Interface-based software testing

Aziz Ahmad Rais

Department of Information Technologies
University of Economics, Prague
Prague, Czech Republic

aziz.ahmad.rais@gmail.com

DOI: 10.20470/jsi.v7i4.277

Abstract: *Software quality is determined by assessing the characteristics that specify how it should work, which are verified through testing. If it were possible to touch, see, or measure software, it would be easier to analyze and prove its quality. Unfortunately, software is an intangible asset, which makes testing complex. This is especially true when software quality is not a question of particular functions that can be tested through a graphical user interface.*

The primary objective of software architecture is to design quality of software through modeling and visualization. There are many methods and standards that define how to control and manage quality. However, many IT software development projects still fail due to the difficulties involved in measuring, controlling, and managing software quality.

Software quality failure factors are numerous. Examples include beginning to test software too late in the development process, or failing properly to understand, or design, the software architecture and the software component structure. The goal of this article is to provide an interface-based software testing technique that better measures software quality, automates software quality testing, encourages early testing, and increases the software's overall testability.

Key words: quality, software quality, software testing, and testing automation.

1. Introduction

Fifty-two percent of IT projects in 2015 had challenges, and nineteen percent failed or did not meet the criteria for success (Standish Group, 2015). Reasons according to Standish Group included late delivery, going over budget, and a failure to deliver required features. Rais (2016) identifies low software quality as one of the common reasons for IT project failure, based on an analysis of the last of the ten success/failure factors from the Standish reports of 2009 and 2010. *Mieritz (2012) looks at Gartner's analysis of why IT projects fail, and notes that eleven percent of IT projects deliver poor quality. According to ISO/IEC 25010 (2008), one of the characteristics of software product quality is functional suitability. In other words, delivering inappropriate, inaccurate, and non-compliant functionality can cause software development projects to fail. VersionOne (2015) provides Agile Initiative's success criteria, and finds that only forty-eight percent of projects delivered product quality, with just thirty-six percent delivering all the features or requirements.*

Most surveys show that software quality is a key factor in the success or failure of an IT project. But what then causes failure in software quality? In order to analyze this, it is first necessary to define software quality, and then to focus on how to control and test it.

2. Software quality

2.1 Quality

Quality can have many meanings, such as, for example, value for money, excellence, zero errors, disposition, essential nature, high grade or high status, essential and distinguishing attribute, degree or grade of excellence or worth, perfection and so on (Harvey, 2014-16). Other interpretations are that quality is conformance to specifications or standards (Juran, 1992); if a product is non-conformant, then it is also not fit for use, so quality can also be seen as "fitness-for-use" (Juran, 2003); fitness for purpose (Hoyle, 2009); and the degree to which a set of inherent characteristics fulfills requirements (ISO 9000, 2005).

These definitions are all valid in helping us examine software quality. However, because software is an intangible asset, we require further clarification in order better to understand and measure its quality. This means that we need a solution able to assess the **environments** where the software is run (its **external quality**), its **users** (its **quality in use**), and its internal processes, the way its own components relate and provide information to each other (its **internal quality**) (ISO/IEC 25000, 2005). Therefore, each software characteristic can be measured in relation to its environment. ISO/IEC 25000 (2005) defines quality as *the capability of a software product to satisfy stated and implied needs when used under specified conditions*. Evaluating software quality in relation to its environment, users, and inter-component processes can be done in two ways:

1. **Statically**: This usually involves reviewing the software product's source code, and mapping it to the architecture. Static analysis tools are sometimes used to check the code against style rules, for clean code, for mistakes, for detailed design, and so on. In other words, static measurement is, for the most part, related to internal quality. This testing method can be carried out with **static test techniques**.
2. **Dynamically**: This is for measuring external quality and quality in use, specifically when the software is deployed to its runtime environments. Runtime may differ according to the testing phase (unit test, integration test, system test, or user acceptance test). This testing method is carried out with **dynamic test techniques**.

Each software quality characteristic is thus tested differently, and each testing type, for example, performance testing, security testing, functional testing, usability testing, and so on, requires different techniques and runtime environments. In order to evaluate and measure software quality effectively, we should regard the software test as a complete process or method that needs to be planned and organized. As a part of this process, we have to design test cases, and define test strategies, from the very beginning of the IT software development project. To that end, we need a testing concept, a testing process, testing documentation, and testing techniques.

