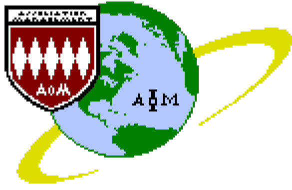


Journal of Information Technology Management

ISSN #1042-1319

A Publication of the Association of Management

THE UNIT TEST: FACING CICD – ARE THEY ELUSIVE DEFINITIONS?

HADAS CHASIDIM
SCE — SHAMOON COLLEGE OF ENGINEERING
hadasch@ac.sce.ac.il

DANI ALMOG
SCE — SHAMOON COLLEGE OF ENGINEERING
almog.dani@gmail.com

DOV BENYOMIN SOHACHESKI
SCE — SHAMOON COLLEGE OF ENGINEERING
b@kloud.email

MARK L. GILLENSON
UNIVERSITY OF MEMPHIS
mgillnsn@memphis.edu

ROBIN POSTON
UNIVERSITY OF MEMPHIS
rposton@memphis.edu

SHLOMO MARK
SCE — SHAMOON COLLEGE OF ENGINEERING
marks@sce.ac.il

ABSTRACT

The widely used term "unit test," which is a quality procedure for testing modules of software code, has been used for decades. To enable unit testing, specialized infrastructures were designed to enable the isolation of unit tested code from its production implementation. Currently, hundreds of tools and add-ons exist for unit testing in almost every software development environment and language. Integration testing, on the other hand, which is a quality procedure for testing multiple modules of software code working together, demands a vastly different kind of infrastructure to enable the interaction of the code with other production implementation components. With the growing popularity of agile methods, the boundaries of unit and integration testing have become blurred with Continuous Integration (CI) and Continuous Deployment (CD) derived from market demand for continuous reply and reaction of the software to a rapidly changing world. CICD enhances the importance of unit testing. Following that, it is important to identify the effect of new agile software development life cycles (SDLC) on unit test activities – assuming it may affect software quality in general. In this paper, we analyze the evolving different definitions and usages of the term "unit test" and attempt to understand the implication of these definitions to the actual use of the term.

Keywords: unit test, component test, unit under test

INTRODUCTION

Motivation

Continuous Integration (CI) and Continuous Deployment (CD) [10] are new trends driven by market demand. This trend is affecting the ability to provide an earlier and conscientious delivery of adaptation and changes to the software product. These new working paradigms signify a different approach towards software development life cycles (SDLC) and the need to understand the impact of this change on software quality and testing procedures. The effect of the change may be very significant at all dimensions of the organization and software development processes, with an examination of its effect on unit testing being the first step. It is essential to ensure a common language among SDLC team members using the term "unit test" consistently. To model the CICD practices of different software industries [16], the authors consider unit testing to be the basic procedure within the CICD approach. Following a review of the literature on CICD, research on CICD reports the consistent and meaningful use of unit test during the CICD process. Building on this work, this research explores the different definitions and usage of the term "unit test" during the last ten years as reflected in the academic literature.

The goal of this research study is to arrive at a better understanding of the term “unit test” and its applications. We explore the primary usages of the term to see whether there are differences of opinion among leading academic researchers. To our knowledge, this is one of the first papers that aims to better understand the influence of current changes in software development approaches and philosophies on testing efforts. This is thus an initial step in trying to arrive at a clearer understanding of the intention behind the value of unit testing.

First, we report the outcome of literature review research done on academic papers. The next section contains our definitions and suggestions to refine the definition and use unit testing in today’s more agile SDLC processes. The last section concludes our findings and gives recommendations for our future research.

Unit Testing in General

Today, almost every programming language has its individual unit testing framework (e.g., JUnit for Java, NUnit for C#), which enables the use of small, automatically executable unit tests. Unit testing has become an accepted practice, often mandated by development processes (e.g., test-driven development). On the other hand, many people use the term “unit test” with different connotations and meanings referring to the test of a specific component or functionality.

It is necessary to differentiate between the “unit under testing” and “unit test.” While **unit testing** is the act of testing at the unit of software code, e.g., one module, level, the term **unit under test** refers to the various portions of software code that are being tested. The first formal definitions of “unit testing” supported by the ANSI/IEEE Std 1008-1987, IEEE Standard for Software Unit Testing reveal some flexibility concerning the meaning of the word “unit” in “unit testing,” see insert below. This definition is not precise enough for today’s various approaches to SDLC, especially within the agile SDLC environments. For example, is a unit a single item or a set of items? Moreover, the definition does not provide us with a definite idea of the actual intention of the unit definition.

A set of one or more computer program modules together with associated control data (for example, tables), usage procedures, and operating procedures that satisfy the following conditions:

1. All modules are from a single computer program.
2. At least one of the new or changed modules in the set has not completed the unit test.
3. The set of modules together with its associated data and procedures is the sole object of a testing process.

Over the years, academics, open-source participants, and the software industry have generated an abundance of different definitions for unit tests [11]. While some definitions are products of necessity, others reflect a given principle. The majority of definitions cite principle buzzwords and concepts, attempting to remain accurate and authentic concerning the absolute definition of “unit.” Such as the case where unit tests cover only functional requirements or when a unit is defined as *writing many tests for one unit* without actually formally defining a unit.

