frame for modelling internal and external quality (shown on Figure 1), which can be instantiated for both using different set of metrics. The model itself describes 6 characteristics: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*. Each of these characteristics has several further sub-characteristics. The importance of the characteristics varies depending on the view point concerned.

3.2 ISO/IEC 25010

ISO/IEC 9126 has been recently replaced by ISO/IEC 25010 [16]. It revises the old standard and incorporates the same characteristics with some amendments.

- The scope of quality models includes computer systems and quality in use from system perspective.
- Several new characteristics and sub-characteristics were added. Some characteristics were escalated from being sub-characteristics (security), some were removed, some were renamed to have more appropriate names.

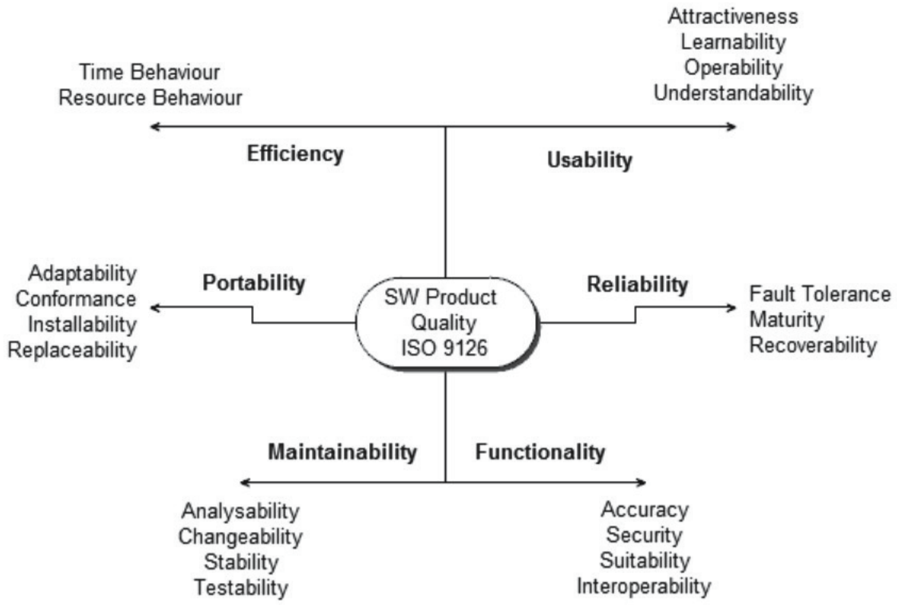

Figure 1: Quality characteristics of a software product in ISO/IEC 9126

- Internal and external quality models were combined as the product quality model.

This standard states that "this International Standard is intended to be used in conjunction with the other parts of SQuaRE series of International Standards (ISO/IEC 25000 to ISO/IEC 25099), and with ISO/IEC 14598 until superseded by the ISO/IEC 2504n series of International Standards".

ISO/IEC 25010 categorizes the product quality properties into eight characteristics: *functional suitability*, *performance efficiency*, *compatibility*, *usability*, *reliability*, *security*, *maintainability* and *portability*. With each characteristics being composed of a set of related sub-characteristics (see Figure 2).

Quality in use attributes are categorized into five characteristics: *effectiveness*, *efficiency*, *satisfaction*, *freedom from risk*, *context coverage*. The definition of quality in use compared to ISO/IEC 9126 also includes the quality of hardware, operating environment and the characteristics of users, tasks and social environment as having an effect on stakeholders.

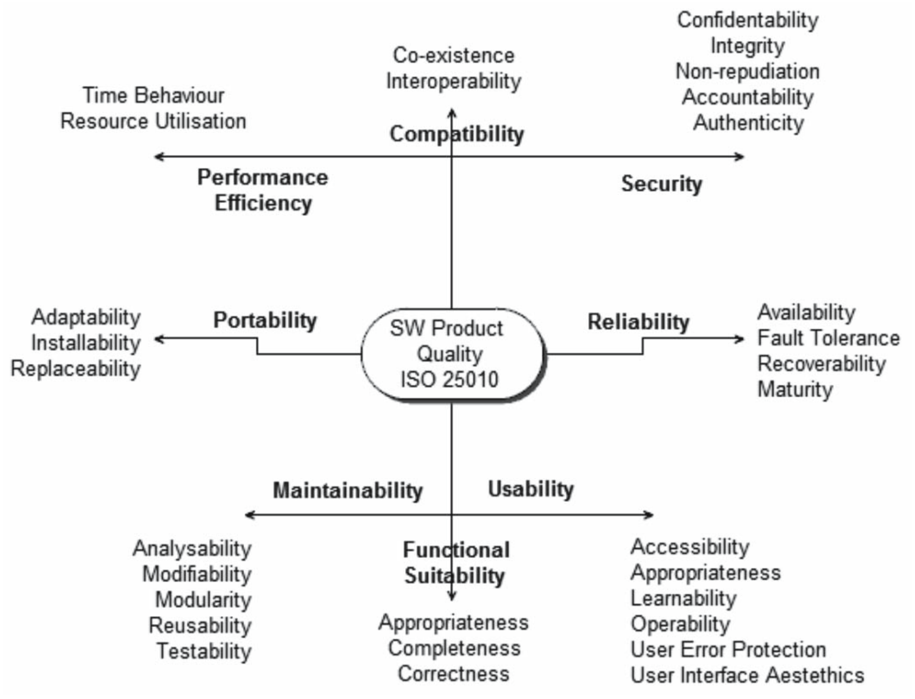

Figure 2: Software product quality according to ISO/IEC 25010

4 Code smells and categorization

As we stated earlier our starting point is different from the previous researches. They usually tried to map constructs from the testing context onto software product quality rules. Our point of view approximates from the other direction, i.e., we map software product quality constructs on software systems written in TTCN-3. We started to analyze existing systems which are actually used for testing.