2.2 Software test concepts

A software test is one of the sub-processes of the software development life cycle (SDLC) (*ISO/IEC 12207, 2008*). During its life cycle, software goes through different phases that can affect its overall quality. Therefore, the testing process cannot be delayed until after the construction phase, but should begin as early as possible. One example of this kind of implementation is the technique of test-driven development (TDD) (Ambysoft Inc., 2013). Essentially, the TDD process is as follows:

1. Write a test (unit test).
2. Run your tests. Often, this will be the complete test suite, which will fail because the new feature has not yet been implemented.
3. Write or update your software source code (implement the new feature) to make the test pass.
    a. Check the design of your software source code, and refactor it to meet design criteria. Run your test again.

If the test still fails, you need to update the source code (refactor), and retest till the design meets certain criteria, and the test passes.

The primary issues with this technique are 1) there is no foolproof way of knowing that our tests are correct, and 2) design corrections are carried out after implementation. Both issues can necessitate many development cycles, and the more features we add, the longer refactoring lasts. Thus, in later software development iterations, refactoring will become time-consuming, and the project manager will have to make a choice between quality and cost. Although we can **automate testing** to support repeated runs, refactoring software source code can result in the automated test code also having to be updated. The TDD approach is, therefore, suitable only when we are unsure of our course of action, and are experimenting with trial and error, and in small software development projects.

Thus, although beginning the testing process earlier in the software development life cycle is a good idea, in practice, this requires a method other than TDD. Rather, a holistic approach is recommended so that the SDLC processes do not negatively affect each other.

Such a holistic approach to testing means we have to regard the software test as a process that must follow other processes at the **organization level**. Further, we also have to consider the processes (**test management processes**) that will manage the software test processes (**dynamic test**

processes) (ISO/IEC/IEEE 29119-2, 2013). Collectively, these three levels of process can be termed multilayer testing processes (ISO/IEC/IEEE 29119-2 2013).

The dynamic test processes are:

- The test design and implementation process
- The test environment set-up and maintenance process
- The test execution process
- The test incident reporting process

In order to design tests and implement a testing process in a software development project, we need an effective strategy for executing test cases. The test cases themselves have to be documented and prepared for every type of test, each of which is executed in a different testing phase. The goal of each such testing type is to verify different software qualities or characteristics. The relationship between all the concepts (e.g. the test plan, the test strategy, the test type, quality standards, etc.) described thus far is demonstrated in the following context:

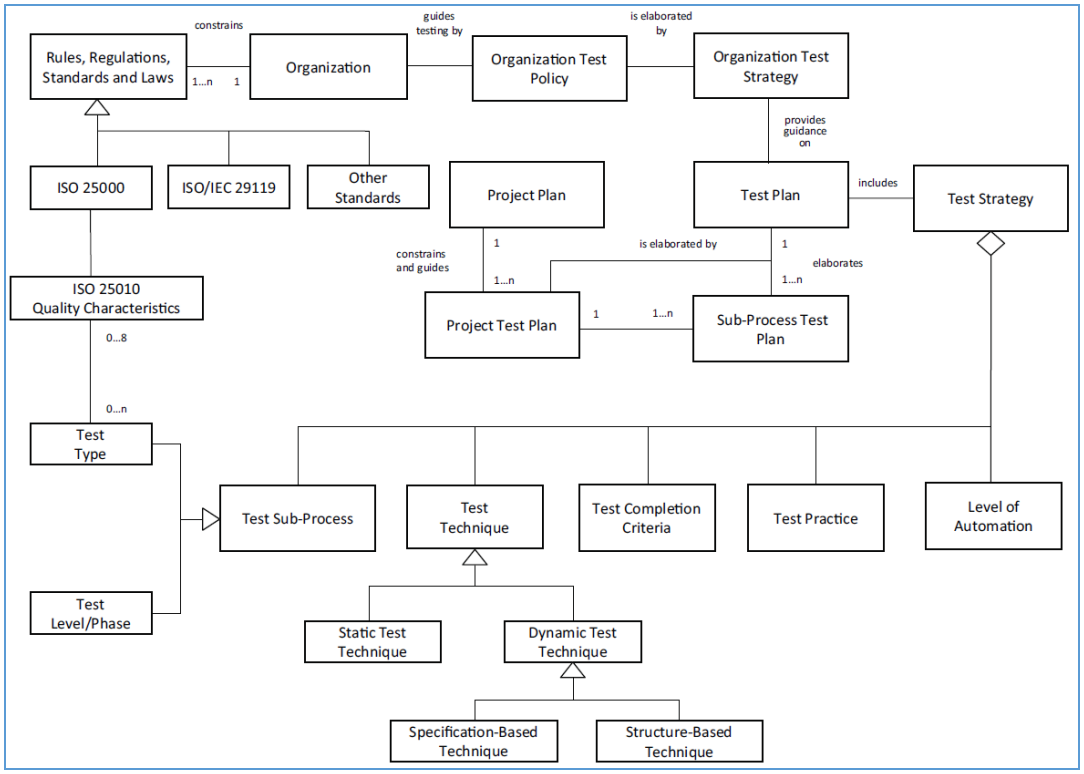

Figure 1: Software test concepts *(source: ISO/IEC/IEEE 29119-1, 2013)*

Maintaining software tests, making decisions based on their results, and accepting software quality all require test documentation. According to the testing layer in question, some documentation, for example, organization-level test documentation or test-management documentation will only need to be created once. Details for how to produce two-layer documentation can be found, for example, in the ISO/IEC/IEEE 29119-3 (2013) standard. Because test documentation can grow extremely lengthy if the software is large, it is important to use such techniques to minimize documentation volume, and facilitate maintenance.

ISO/IEC/IEEE 29119-4 (2015) defines three categories of dynamic test technique:

1 **Specification-based**: Such techniques deal with partitioning and classifying software functionalities into various testing items. These test different scenarios, inputs, relations and dependencies between different software functions, and transition states between different actions and events. Further, they carry out combination testing, randomly test functionalities, and so on. All these techniques are known as **black-box testing** (Galin, 2004).

2 **Structure-based**: These techniques are based on code structure, for example, testing branches in a control flow, decisions in a control flow, data flow, and so on. Testing according to code structure is called **white-box testing** (Galin, 2004).

3 **Experience-based**: These could be any technique such as, for example, guessing or predicting errors.

3. Interface-based software testing

The previous section, "*Software quality*," shows that high quality software depends on effective and successful testing, while successful testing in turn depends on utilizing the proper testing techniques. These techniques should not negatively affect other SDLC processes, but ought to support testing automation, and both parallel and iterative development. Internal software quality is also critical, and necessary for external quality and quality in use characteristics.

3.1 Problem definition

Once software requirements gathering and analysis is complete, most software development projects start with the architecture, followed by detailed design, and then construction. In other cases, either the detailed design or the architecture, or both, is skipped, and software developers begin with implementation. In both scenarios, dynamic test design, primarily white-box test design, can start during software construction. Dynamic test execution can then begin at the end of the software construction phase. Similarly, some black-box tests can also be designed based on the software requirements. However, not all types of black-box test can be designed at this stage because, in most software development projects, business requirements are not analyzed before software requirements specification. For business requirements gathering and specification, most software development methodologies utilize use cases, user stories, or other techniques reliant upon natural language. Natural languages are inherently indefinite (IEEE Std 830, 1998). As a result, software requirements specification becomes time consuming, and the cost of its maintenance increases the overall cost of the project. Thus, in many projects, software construction is carried out based on raw business requirements.

3.2 A definition of "interface"

Before diving into the interface-based test concept, it is important here to clarify, at least to some extent, the term "interface", which is used everywhere, and which sometimes causes confusion. The EOLD (2016) defines an interface as a *point where two systems, subjects, organizations, etc. meet and interact*. According to IIBA (2009), an interface is a connection between two components. Meanwhile, OMG (2015) describes an interface as *a kind of Classifier that represents a declaration of a set of public Features and obligations that together constitute a coherent service*. At the programming-language level, Java explains an interface as *a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface (Java tutorial, 2015)*.