The adoption of the use of xUnit tools in practice is a rather straightforward process, and the administration of tests involving the use of these tools is also relatively easy. The main drawback is the misconception that such tests are categorized as unit tests. With the help of familiar testing tools, unit testing has become a chameleon that can almost imperceptibly camouflage itself and transform itself into other types of testing. As Andrew Hunter has aptly noted, “Unit tests have quickly become the proverbial hammer that makes everything look like a nail” [7].

When codebases were still relatively small, and implementations were more transparent, the notion of a

unit was straightforward. Original software systems delivered anywhere from a single service to a series of services and behavior patterns. The behavior patterns were modest, and they focused on a specific goal. The original (still true to this the day) philosophy behind the Unix operating system (OS) is DOTADIW: "Do One Thing and Do It Well" [13]. The entire OS was a scaffold, using "very narrow, very tightly specified interfaces" [7]. Unix and Interlisp quickly gained popularity due to their separation of responsibilities through the use of pipes, filters, and interfaces [7].

The unit can be complex in its capabilities, but it is intended to be solely focused on a unique output. This characteristic transforms the unit into a *black box*; when it is provided with input, the unit will not deviate from its established, predetermined goal.

Unit testing is the sort of testing that is usually close to the heart of developers [18]. The primary reason is that, by definition, unit testing tests distinct, well-defined parts of the system in isolation from other parts. Thus, they are comparatively easy to write and use. Many build systems that have built-in support for unit tests, which can be leveraged without undue difficulty. With Maven [9], for example, there is a convention that describes how to write tests such that the build system can find them, execute them, and finally prepare a report of the outcome. Writing tests boils down to writing test methods, which are tagged with source code annotations to mark the methods as being tests. If the test code starts to require complicated setup and runtime dependencies, we are no longer dealing with unit tests. Here, the difference between unit testing and functional testing, which requires integration testing, can be a source of confusion. Often, the same underlying technologies and libraries are reused between unit and functional testing. This is a good thing, as reuse is good in general and lets you benefit from your expertise in one area as you work on another. Still, it can be confusing at times, and it pays to raise your eyes now and then to see that you are doing the right thing.

The testability of such units is optimal and easily implemented. Units are exclusively responsible and are independent of other units. Planning and designing systems composed solely of such units would be ideal but the possibility of the development of such systems amounts to wishful thinking. Today's systems are so large and intricate that the dismantling of a functional group into single units would be an exhausting process at best. The majority of systems today consist of compound or interdependent units. The interdependency of units degenerates the testing scheme and its capabilities: Instead of single regulatory outcomes from one unit, we find ourselves imitating input from various units to conclude about a given unit. Developers find themselves uncovering the encapsulation of units and their dependencies to perform unit tests, a task that is tantamount to *unit integration testing*, a topic that is beyond the purview of this paper.

Previous Surveys on Unit Tests

In 2006, a unit testing practices survey [14] was performed based on focus group discussions in a software process improvement network (SPIN) and a questionnaire was employed to validate the results (shown in Table 1). The purpose of the survey was to investigate what practitioners refer to when they talk about unit testing.

Since this survey has been conducted, more than 11 years ago, it seems logical to reexamine the use of its terminology and practices. Programming languages, as well as design implementations, have changed due to more complex software capabilities. A more recent survey 2014 [5] focused on the possible test automation benefits derived from the unit test:

1. What motivates developers to write unit tests? The driving force behind unit testing are the developers' conviction and management requirements
2. What are the dominating activities in unit testing? Writing new tests are perceived as less dominant than writing, refactoring, and fixing the code. Often, a failing test s treated with an amendment of the test (rather than the code) or a deletion of the test.
3. How do developers write unit tests? Developers claim to write unit tests systematically and to measure code coverage, but do not have a clear priority as to what makes an individual test good.
4. How do developers use automated unit test generation? The main uses of automated test generation are those that do not require any type of specification.
5. How could unit testing be improved? Developers do not seem to enjoy writing tests; they want more tool support in order to identify what to test and how to produce robust tests.

Our survey, on the other hand, is focused on the definition of the term "unit test" and on the ability to differentiate between the various units participating in unit testing. Naturally, we have chosen to explore the option of academic publication.

Table 1: Results of Unit Testing Practices Survey 2006 [14]

	Definition	Strength	Problem
What?	Test of smallest unit or units	Unit identification Test of surrounding modules	GUI test Unit identification Test scripts and harness maintenance Data structures
How?	Structure-based Preferably automated	Test framework Documentation	Framework tailoring Test selection Test metrics
Where?	Solution domain	Not found	Not found
Who?	By developer	Independent test competence network	competency independence introduction strategy
When?	Quick feedback	Continuous regression test	Stopping criteria
Why?	Ensure functionality	External requirement (safety) agile methods	Cost versus value

UNIT TEST DEFINITION SURVEY

Our survey intends to look for most of the publications that have appeared in major journals and at major academic conferences from 2002 and later (see Appendix 1). We omitted repetitive papers and excluded instructional books. Overall 112 papers were selected for this survey, all of which are listed in Appendix 1.

The following two selection criteria were employed:

- The term "unit test" appears in the paper's title and abstract.
- The term "unit test" is used more than ten times in the paper, the assumption being that anyone who uses a term that frequently must have a specific definition in mind.