Another serious difference is that we are able to consider incomplete systems: software products which are part of some larger software products (and might contain smaller software products within themselves). In this context software code itself is viewed as a product. For example a library of functions that can be used to send and receive messages on a protocol, or a framework that enables the user to do load testing (after configuring the specific messages to be sent/received). Both examples are software systems that could be used in a standalone mode or as part of a bigger architecture. At the same time both examples share the property of being products that are used by other programmers or test designers. These software products are delivered to the customer in the form of source codes.

This point of view means that *usability* also becomes a characteristic that can be measured by a static source code analyzer. This way *usability* and *maintainability* seems to be hard to separate so we have to refine the distinction. In our understanding *maintainability* means the attributes related to maintaining the actual product, *usability* means the usage from a higher abstraction level or by an external product. In case of a TTCN-3 function, in a vague sense, this could mean that the quality attributes of the function's body most likely belong to the *maintainability* characteristic, while the quality attributes of the formal parameter list belongs to the *usability* characteristic. If a function has too many formal parameters than it may cause problems for the party trying to invoke (use) it. While the actual implementation of some feature is the internal responsibility of the product, it could be changed without creating any noticeable side effect for the external calling party.

4.1 Code smells

A *code smell* is a hint that something has gone wrong somewhere in the code. In order to find valid code smells that can be used to measure the quality of TTCN-3 testing systems we have analyzed the archives of the Quality Assurance Organization of a large telecom company and the code smells already detected by existing static analyzers in other languages.

First we have reviewed every Inspection Record (source code review documents) created from year 2006. We have also reviewed all entries in the Trouble Report (errors and problems found in released products) database. These records have proven to be very usefull for the rest of the research. All of them were code quality issues which may became show stoppers at some point in the project's life cycle.

Second, we have also reviewed the rules of PMD [24], FxCop [8], Checkstyle [3], FindBugs [6], xUnit Patterns [20], Martin Fowler's book on refactoring [7] and TRex [28] to see if there are any static analyzer rules that can be used in testing and in particular for the TTCN-3 language. Checkstyle is a highly configurable static code analyzer for Java, mostly directed to issues related to breaking naming conventions. Findbugs is an open source static code analyzer operating on Java bytecode. xUnit Patterns is an extensive collection of unit testing specific code smells. Martin Fowler's book provides an extensive list of code smells, their description and refactoring methods for “deodorizing them”. As TTCN-3 is not an object oriented language and was designed with a testing oriented mindset, most of the reviewed rules could not be applied. Another important difference is that the standard of the TTCN-3 language is changing rapidly. Visibility checking was introduced to the standard in year 2009 [5]. This change came with the *public*, *private* and *friend* words becoming keywords and no longer usable as identifiers. After the introduction of the keywords the source codes that contained them were no longer backward compilable.

TRex [22, 23] has an extensive list of code smells for TTCN-3 with the following entries: (1) Duplicated Code, (2) References, (3) Parameters, (4) Complexity, (5) Default Anomalies, (6) Test Behaviour, (7) Test Configuration, (8) Coding Standards (9) Data Flow Anomalies, (10) Miscellaneous, having altogether 38 smells. The Duplicated Code smell mentioned in this article describes 7 separate smells differentiating the dupplication of template data, local data, inline-data, component data, statement, conditional branches and Alt branches. They showed the practicability of their smells in the test systems *SIP v4.1.1*, *HiperMan v2.3.1* and *IPv6 v1.1* having altogether 61282 lines of code. Based on this work we extended and redesigned the list of code smells we found to be applicable to TTCN-3. They are as follows:

1. FIXME tags: Developer markings of severe incorrect or missing features.
2. TODO tags: Developer markings of incorrect or missing features.
3. Circular importation: The import relation of modules forms at least one loop.
4. Duplicated code: Very similar code exists in more than one location.
5. Similar functions: Several functions differing only in literal values.
6. Mergeable templates: Similar data structures, that could be merged into a single parameterized one.
7. Long statement blocks: A block of statements that has grown too large.
8. Too many parameters: A long list of formal parameters.
9. Excessively short identifiers: The name of an identifier is too short to reflect it’s functions.
10. Excessively long identifier: The name of an identifier is too long.
11. Divergent naming: The identifier breaks the naming conventions.
12. ”Private” group: Public definitions categorized in a group called ”private”.
13. Internal comments: Internal comments indicate too complicated code.
14. Missing comments: All methods should be commented.

15. Type in method name: The return type's name is redundant in the method name.
16. Module in method name: The containing module is mentioned in the method name.
17. Visibility embedded in name: Visibility rules evaluated by user.
18. Incomplete literals: Some fields of literals and constants are initialized unbound.
19. Initialize with constant: Structured value declared without initial value.
20. Dummy fields in constants: Field always overridden, should be left unbound.
21. Goto detection: Goto is considered to break structured programming rules.
22. Unnecessary imports: Module importations that are unnecessary.
23. Unused global definitions: Some global definitions are not used.
24. Unused local definitions: Some local definitions are not used.
25. Unnecessary operations: Operations never executed.
26. Unchecked module parameter: The module parameter is used before being checked.
27. Push definition to component: Functions running on a component define the same local variable.
28. Pull definition to local: A component member is only used in a few functions.
29. Unused return value: The result or error handling of the function call is missing.
30. Unused started return value: Information sent back is not reachable.
31. Infinite loops: Loops the code could not exit from.
32. Busy wait: Waiting for message in an event based system with polling.
33. Non-private private definitions: Public definitions used only internally.
34. Excessive rotation size: List rotation size should not exceed the size of the list.
35. Consequtive assignments to an entity: Assignments could be merged to a single assignment.
36. Sequential "if" statements: If possible should be changed to "if-else" conditions.
37. Size check in loop limit: The size of an unchanged list is checked in every iteration.
38. Reused loop variables: Loop variable declared and used outside the loop.
39. Unnecessary condition: The condition can be evaluated by the static analyzer.