The definitions provided by the OED and IIBA are similar, and can be extended to describe an interface as an interaction, or connection, **point** between two **entities** (systems, sub-systems, application software, components, software components, organizations, classes, objects, etc.). The OMG definition describes that interaction point (*public Features and obligations that together constitute a coherent service*). At the programming-language level, the expression of an interaction point is a contract between a class and the outside world.

The confusion surrounding the term "interface" is mostly due to the fact that there are different types of interface, for example, a graphical user interface (the interaction point between a human user and the software), a command-line interface, an interface between two classes (how one class instance uses the service of another class, or what service one class provides to another), and so on. This confusion between types can, at times, be an advantage. For example, the interface between two organizations might need to be automated through the use of a software service, so if we specify and define it once then we can reuse it at the software-component or class level without exerting a big effort, or carrying out a large-scale transformation. The term "interface" is a universal concept that can describe a real-world service, and is represented in many object-oriented programming languages. This means that the entity that implements the interface only has the specifications of *what* needs to be implemented, but not *how* it ought to be implemented.

Another issue with the usage of the word “interface,” is its representation, meaning how an interface is concretely described. For example, in object-oriented programming languages like Java, an interface can be represented by a **key-word interface**, which is compiled to a class with a method declaration without implementation. In a unified modeling language (UML), meanwhile, it can be represented by classes, components, objects, or interface classifiers.

3.3 The concept

The concept behind interface-based testing is not to delay its design and execution until all the classes that implement the business functionalities are ready. All software provides some type of service to different consumers. Therefore, we have to differentiate between business functionalities, and all the other functionalities that support the services provided by the software.

For example, **client registration requirement**: the business requirement is that the software must provide functionality that obliges clients to register before they can request another service, such as buying goods or adding items to a shopping basket. Thus, a client registration service can be specified by one interface, while the client accesses this service via another, a graphical user interface (GUI). Then, once data gathering and manipulation is complete, the client registration service has to store the client’s data in a repository.

The interface that specifies the client registration service is the **business requirements interface**. Storing the data, accessing the client registration service via the GUI, and implementing the different types of validation, however, are carried out by **technical functionality or technical design interfaces**.

After separating functionalities in this manner, it becomes very easy to design interfaces for the services to be provided by the software. All these business service interfaces can be grouped into service components, while all the technical functionalities can be grouped as separate components. Identifying their interfaces also becomes simple when based on the service interfaces. Because component can have provider interfaces and consumer interfaces (OMG, 2015), therefore the technical interfaces will become the required interfaces of the business service interfaces.

After specifying a software service’s interfaces, the software tester can write test cases and implement, for example, unit tests, or automated dynamic tests. The software designer can identify all the technical interfaces, design the software, and then implement those interfaces. The fact is that classes can be changed till they work properly, but changing an interface means changing the business requirements, which can only be done in agreement with the business stakeholder. Interfaces can easily be modeled using a UML and unit tests and documentation can be generated from the resulting models. The concept of the interface-based test process can be modeled in the following manner:

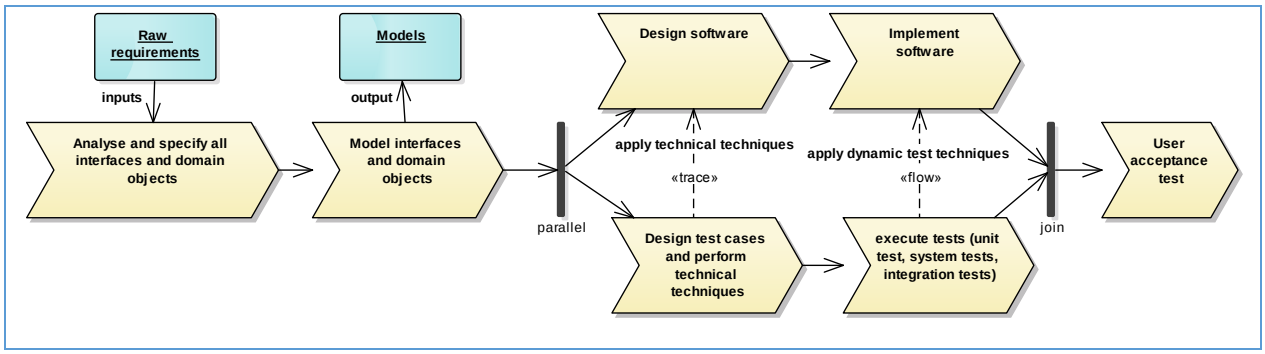

Figure 2: interface-base software test process *(source: author)*

3.4 Example

The UML tools Papyrus[1], Eclipse Mars[2], and the Acceleo plugin[3] for Eclipse can be used to demonstrate the concept. The example below shows an analysis of the interfaces and domain objects

[1] *http://www.eclipse.org/papyrus/download.html*
[2] *https://eclipse.org/mars/*

for the client registration requirement. This is a modeled UML class diagram that can, using Acceleo, be transformed into the source code of a programming language, for example, Java. The Java source code and the model can be used to enable a software test team, a software tester and a software designer, and, in some cases, a software developer to work in parallel. The interface is thus established as a kind of contract between them. The goal of using this model, and the Acceleo templates, is to show both the concept's inherent simplicity, and the reusability of the work (analysis).

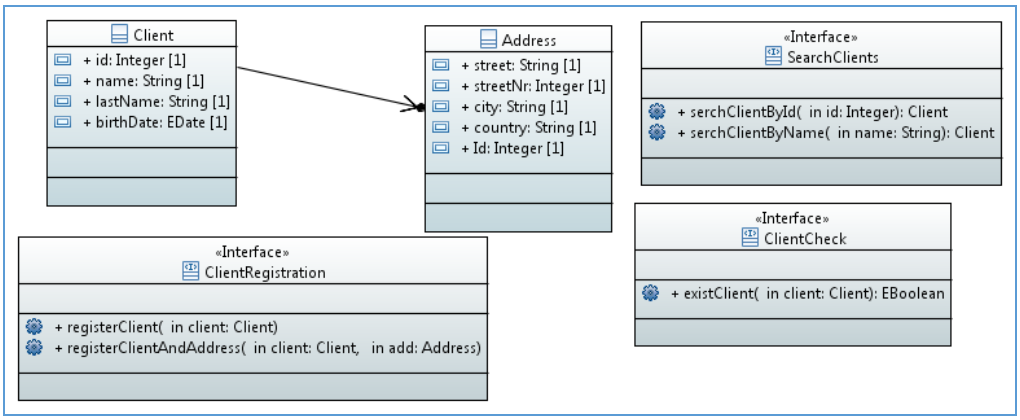

Figure 3: An analytical model of client registration requirements *(source: author)*

The code below shows the Acceleo templates for generating the Java source code. Acceleo is the implementation of the OMG MOF Model to Text Transformation Language (OMG, 2008). There are three templates bellow written in OMG MOF Model to Text Transformation Language, the interface to generate the interface, template for generating the domain objects required by interfaces operations and template for generating unit test skeletons.

```
[comment templates for generating interfaces /]
[module genInterface('http://www.eclipse.org/uml2/5.0.0/UML')/]

[template public generateInterface(i : Interface)]
[file (i.name+'.java', false, 'UTF-8')]
public interface [i.name.toUpperFirst()/] {
[for (o : Operation | i.ownedOperation)]
    public [o.returnType()/]([o.getInParameter()/]);
[/for]
}
[/file]
[/template]
[template public getInParameter(o : Operation)]
[for (p : Parameter | o.ownedParameter) separator(',') ? (p.direction <> ParameterDirectionKind::return)][p.type.name/]
[p.name/][/for]
[/template]
[template public returnType(o : Operation)]
[if (o.upper = -1 or o.upper > 1)]
      [if (o.type.oclIsUndefined())]void [o.name/][else]List<[o.type.name/]> [o.name/][/if]
[else]
   [if (o.type.oclIsUndefined())]void [o.name/][else][o.type.name/] [o.name/][/if]
[/if]
[/template]
[comment templates for generating domain object models /]
[module genBeans('http://www.eclipse.org/uml2/5.0.0/UML')/]
[template public generateClass(c : Class)]
[file (c.name+'.java', false, 'UTF-8')]

public class [c.name.toUpperFirst()/]
      [for (sc : Class | c.superClass) before(' extends ') separator(',')] [sc.name/][/for] {

[for (p : Property | c.getAllAttributes())]
  [collectionOrNonProperty(p)/]
[/for]
[for (p : Property | c.getAllAttributes())]
    public [collectionOrNonType(p)/] get[p.name.toUpperFirst()/]() {
        return this.[p.name/];
```