Survey Questions

To make sure our collection was representative, most of the papers we chose were published in the last eight years. The distribution of the reviewed papers is shown in Figure 1. The data gathered from each paper included answers to the following questions:

1. What categories and affiliations in unit test definitions can we identify?
2. In addition to the formal definition within the paper, what attributes of usage can be found in the employment of the term?
3. Is there a connection between the main topic of the paper and its usage of the term "unit test"?

A textual and formal semantic analysis of the papers convinced us that we needed to spend more time reviewing each paper to try to understand the intention of the precise usage of terms. To answer these questions, we had to understand and evaluate each of these papers.

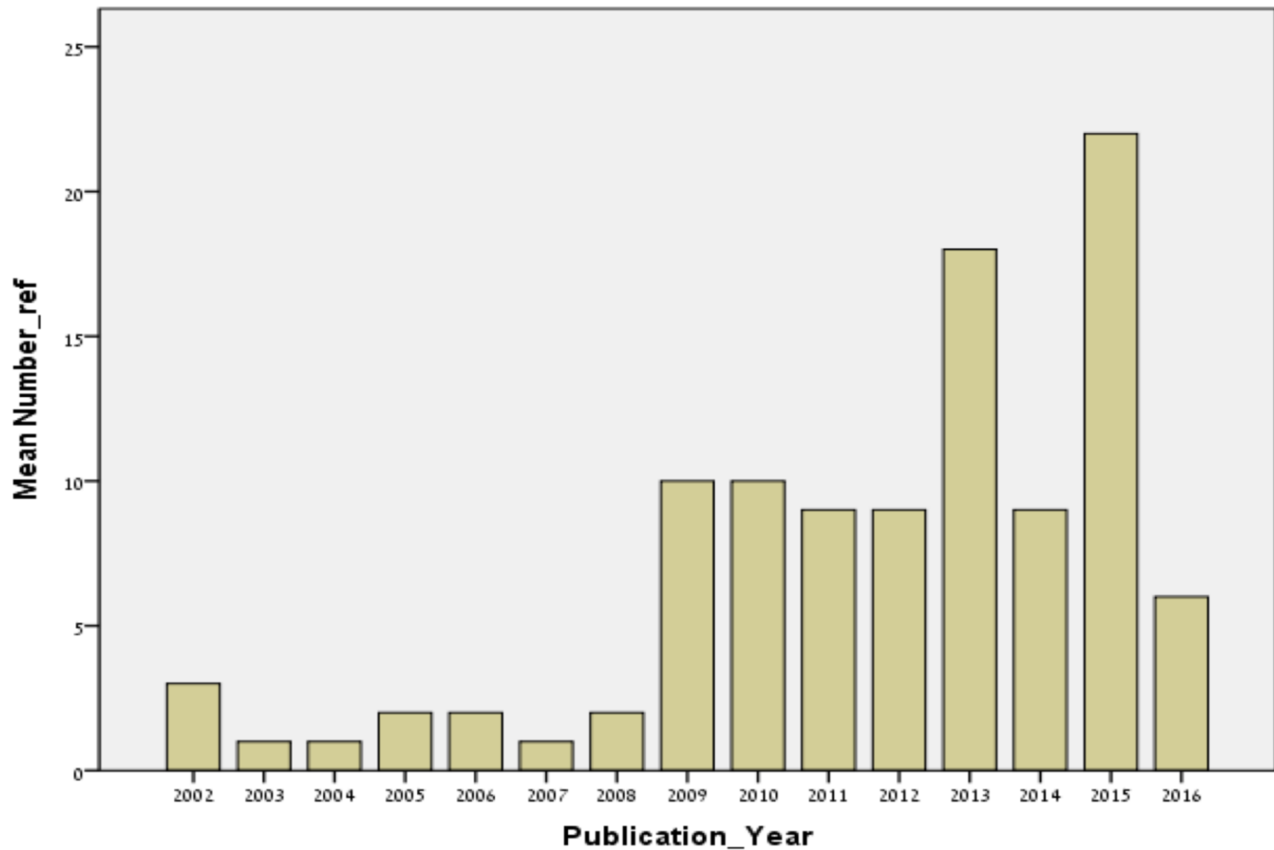


Figure 1: Articles Referred to by Publication Year

Categories

In our analysis of the content of the definitions, we identified the following categories of motives for defining unit test:

- Atomic – The definition states that the unit test case focuses on the smallest indivisible (atomic) fraction of code in the codebase.
- Isolation – The definition states that the unit test is being administered for code that has been isolated from the rest of the program.
- Code-related – The definition relates the unit to a specific code.
- External dependency – The definition states that there is a need to externalize all dependencies.
- Who is doing the test? – Who is performing the unit test (e.g., programmer)?
- Environment – The definition includes the characteristics of the required test environment.
- Methodology – The definition focuses on the methodology and technique used for the unit test.
- Automation – There is a direct relationship with test automation.
- Domain – A specific domain is included in the definition.
- Contribution – The definition contains a declaration of the value of the unit test.

A one-sample t-test was conducted to examine the differences between the definitions. Results showed that there were significant differences between all the definitions except for domain, environment, external dependency, and automation. See results in Table 2.

We concluded from our analysis that many of the definitions might belong to more than one category. Figure 2 presents the most common category affiliations found in our survey.