40. Conditional complexity: Too large conditional logic blocks.
41. Explicit condition check: Explicitly check the value of a boolean condition.
42. Boolean evaluation with branching: All of the branches only set a single logical value.
43. Mergeable conditions: Consecutive conditionals do exactly the same operations.
44. If without else: In testing software all execution paths should be handled, at least logged.
45. Method with single condition: All statements of a function are in a single conditional.
46. Too many branches on a value: Switching on a value with consecutive conditionals.
47. Not written inout parameter: Reference passing used when not needed.
48. Not written out parameter: Result not calculated and passed back.
49. Not written variable: Variable declaration when constant would suffice.
50. Restrictable templates: Templates that could be more restricted based on their usage, but are not.
51. Dead code: Code fragment which is executed but not used anywhere.
52. Code commented out: Instead of removing it code was commented out.
53. Empty blocks: An empty code block.
54. Setverdict without reason: The testcase verdict is set without attached reason.
55. Variant outside Encodes: Encoding variants are specified without context.
56. Functions containing Stop: The execution is stopped inside a function, instead of the testcase.
57. Valueof used with value: The valueof function (used to convert a template to a value) is used with a value parameter.
58. Magic number: Numeric literals in the code.
59. Magic string: String literals inside the code.
60. XML tags in strings: XML encoding is simulated via string manipulation.
61. Nested block depth: The nesting of constructs exceeded a given level.
62. Indicent exposure: Too much of the module is exposed to the public.
63. Inappropriate intimacy: Dependencies on other module's implementation details. Functions using definitions only from an other module should be moved there. Members used only by a single external module should be moved there.
64. Feature envy: The function uses only an other module's attributes.

65. Divergent change: Changes touch completely different parts of a module.
66. Shotgun surgery: A change requires several changes in several modules.
67. PTC created, not started: A Parallel component is not started.
68. Isolated PTC: A parallel component is not connected to the test system.
69. Un-needed "runs on": There is no need for restricting a function to a specific component.
70. Contrieved complexity: Complex design patterns, where simpler would suffice.
71. Incorrect indentation: The code is not well indented.
72. Divergent naming of files: The names of files does not follow the naming conventions.
73. Incorrect pre-processability indication: Pre-processablity is not indicated in file extension.
74. Ordering of definitions: Definitions declared out of order.
75. Filling in values one-by-one: Structured value is filled in in several statements.
76. Private definitions published: A public function returns with a private definition creating a potential security hole.
77. Floating point equality check: Floating point numbers should not be compard directly.
78. Public/private keywords: The public/private keywords are used as identifiers.
79. Select without default branch: A select statement does not have "case else" branch.
80. Switch density: The ratio of branches are too high in the code.
81. Logic inversion: the whole conditional expression is negated.
82. Cyclometric complexity: The number of decision points in a method, plus one for the method entry.
83. NPath complexity: The number of acyclic execution paths in a method. Similar to Cyclometric complexity, but also takes into account the nesting of statements.
84. Break/continue usage: Break and continue statements are used incorrectly.
85. Unreachable code: A part of the code that can not be reached.
86. Using "*" for mandatory fields: Optionality is indicated for a mandatory field.

4.2 Categorization

The classification process was a technical review. In the review meeting each rule was discussed and decisions were made for belonging or not to some characteristic and sub-characteristic class. Each rule was categorized into the class to which it most likely belongs. Most likely means that more than 66% of the review meeting members agreed. In this way there were several rules which fell into multiple categories. For example the rule "infinite loops" belongs to *functionality/suitability* as most likely the program was not intended to operate like that, while it also belongs to the *efficiency/time behavior* since a program running in an infinite loop is most likely wasting resources. During the review we have agreed not to categorize the "FIXME / TODO tags" rule. The content and severity of this rule depends on the information the developers wished to make visible. As such each instance may belong to any of the characteristics, completely independently from any other instance. The result of the categorization review can be seen on Figure 3 and Figure 4.

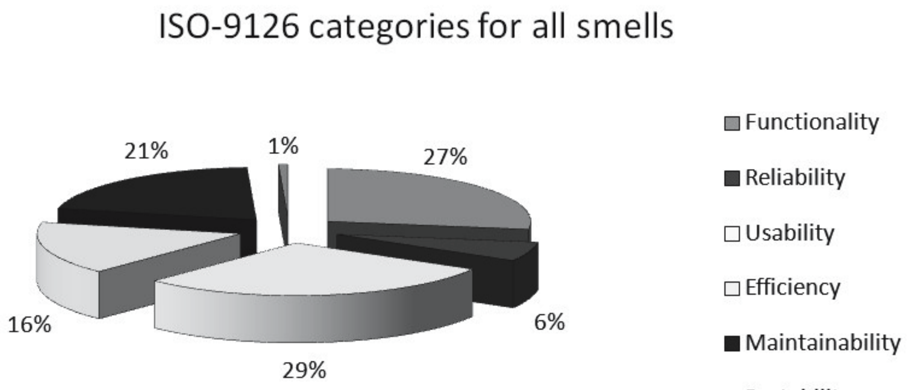

Figure 3: Code smells categorized according to ISO/IEC 9126-1

It can be observed that some of the categories of ISO/IEC 9126 kept their coverage ratio when the smells were categorized against ISO/IEC 25010, while others were re-partitioned. *Functionality*, *reliability*, *efficiency* and *portability* were almost identically mapped with respect to the categories of *functional suitability*, *reliability*, *performance efficiency* and *portability*. While the contents of the *usability* and *maintainability* categories were distributed among