[3] *https://eclipse.org/acceleo/*

```
    }
    public void set[p.name.toUpperFirst()/]([collectionOrNonType(p)/] a_[p.name/]) {
        this.[p.name/] = a_[p.name/];
    }
[/for]
[for (p : Property | c.getAssociations().memberEnd)]
[collectionOrNonProperty(p)/]
[/for]
}
[/file]
[/template]
[template public collectionOrNonType(p : Property)]
[if (p.upper = -1 or p.upper > 1)]List<[p.type.name/]>[else][p.type.name/][/if]
[/template]
[template public collectionOrNonProperty(p : Property)]
[if (p.upper = -1 or p.upper > 1)]
    private List<[p.type.name/]> [p.name/];
[else]
    private [p.type.name/] [p.name/];
[/if]
[/template]
[comment templates for generating test cases /]
[module genTests('http://www.eclipse.org/uml2/5.0.0/UML')/]
[import genInterface /]
[template public generateTestLoc( intf:Interface )]
[file (intf.name+'Test.java', false, 'UTF-8')]
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class [intf.name.toUpperFirst() /]Test extends TestCase {
      private [intf.name.toUpperFirst() /] [intf.name.toLowerFirst() /];
    @Before
    public void setUp() {
        //TODO instantiate interface [intf.name.toLowerFirst() /].
    }
    [for (op:Operation | intf.getAllOperations())]
    [generateMethod(op) /]
    [/for]
}
[/file]
[/template]
[template public generateMethod(op:Operation )]
      @Test
      public void [op.name/]Test ()
      {
      // TODO add test code of method [op.name/] ([getInParameter (op)/]) of the interface [op.interface.name/].
      }
[/template]
```

4. Conclusion

This article has endeavored to demonstrate the importance of software quality, highlighting what exactly is meant by quality in this context, and giving an overview of software testing. It has offered an analysis of test-driven development because particular testing techniques are not alone in affecting the software development life cycle; various methods must also be taken into account. Further, in order to develop a new software testing concept, “interface” is defined alongside the fundamental aspects of an interface-based testing concept. The examples provided demonstrate the ease-of-use of such a concept, and show that there are many open-source tools that can be used to develop it.

Further goals of this article have been to show the value of:

- An interface-based concept that involves testing software from the beginning of its development (see Figure 2). This concept supports all types of quality measurement, (internal quality, external quality, and quality in use).
- Measuring software quality: an interface can describe real-world services, and can have an exact representation in many object-oriented programming languages without imposing

limitations on either the software developer or the software designer. The assumption here is to have a choice between whether to generate *all* types of source code, or only to generate what is necessary and then to develop the remainder. By using an interface, we can easily calculate how many business functionalities were requested and how many have been delivered. Interface specifications also make it easy to measure interaction with the service provided by the software in question.

- Automating software quality testing: with an interface, we can prompt software development clearly to define a service layer and its own corresponding interfaces, which also improves integration testing. This kind of layer can be automated with less effort.
- Increasing software testability: the necessity of repeating tasks is one of the main reasons why software quality checks are sometimes skipped. Because an interface is a contract between the business and the software developer, keeping a clear separation between the functional requirements interface and the technical interface simplifies and improves software maintenance.