Table 2: One Sample T-test report

Category	(M, SD)	t(79)	$p < 0.05$
Code	M=1.5, SD=6.68	2.019	Yes
Atomic	M=1.3, SD=5.78	2.02	Yes
Isolation	M=0.83, SD=3.7	2.0	Yes
Who is doing test	M=0.53, SD=2.4	1.96	Yes
Contribution	M=0.45, SD=2.2	1.96	Yes
Methodology	M=0.45, SD=2.2	1.97	Yes
Environment	M=0.23, SD=1.04	1.94	Yes
External dependency	M=0.13, SD=0.64	1.75	Not significant
Automation	M=0.13, SD=0.6	1.9	Not significant
Domain	M=0.03, SD=1.6	1.43	Not significant

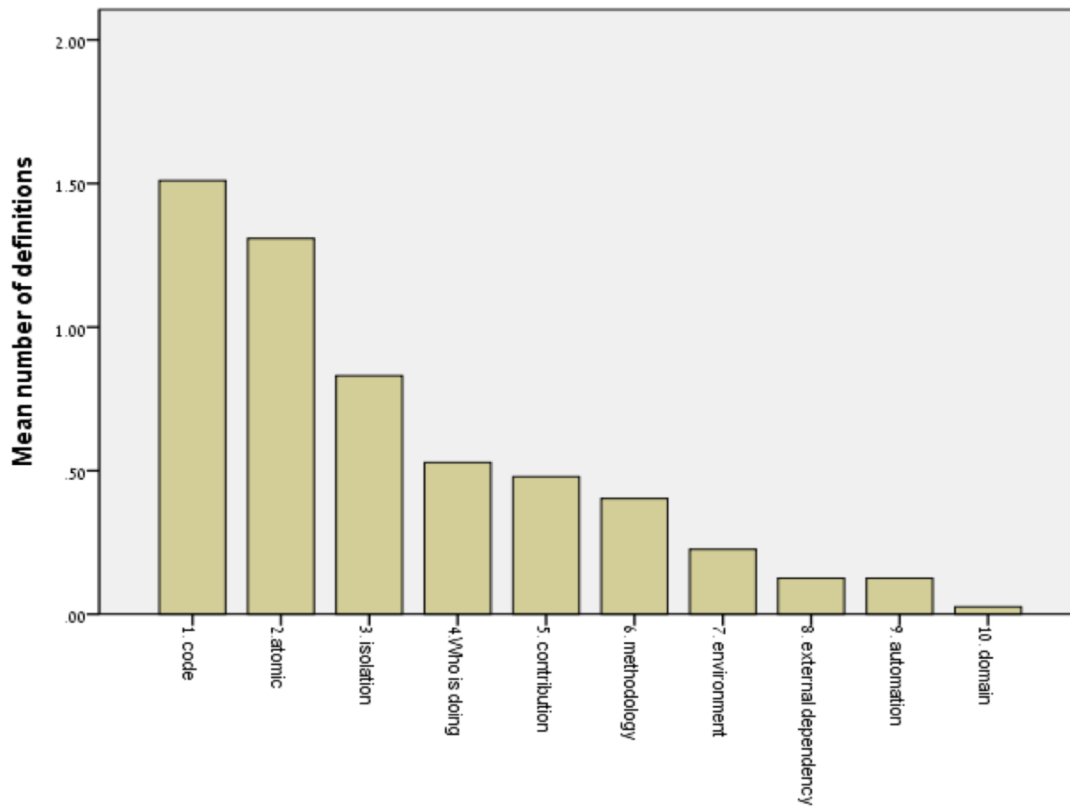


Figure 2: Categories of Definitions

At this stage, it was evident that there was no agreed-upon definition. Thus, we first attempted to extract affiliations and categories, relying on semantic relationships (semantic net) [4]. Definitions such as atomic, isolation, and code-related were classified as "classic" definitions in which the usage of xUnit test code tools was assumed. When the definition was more general, we called it a "component" definition. Figure 3 describes the distribution of the definitions using three categories.