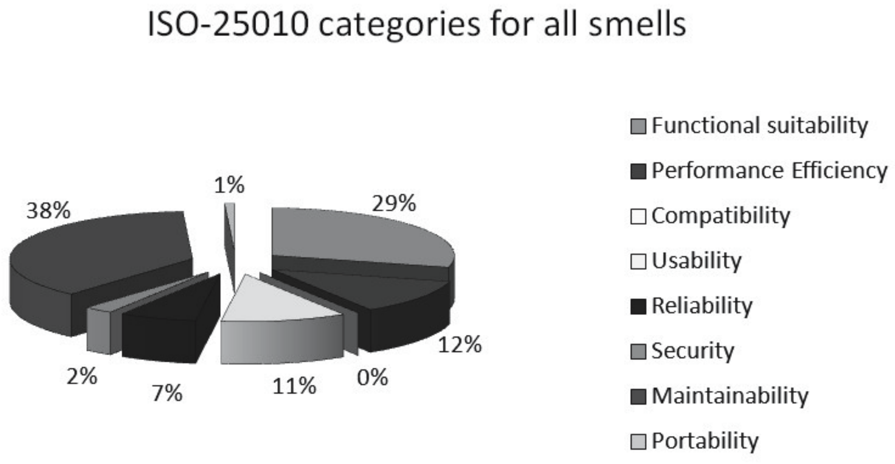

Figure 4: Code smells categorized according to ISO/IEC 25010

the *usability*, *security* and *maintainability* with much more focus placed on *maintainability*. The reason behind this comes from our understanding of these categories. For example the code smells "unused global definitions" and "unused local definitions" were categorized into *functionality* and *usability* according to ISO/IEC 9126. When evaluated against ISO/IEC 25010 we found them to be fitting only into the *functional suitability/completeness* category. This categorization is still valid, since having unused definitions (either global or local) in the source code can indicate that something might not work correctly, since some elements are unused that were planned to be used. The question of why we choose to not fit these smells into the *usability* category is somewhat less obvious. ISO/IEC 9126 defines *usability/understandability* as "attributes of software that bear on the users' effort for recognizing the logical and its applicability". The decision in the review meeting was that unused global and local definitions fall into this category, since for someone who should like to understand the relations and operations of such code, these definitions provides confusion and a large amount of extra work. ISO/IEC 25010 defines *usability* as the "degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". Although unused definitions decrease the effectiveness of using a software framework, we decided that it may have just a little obstacle to actually hinder usability. In modern IDEs the effect of these smells is only having a few more elements in the tools "code completion" or "recommendation" lists which can quickly be overcome by the user.

Another example is the case of the code smell "conditional complexity". It is clear that this is a maintenance related problem according to both standards. Yet, we decided that this code smell only fits the *usability* category in ISO/IEC 9126, but not in ISO/IEC 25010. While"conditional complexity" provides problems for users of the functionality willing to understand its internal operations, it is not necessarily hindering them to achieve specific goals in a single specified context of use.

5 Measuring the test quality via code smells

To make sure that our code smells are useful we have analyzed 16 projects. All projects were real life projects at a large telecommunication company except 6 of them which were standard ETSI TTCN-3 test projects.

The size range of this set of projects[12] stretches from 21 174 to 794 829 with the average of 205 438 effective lines of code. The number of modules per project varies from 4 to 583 with the average of 174 modules. The average module size was 1183 lines of TTCN-3 code. Some projects contain only a few tests for very simple systems, others are large frameworks with complex tests. Some of these are organized in a single project, others form hierarchies of sub-projects. We believe that this range is not only the widest studied so far, but should also cover most usages and ways of working with TTCN-3.

It is important to mention that the measurements data were gathered in the time frame of a full year. For this reason the results do not represent the general level of quality at any given point of time.

In the time-frame of the research project we were able to implement and measure 35 code smells. Most of the measured code smells are valuable as they point out existing issues. In fact, most of them were present in the projects in a large quantity. There was only a single code smell (excessive rotation size) which was not manifesting in any analyzed projects. Other smells either manifested in almost every project, or were localized in only a few ones.

1 FIXME tags: Together there were 121 occurrences, 91% of them were in 3 projects.

2 TODO tags: There were 522 occurrences, 78% of them were in 4 projects.

[12]These were complete projects, i.e., a project were considered together with all its subprojects

3 Circular importation: There were 70 occurrences. In one of the projects 25% of the modules belonged to a circular import cycle.

7 Long statement blocks[13]**:** There were 499 occurrences, 97% in two projects. One of them had a hit in every 600 lines in average.

8 Too many parameters[14]**:** There were 1648 occurrences, we have found hits in almost each project proportionally to their sizes. There were only two projects without this warning sign.

11 Divergent naming: There were 244480 occurrences, 79% of them in 5 projects. There was a project with extremely high hit ratio comparing to its size (more than 50%).

14 Missing comments: There were 2202 occurrences, 75% of them in three projects. The highest hits/project size ratio was in one of the smallest project.

15 Type in method name: There were 12343 occurrences, mainly proportional to the sizes of the projects, except one case: the fourth biggest project had no hits to this smell.

16 Module in method name: There were 731 occurrences, 81% of them in three projects. There were 5 projects without any hits.

17 Visibility embedded in name: There were 250 occurrences, 75% in two projects. One of them, a small sized project, had 88 hits. There were 11 projects without any occurrences.

19 Initialize with constant: We found together 61333 occurrences, 66% of them were in two projects. One of them (a middle sized project) had 42% of all hits.

21 Goto detection: We found occurrences in 6 projects, one (middle sized) project had the 98% of the hits.

22 Unused imports: There were 25597 occurrences, 90% in the two largest projects.

23 Unused global definitions: There were 26032 occurrences, 61% in three of the four largest projects. It is interesting to note that there were not any hits in the third largest project.

24 Unused local definitions: There were together 9717 occurrences, 57% in the largest project. In two (a large and a medium sized) projects there were no hits at all.

29 Unused return value: There were 5426 occurrences, 66% in two (a small and a middle sized) projects. In 9 projects there were no hits at all.

30 Unused started return value: There were 416 occurrences in two projects, basically one of them (a middle sized project) had all of the hits.