References

Ambysoft Inc., 2013: *Introduction to Test Driven Development (TDD)*. [Online] Available at: http://agiledata.org/essays/tdd.html [Accessed 27 September 2016]

GALIN Daniel, 2004: *Software Quality Assurance: From theory to implementation.* Addison Wesley. ISBN 0201 70945 7

Harvey, L., 2004: *Analytic Quality Glossary*. Quality Research International. [Online] Available at: http://www.qualityresearchinternational.com/glossary/ [Accessed 27 September 2016]

Hoyle D., 2009: *ISO 9000 Quality Systems Handbook*. Sixth edition. Elsevier. ISBN: 978-1-85617-684-2

IEEE Std 830, 1998: IEEE Recommended Practice for Software Requirements Specifications. ISBN 0-7381-0332-2

International Institute of Business Analysis (IIBA), 2009: *A Guide to the Business Analysis Body of Knowledge (BABOK Guide).* Version 2.0. ISBN-13: 978-0-9811292-2-8

International Organization for Standardization (ISO 9000), 2005: *Quality management systems — Fundamentals and vocabulary*. Third edition. ISO 9000:2005(E)

International Organization for Standardization (*ISO/IEC 12207), 2008: System and software engineering – Software life cycle processes.*

International Organization for Standardization (ISO/IEC 25000*),* 2005: *Software Engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Guide to SQuaRE. ISO/IEC JTC1/SC7 N3163*

International Organization for Standardization (ISO/IEC/IEEE 29119-1), 2013: *Software and systems engineering —Software testing — Part 1: Concepts and definitions*. ISO/IEC/IEEE 29119-1:2013(E)

International Organization for Standardization (ISO/IEC/IEEE 29119-2), 2013: *Software and systems engineering —Software testing — Part 2: Test processes*. ISO/IEC/IEEE 29119-2:2013(E)

International Organization for Standardization (ISO/IEC/IEEE 29119-3), 2013: *Software and systems engineering —Software testing — Part 3: Test documentation.* ISO/IEC/IEEE 29119-3:2013(E)

International Organization for Standardization (ISO/IEC/IEEE 29119-4), 2015: *Software and systems engineering —Software testing — Part 4: Test techniques*. ISO/IEC/IEEE 29119-4:2015(E)

International Organization for Standardization, *ISO/IEC 25010, 2008: Software engineering –Software product Quality Requirements and Evaluation (SQuaRE) –Software and quality in use models. ISO/IEC JTC1/SC7 N4098*

Juran J. M., 2003. *Juran on Leadership For Quality*. [Online] Available at: https://books.google.com/books?isbn=0743255771 [Accessed 27 September 2016]

Juran, J. M., 1992: *Juran on Quality by Design: The New Steps for Planning Quality Into Goods and Services*. [Online]. Available at: https://books.google.com/books?isbn=0029166837 [Accessed 27 September 2016]

Mieritz L., 2012: *Gartner Survey Shows Why Projects Fail. ID: G00231952.* [Online] Available at: https://thisiswhatgoodlookslike.com/2012/06/10/gartner-survey-shows-why-projects-fail/ [Accessed 26 September 2016]

English Oxford living Dictionaries (EOLD), 2016: *Definition of interface in English*. [Online]. Available at: https://en.oxforddictionaries.com/definition/interface [Accessed 27 September 2016]

Object Management Group (OMG), 2008: *MOF Model to Text Transformation Language, v1.0*. *[Online], Available at:* http://www.omg.org/spec/MOFM2T/1.0/PDF *[*Accessed 28 September 2016*]*

Object Management Group (OMG), 2015: *OMG Unified Modeling Language*. Version 2.5. formal/2015-03-01. [Online] Available at: http://www.omg.org/spec/UML/2.5 [Accessed 28 September 2016]

Oracle (Java tutorial), 2015. Object-Oriented Programming Concepts. [Online] Available at: https://docs.oracle.com/javase/tutorial/java/concepts/ *[*Accessed 28 September 2016*]*

Rais Aziz Ahmad, 2016: Interface-based software integration. *Journal of Systems Integration,* 7(3), pp 79 – 88, DOI: 10.20470/jsi.v7i3.261

Standish Group, 2015. *CHAOS Report 2015 [online], Available at:* http://www.infoq.com/articles/standish-chaos-2015 [Accessed 26 September 2016]

VersionOne, 2015. The 10th annual state of agile report. [Online]. Available at: https://VersionOne.com/pdf/VersionOne-10th-Annual-State-of-Agile-Report.pdf [Accessed 26 September 2016]

JEL Classification: C88