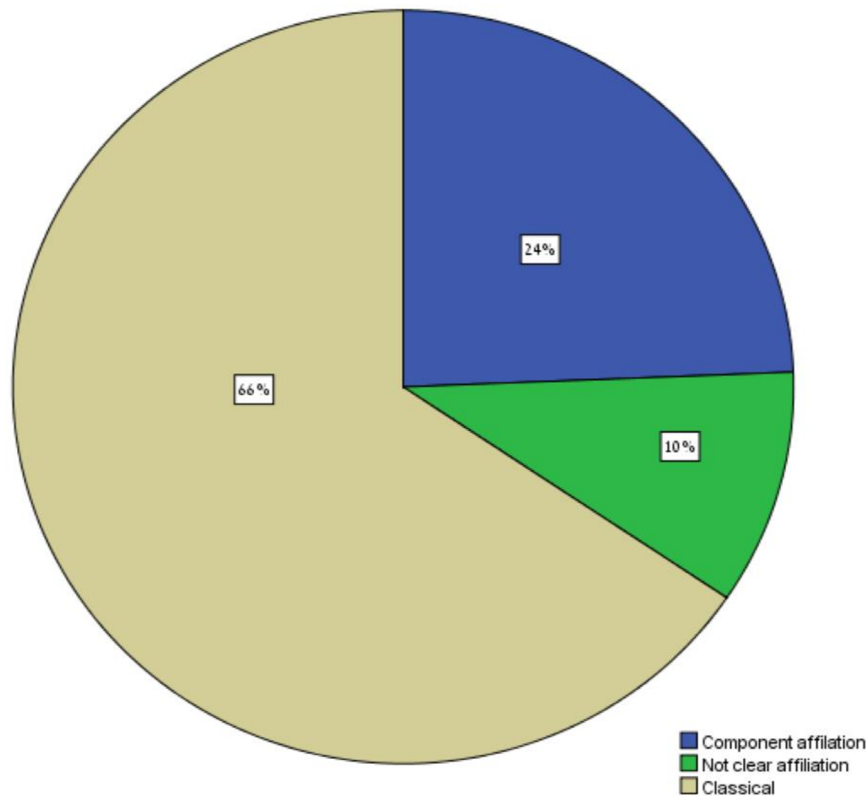


Figure 3: Definitions of "Unit Test"

While most (i.e., 66%) of the definitions were classical, we could not ignore the component definitions (24%); the remaining 10% represented cases where we could not extract a precise definition from the manner in which the term was being used.

"Unit Test" Usage Attributes

As reflected in published academic papers (see Appendix 1), emphasizing that the definition in itself is insufficient for an explanation of the usage of the term, Figure 4 maps how different publications use the term "unit test" as they address their specific readerships.

The theme of a given paper is sometimes correlated to the use of the term. We distinguished between papers that centered on the use of a specific tool and those that were concerned with general use or with theory. Almost half (47%) of the papers reported about new tool or solution in order to answer a need related to the unit test as displayed in Figure 5.

Topic of the Paper and Usage Attributes

A logistic regression was applied to examine the effect of the papers' topic on the use of unit test (Cox & Snell's $R2 < 0.1$). We failed to notice significant relations between the topic of the paper and the usage affiliation of the term in this case (Figure 6).

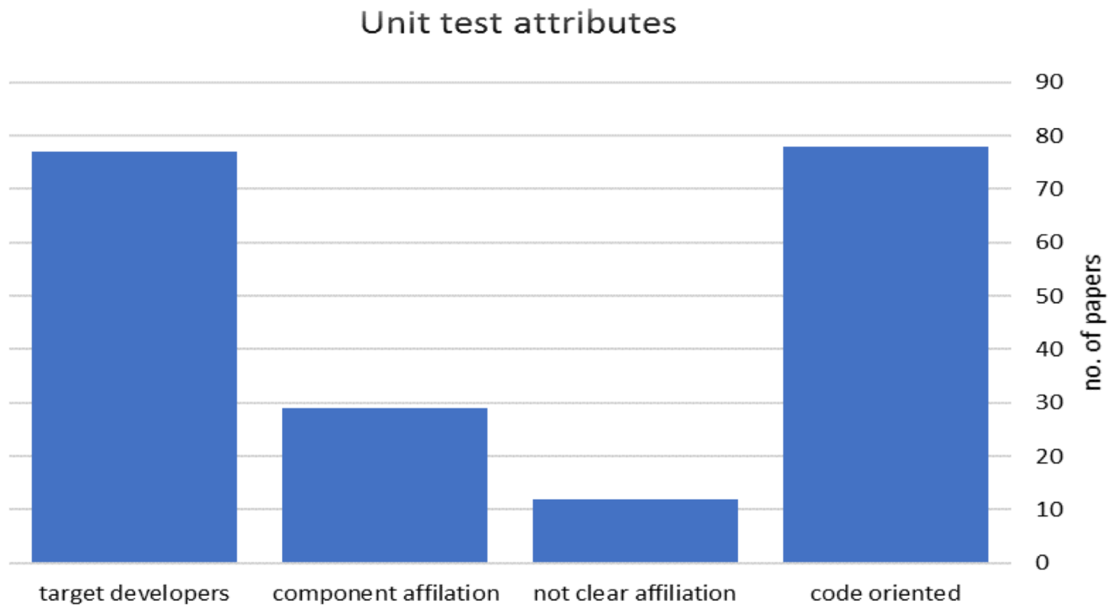


Figure 4: Attributes of Usage of the Term “Unit Test”