31 Infinite loops: There were 35 occurrences in 9 projects, 24 hits in two (a large and a middle sized) projects.

32 Busy wait: There was only one occurrence.

33 Non-private private definitions: There were 14390 occurrences, the second largest project has 21% of them. It must be noted that a middle sized project had 11% of all the hits.

34 Excessive rotation size: There was no occurrence at all.

37 Size check in loop limit: There were 3251 occurrences, 85% in three of the four largest project. In the third largest project there was only 1 hits.

40 Conditional complexity: There were together 3089 occurrences, 72% in three projects. One of them (a middle sized project) had 38% of the hits.

47 Not written inout parameter: There were 771 occurrences, 62% in two projects. The second largest project had 40% of all hits.

48 Not written out parameter: There were 68 occurrences, 55% of them in two large projects. In 6 projects there were no hits at all.

49 Not written variable: There were 36133 occurrences, 69% of them in the largest project.

53 Empty blocks: There were 3358 occurrences, 50% of them in one (middle sized) project. 63% of all the hits were in two projects.

54 Setverdict without reason: There were 15525 occurrences, 72% of them were in one (middle sized) project.

55 Variant outside encode: Altogether there were 290 occurrences in 2 (a large and a middle sized) projects.

56 Functions containing STOP**:** There were 1075 occurrences, 91% of them in the third largest project. In 6 projects there were no hits at all.

57 Valueof used with value: There were 49 occurrences in two projects, almost all hits were in the largest project.

58 Magic number: There were 54592 occurrences, 64% in two (the largest and a middle sized) projects.

59 Magic string: There were together 457288 occurrences, 74% were in two (the largest and a middle sized) projects.

81 Logic inversion: There were 2019 occurrences, 67% of them in a middle sized project.

85 Unreachable code: There were only 8 occurrences in four projects.

The connection of the implemented smells to the examined international standard can be seen in the Figures (5–6).

Zhang et al. [30] provided a systematic literature review on code smells

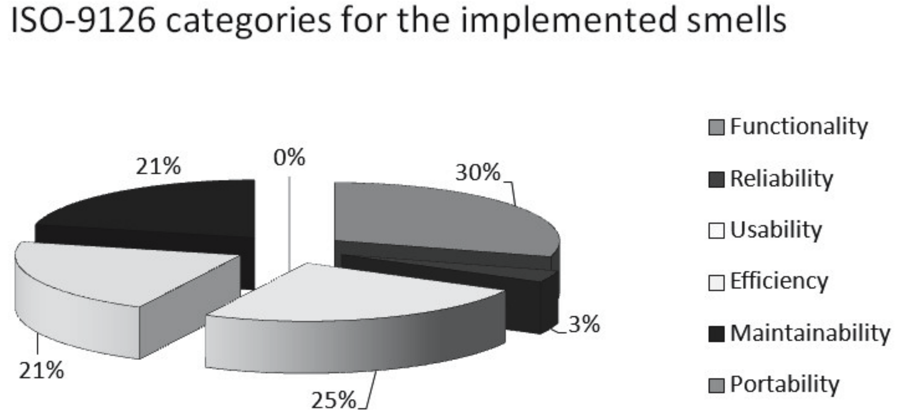

Figure 5: The implemented code smells categorized according to ISO/IEC 9126-1

and refactoring strategies based on papers published by IEEE and six leading software engineering journals from 2000 to June 2009. Nearly half of the identified papers (49%) described methods or tools to detect code smells, one-third focused on the interpretation of code smells, and 15% centered on refactoring. There were only a few study investigating the impact of code smells on maintenance or other quality attributes [21, 19, 18], but none of them were applicable to our test quality smells. We found an interesting machine learning approach for the calculation of thresholds for software metrics to evaluate quality attributes [9] but we decided to apply a pragmatic approach.

In order to have an impression about the usefulness of the previous smells we calculated the project risk factors in the usual way:

$$\mathrm{RiskFactor}(\mathrm{proj}) = \sum_{\mathrm{smell}} \mathrm{RelativeOccurrence}(\mathrm{proj}, \mathrm{smell}) \times \mathrm{Impact}(\mathrm{smell})\,.$$

For the impact estimation we used three classes:

1	– small impact,
2	– medium impact,
3	– large impact.

There were 4 smells classified into the large-impact class (with ordinal numbers from the smell enumeration): 12, 17, 18, 19; nine smells were classified into

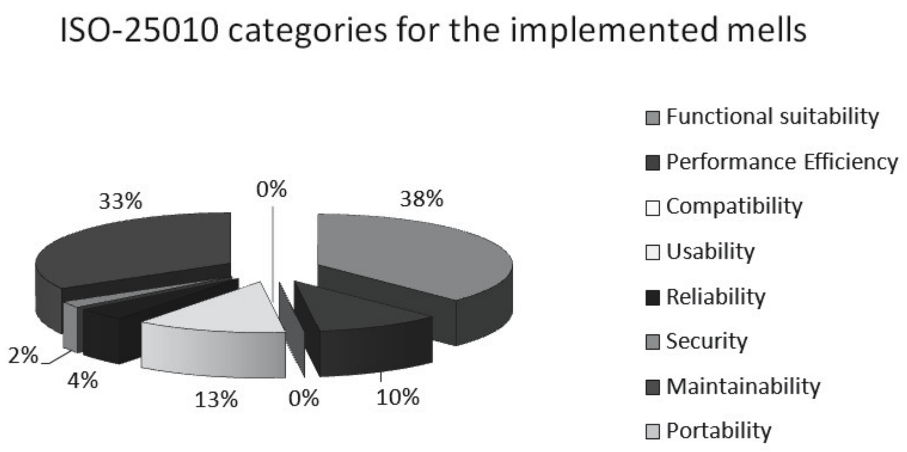

Figure 6: The implemented code smells categorized according to ISO/IEC 25010

the small impact class: $2, 3, 6, 13, 14, 20, 27, 29, 34$; all the others belonged to the medium impact category.