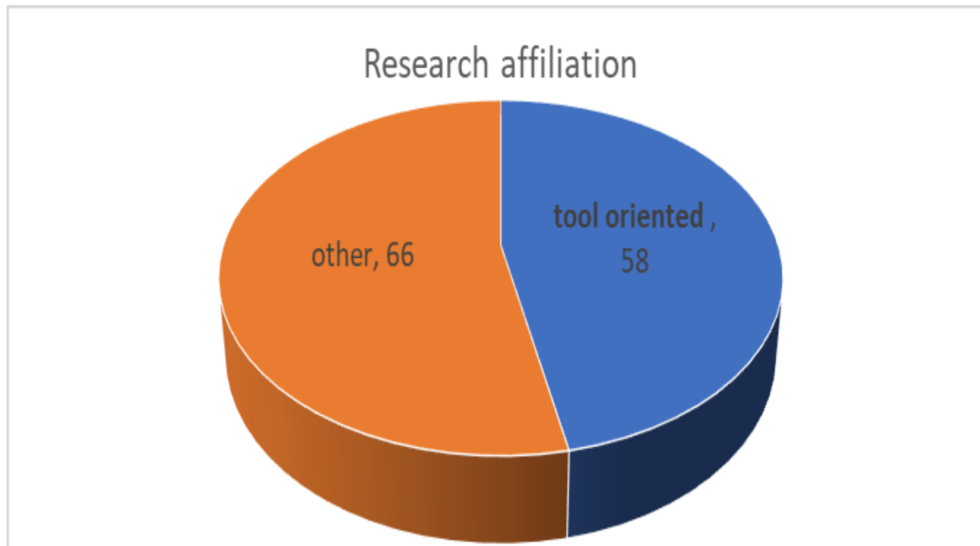


Figure 5: Topic Affiliation of the Research

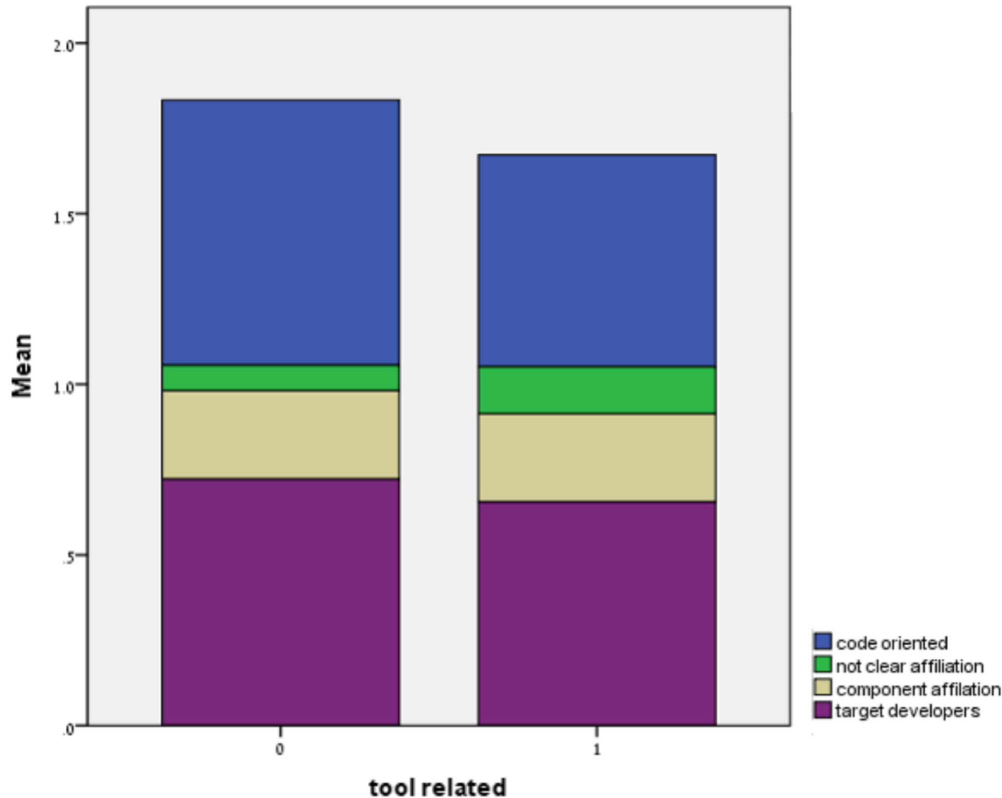


Figure 6: Topics of Papers and Affiliation of the Use of the Term "Unit Test"

Survey Summary Results

1. What categories and affiliations in the unit test definition can we identify? There are basically two diverse ways to define the term "unit test":
   - The classic way – About two-thirds of the papers related to the unit test as the smallest, isolated, atomic and code-related test that is mainly performed by the developers.
   - The component way (24%) – The focus is on a unit of functionality, not necessarily on the perception of the unit test as the smallest, indivisible portion of the program; here the unit test is administered mainly by testers.

   The rest of the papers (10%) did not define the term at all and require further examination in order to reveal how they understand the meaning of the term "unit test."
2. In addition to a formal definition within the paper, what attributes of usage could be found in the use of the term? Emphasizing that the definition is insufficient for an explanation of the usage of the term, we distinguished between papers that centered on the use of a specific tool and those that were concerned with general usage or with theory. We failed to relate a precise definition to specific usage terminologies.
3. Is there a connection between the main topic (or theme) of the paper and its use of the term "unit test"? Almost half (47%) of the papers were published reports of a new tool or a solution provided to facilitate a need related to the unit test. There was no significant connection between the topic of the paper and use of the term in this case.

OUR SUGGESTED DEFINITIONS

This research illustrates that at the most general level unit testing refers to the practice of testing specific functions, modules, or areas – or units – of software code.

This testing enables us to verify that the isolated piece of code functions as expected. In other words, for any function and a given set of inputs, one can determine whether the function is returning the proper values and will smoothly handle failures during execution should invalid input be provided. Developing software on a module level basis allows for more straightforward unit testing because the code under test has been isolated and/or is independent of other procedures in the code base. To enable well-defined unit testing, the code should be built with tight cohesion and loose coupling, with a more significant number of smaller, more focused functions that provide a single operation for a unique set of data rather than large functions performing several different procedures.