In order to determine the classification of the relative occurrences[15] of the smells we used *smell-baselines* on the measured data. For a smell S the smell-baseline $\mathsf{S}_\mathfrak{b}$ means that the smell S is acceptable to occur in every $\mathsf{S}_\mathfrak{b}$ effective lines of code in average. Then, we applied the following categories:

0	– no smell occurrence,
1	– rare occurrences ($\mathsf{S}_{\text{actual}} > \mathsf{S}_\mathfrak{b}$),
2	– occasional occurrences ($\mathsf{S}_\mathfrak{b} \geq \mathsf{S}_{\text{actual}} > \mathsf{S}_\mathfrak{b}/2$),
3	– likely occurrences ($\mathsf{S}_\mathfrak{b}/2 \geq \mathsf{S}_{\text{actual}} > \mathsf{S}_\mathfrak{b}/8$),
4	– frequent occurrences ($\mathsf{S}_\mathfrak{b}/8 \geq \mathsf{S}_{\text{actual}}$).

Here $\mathsf{S}_{\mathsf{actual}}$ means the actually measured relative occurrence in a given project.

Let see an example. Based on the ETSI projects the smell-baseline for the smell *MagicNumber* is 50. In project P with size 135845 eLOC the actual (measured) value was 5657 occurrences, i.e.,

$$\textit{MagicNumber}_{\text{actual}} = 135845/5657 = 24.$$

[15]Here relative occurrence means the size normalized occurrence.

Hence, this smell occurs more than twice often then the baseline, therefore

$$\text{RelativeOccurrence}(\mathsf{P},\ \mathit{MagicNumber}) = 3.$$

After calculating the relative occurrences for all smells in project P we are able to determine the risk factor of this project. Then, we can compute the quality level of the project P by

very high	if	$0 < \text{RiskFactor}(\mathsf{P}) <= \mathsf{T}$,
high	if	$\mathsf{T} < \text{RiskFactor}(\mathsf{P}) \leq 2\mathsf{T}$,
medium	if	$2\mathsf{T} < \text{RiskFactor}(\mathsf{P}) \leq 3\mathsf{T}$,
low	if	$3\mathsf{T} < \text{RiskFactor}(\mathsf{P}) \leq 4\mathsf{T}$,
very low		otherwise.

The threshold T was calculated as the average value of the risk factors times $2\mathsf{C}/5$, where the correction factor C is between 1 and 2. In our case C was 1.6.

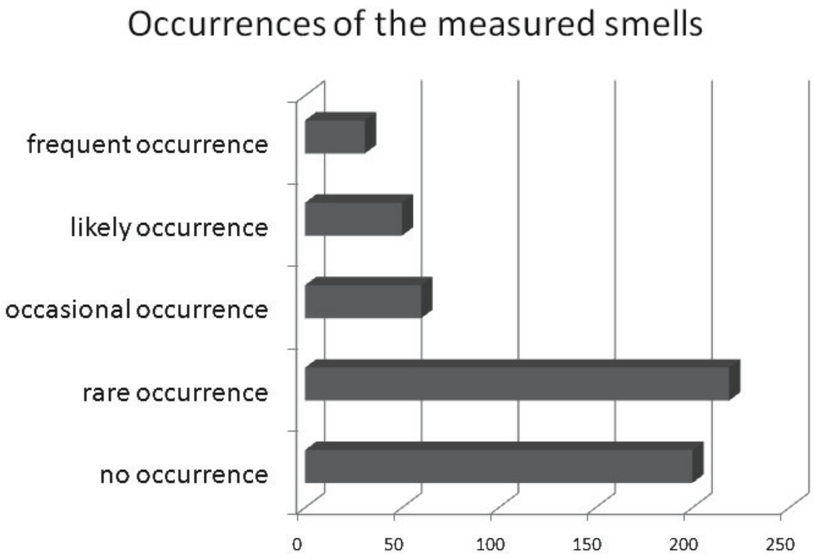

Figure 7: Relative occurrence distribution of the measured smells for all projects

The smell-baselines were determined on the basis of the 6 ETSI projects. We assumed further that the ETSI projects have good (or very good) quality, i.e., we forced them to fall into the high or very high quality category by adjusting the correction factor C. Figure 7 shows the relative occurrence distribution of the measured projects for all smells.

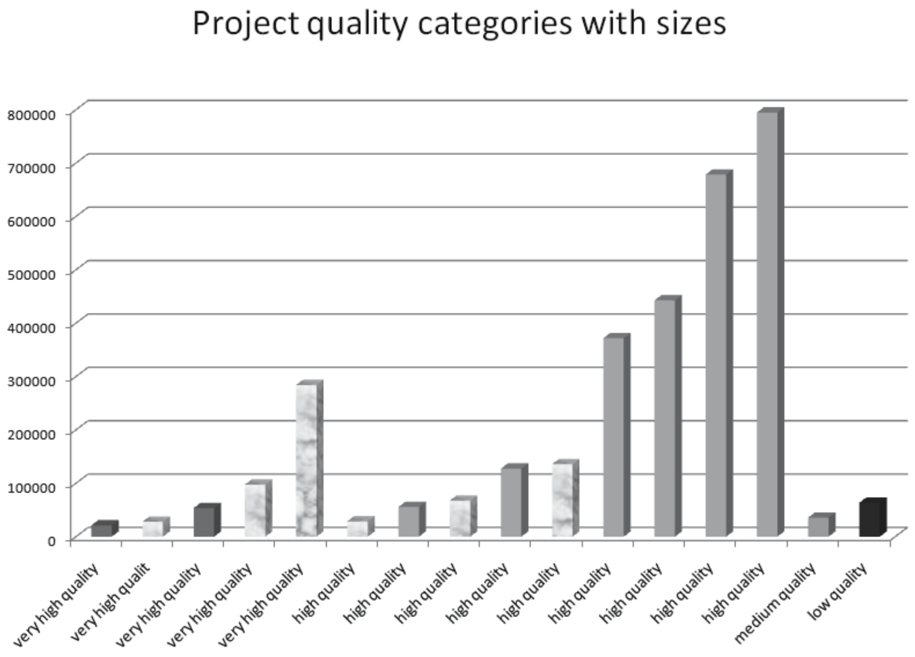

Figure 8: Project qualities with sizes

The average value of the risk factors were 67.75, hence $T = 43$. The quality categories of the projects can be seen in Figure 8. We note that we found no correlations between quality categories and project sizes, but there were no “big” project with very high quality. It must also be noted that altogether there were only one medium and one low quality project. The ETSI projects in Figure 8 are marked with marble filling.

Figure 9 and 10 show the most occurred (i.e. likely or frequently) code smell penetration for the low quality project according to the ISO 9126 and ISO 25010 models.

It would be an interesting continuation of this research to analyze the spreading of both the widely spread and the localized smells further. Local smells might be the result of misunderstanding some documentation or not sufficient knowledge transfer. It might also happen that some of the functionalities of the language are only used, by (a few) users. At the other end of the spectrum the

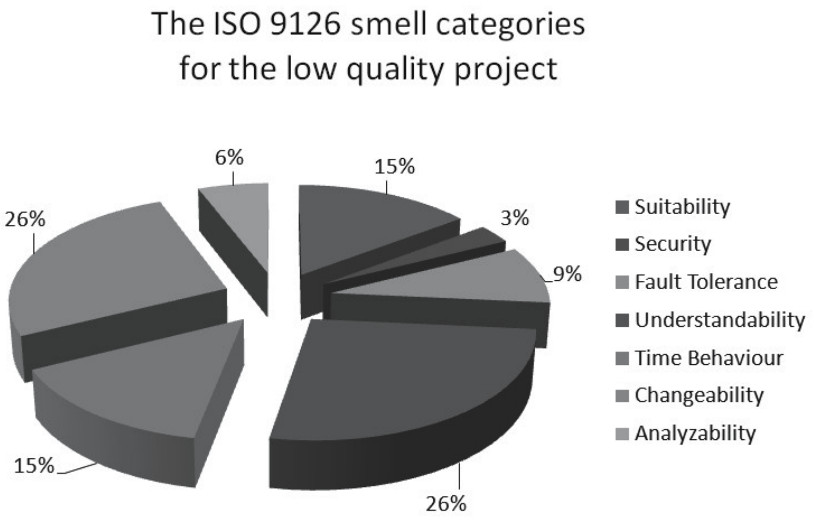

Figure 9: The most occurred code smells for the low quality project categorized according to ISO/IEC 9126-1

widely spread smells might be the results of misconceptions spreading among projects.

6 Summary

In this paper we examined the ISO/IEC 9126 and ISO/IEC 25010 software quality standards from the point of view of test software products. We analyzed how large scale test frameworks written in TTCN-3 can be adapted to these standards. We presented a list of code smells which were collected from PMD, FxCop, Checkstyle, FindBugs, TRex and code smells which were identified during reviews or trouble reports in actual industrial systems.

We conclude that the set of code smells we have shown overlaps with all main characteristics in the ISO/IEC 9126 quality standard. On the other hand they overlap all but the *compatibility* main characteristic of the ISO/IEC 25010 standard. We found that *compatibility* in the ISO/IEC 25010 meaning is not applicable in our situations. ISO/IEC 25010 defines compatibility as "degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while

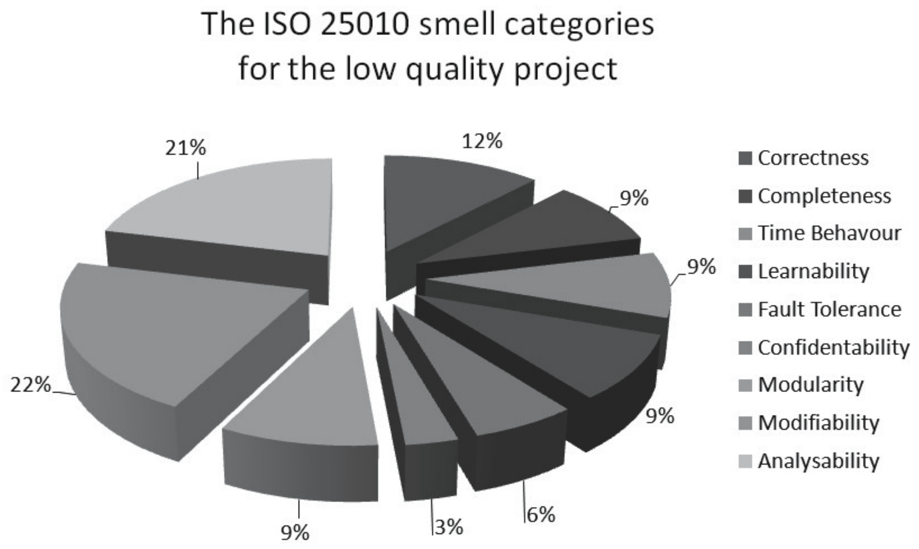

Figure 10: The most occurred code smells for the low quality project categorized according to ISO/IEC 25010

sharing the same hardware or software environment". The TTCN-3 language abstracts away all hardware and software environments standardizing the internal communications of the system.

It is interesting to see that the rules according to ISO/IEC 9126 are how well distributed. This is no longer true for ISO/IEC 25010 where *maintainability* takes up 38% of all rules while it was only 21% according to ISO/IEC 9126.

7 Further work

Our future plans involve the measurement of all of the code smells in actual projects to see how well we could *estimate* the quality of a TTCN-3 test software product. Later we would like to rise the abstraction level and include high level metrics and measure architectural quality. This way we could profile the quality of the project on several levels. We could then measure quality for designers, architects and managers. All of which are valid point of views but have different understanding and requirements. We also plan to measure runtime characteristics in order to increase our understanding of software product quality.