Unit tests are short code fragments created by programmers or occasionally by a white box (structural) or grey box (functional and structural) [2] – that is, by testers during the development process. The unit test is most often considered a lower test level. Unit testing is, roughly speaking, the testing of a small portion of the code in isolation from the test code. This is considered the first testing step during development and the most granular aggregate of the testing scheme. By its very nature, the unit test is attached to the code from which it is created. At the beginning of software development history, when attempting to test a particular functionality, testers were faced with the challenge of needing to have the program ready and operational before attempting to execute the test; the compiler would not let the code be executed before completing all the necessary declarations and building all of the affiliated infrastructures. Only then could testers perform the specific test. Apart from running the program in a debug mode, when they needed to test the precise functionality they had to develop an isolation mechanism to ensure the testing of specific code behavior. Unit tests are written as test classes with test methods. In the past, to display a similar behavior one had to develop a new code to mask the tested unit and had to inject artificial information into the tested object so that the program could be executed (this was sometimes called a mock mechanism). In general, mocking refers to the use of replacement classes that are easily configurable to react to input and provide the output during testing rather than the use of real classes. There are different related terms: for example, a stub usually has a fixed default behavior, whereas a mock-up typically must be set up as part of a test in order to verify expected interactions. Usually, the tested program and the actual final code were very different – because of the need to add instrumented observational code to the application code. It was only when the world moved into interpreter mode program execution that isolation was enabled more naturally, and unit test infrastructure appeared.

An important aspect of unit testing is the environment where the class within the unit under test is being operated [1]. Figure 7 demonstrates the internal environment needed for the execution of the test. Unit test infrastructure was designed as a pivotal element to enable isolation of the tested code before the full implementation of each object.

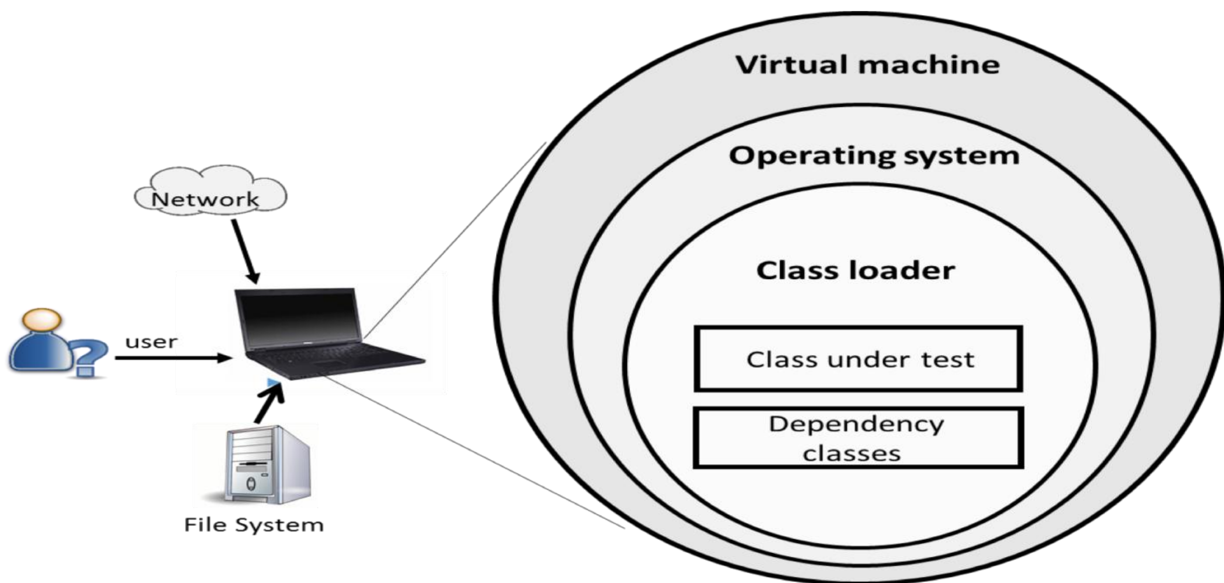


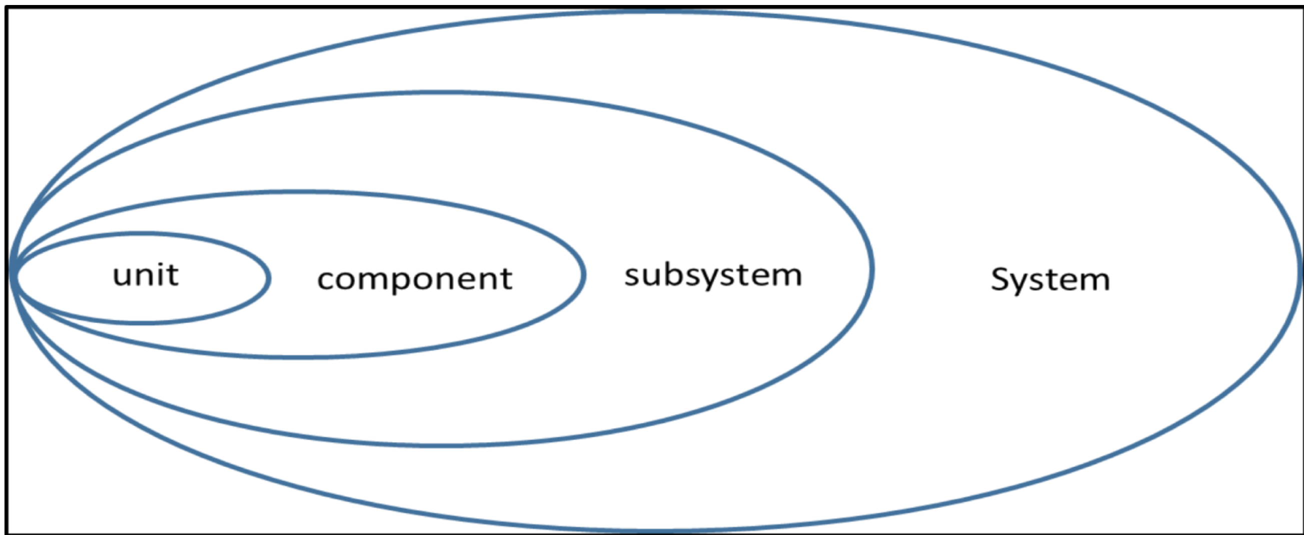
Figure 7: The Environment of a Test Class