It is also in our plans to analyze the spreading of issues among projects. We would like to research why some of the code smells are localized to only a few projects, while others are present in almost every project. This might bring up very interesting questions related to ways of working, information distribution, or other higher level quality aspects of the projects.

8 Acknowledgments

This research would not have been possible without the help of the Quality Assurance Organization and the Software Technology (DUCI SWT) department of our industrial partner Ericsson. They were kind enough to allow us access to their databases and some of their TTCN-3 source codes and provide funding. These proved to be invaluable to find code smells that are of real life significance.

We would also like to thank Bernadett Diana Iván for her help in processing the fault and review databases. Her data mining knowledge allowed us to quickly extract and process large amount of data recorded in the span of 6 years. We would like to thank also Dániel Poroszkai for implementing many code smell metrics thoroughly.

References

[1] A. Bánsághi, B. G. Ézsiás, A. Kovács, A. Tátrai, *Source Code Scanners in Software Quality Management and Connections to International Standards*, *Annales Univ. Sci. Budapest. Sect. Comput.*, **37** (2012) 81–92. ⇒79, 80

[2] B. W. Boehm, Ph. N. Papaccio, Understanding and Controlling Software Costs, *IEEE Transactions on Software Engineering*, **14,** 10 (1988) 1462–1477. ⇒78

[3] *Checkstyle*, http://checkstyle.sourceforger.net ⇒84

[4] *CMMI institute*, http://cmmiinstitute.com/ ⇒78

[5] ETSI ES 201 873-1 v4.5.1: *Methods for Testing and Specification (MTS)*, The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. ⇒85

[6] *FindBugs*, http://findbugs.sourceforge.net ⇒84

[7] M. Fowler, *Refactoring, Improving the Design of Existing Code*, Addison-Wesley, 1999. ⇒84

[8] *FxCop*, http://msdn.microsoft.com ⇒80, 84

[9] S. Herbold, J. Grabovszki, S. Waack, *Calculation and Optimization of Thresholds for Sets of Software Metrics. Empirical Software Engineering*, Springer, Netherlands, May 2011. ⇒94

[10] W. S. Humphrey, *The Team Software Process*, (2000) Technical Report, CMU/SEI-2000-TR-023, ESC-TR-2000-023 ⇒78

[11] ISO/IEC 9126:1991, *ISO Standard for Software Engineering – Product Quality* Revised by ISO/IEC 9126–1:2001. ⇒78, 81

[12] ISO/IEC 9126-1:2001: *ISO Standard for Software Engineering – Product Quality – Part 1: Quality Model.* ⇒81

[13] ISO/IEC TR 9126-2:2003: *ISO Standard for Software Engineering – Product Quality – Part 2: External Metrics.* ⇒81

[14] ISO/IEC TR 9126-3:2003: *ISO Standard for Software Engineering – Product Quality – Part 3: Internal Metrics.* ⇒81

[15] ISO/IEC TR 9126-4:2004: *ISO Standard for Software Engineering – Product Quality – Part 4: Quality in Use Metrics.* ⇒81

[16] ISO/IEC 25010:2011: *ISO Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models.* ⇒78, 81

[17] S. H. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley, Boston, 2003. ⇒80

[18] F. Khomh, M. Di Penta, Y-G. Gueheneuc, An Exploratory Study of the Impact of Code Smells on Software Change-proneness, *Proc. 16th Working Conference on Reverse Engineering*, 2009, pp. 75–84. ⇒94

[19] W. Li, R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution, *Systems and Software* **80,** 7 (2007) 1120–1128. ⇒94

[20] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley Professional, 2007. ⇒84

[21] A. Monden, D. Nakae, T. Kamiya, S. Sato, K. Matsumoto, Software quality analysis by code clones in industrial legacy software, *Symposium on Software Metrics*, 2002, pp. 87–94. ⇒94

[22] H. Neukirchen, B. Zeiss, J. Grabovszki, An approach to quality engineering of TTCN-3 test specifications, *International Journal on Software Tools for Technology Transfer (STTT),* **10,** 4 (2008) 309–326. ⇒85

[23] H. Neukirchen, M. Bisanz, Utilising code smells to detect quality problems in TTCN-3 test suites, *Proc 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007)*, June 26-29, 2007, Tallinn, Estonia. *Lecture Notes in Computer Science (LNCS)* **4581,** 2007 pp. 228–243. ⇒ 85

[24] *PMD*, http://pmd.sourceforge.net ⇒80, 84

[25] R. van Solingen, E. Berghout, *The Goal/Question/Metric Method, a Practical Method for Quality Improvement of Software Development*, McGraw-Hill, 1999. ⇒78

[26] K. Szabados, Structural analysis of large TTCN-3 projects, *Proc. 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, *Lecture Notes in Computer Science* **5826**:, *Testing of* Software and Communication *Systems*, Springer-Verlag Berlin, Heidelberg, 2009 pp. 241–246. ⇒80

[27] *The Personal Software Process (PSP) Body of Knowledge*, Version 2.0; Special Report; CMU/SEI-2009-SR-018 ⇒78

[28] *TRex*, http://www.trex.informatik.uni-goettingen.de/trac ⇒84

[29] B. Zeiss, D. Vega, I. Schiferdecker, H. Neukirchen, J. Grabovszki, *Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications*, Software Engineering, *Lecture Notes in Informatics (LNI)* **105**, 2007, pp. 231–242. Gesellschaft für Informatik, Köllen Verlag, Bonn, ⇒79

[30] M. Zhang, T. Hall, N. Baddoo, Code Bad Smells: a review of current knowledge, *J. Softw. Maint. Evol.: Research and Practice* **23,** 3 (2011) 179–202. ⇒93

Received: April 10, 2013 • Revised: June 7, 2013