In the modern development environment, a unit test method is a method without input parameters. It represents a test case and typically executes a method where the class has fixed arguments (use of assert class) and verifies that it is returning the expected answer [17]. Unit tests include a test oracle that verifies the observed behavior with the expected result. By convention, the test oracle of a unit test is encoded using assertions. The test fails if any assertion fails or if an exception occurs [17]. An assert class failure will usually stop all execution, and no exceptional treatment mechanism will be able to salvage the program execution during integration testing. Such a mechanism prevents us from refining the testing process, and all failures are equally critical to the testing process. The isolation and the predefined static value of the tested parameters lead by their very nature to the negation of any kind of integration between the different elements of the software under test.

Differentiation of the Unit Test from an Integration Test

For clarification, this paper sees an essential distinction between the unit test, which refers to the use of xUnit testing [18], and component testing, which is more general and means the testing of only a portion of the program. Another distinction might derive from the abstraction level, in which case the unit test will usually be affiliated with the code itself, and the component test may be expressed in functional or business terminology.

The unit test may form the basis for component testing that can be considered a higher level of testing. Component testing is sometimes known as module and program testing. Component testing is mostly done by the tester. Component testing may be done in isolation from the rest of the system depending on the development life cycle model chosen for that application. In such cases, the missing software is replaced by stubs and drivers that simulate the interface between the software components in a simple manner.



$$System = \sum Subsystems \left\{ \sum components \left[\sum units \right] \right\}$$

Figure 8: Architectural Buildup of a Testing Level

The greatest pitfall might be encountered when developers test too large a unit or when they consider a method to be a unit. This is particularly true if you do not understand Inversion of Control, in which case your unit tests will always turn into end-to-end integration testing. Unit testing should test individual behaviors – and most methods have many behaviors [6]. In some NASA projects, such as the agency's "Flight Software Product Line" [6], one could see the possibility of the creation of unit tests without the use of traditional xUnit infrastructure (and without the use of the assert class).

It is therefore vital to explicitly define the terminology when talking about the unit test and unit test integration. For example, Unit Test Virtualization [3] is a

new approach for reducing the execution time of long test suites. VmVm (pronounced “vroom-vroom”) is an easy-to-use device for the implementation of Unit Test Virtualization wherever Java is used. It allows us to perform a pre-initiation of the test environment and to avoid the need for restarting the execution process following each failure. The idea behind the approach is obvious but fails to distinguish between the unit test and unit test integration nor does it address issues raised by the use of an assert class. This approach is perhaps more suitable for component tests.

Another example is work with Open Worm, as reported by Sarma et al. [15], where a unit test applies to the smallest functional unit of code and has no external dependencies. On the other hand, tests intended to verify that different components are working together are classified as integration tests. They assess whether multiple components have been integrated correctly. Some of the tests discussed below focus on another distinction that the authors make, rather than distinguishing between ordinary verification tests (designed to verify that the code is working as intended) and model validation tests (designed to validate a model against experimental data).

Usually, the isolation issue is presented through the use of mock object technology [8]. While mock objects help us remove unnecessary dependencies in tests and make the tests fast and reliable, the use of mocks manually written in C++ is problematic:

- Someone must implement the mocks. The job is usually tedious and error-prone, and it is no wonder that researchers go great distances to avoid it.
- The quality of these manually written mocks is somewhat unpredictable. You might see some polished mocks, but you will also see some that have been hacked up in a hurry and which have a large number of ad hoc restrictions.
- The knowledge you gain from using one mock cannot be applied to the next one.

Discussions and Considerations

In contrast, Java and Python programmers have some excellent mock frameworks that automate the creation of mocks. Tests that rely on external API (application protocol interface), network connections, user input, threading, and other external dependencies must be mocked. A passing test must continue to be administered as long as the codebase remains in the same state. If the network connection suddenly becomes disconnected, the code will subsequently fail. However, if a mock is implemented in place of the actual network connection, the tests will continue to pass. Thus, mocking has shown itself to be a proven and effective technique and is a widely adopted practice. Having the right tool absolutely makes the difference.

Another important aspect worth considering is the identity and research specialization of those who are administering the different testing levels. We argue here that another set of skills and knowledge should be considered when you must choose the right person to plan and perform these testing activities.

It is our experience that when developers focus on the unit level, it becomes more difficult for them to comprehend the state of their code in the larger scheme of the code base. Consequently, their units cannot directly interact with other sections of the overall codebase, as the developed units are essentially too isolated.

On the other hand, the refactoring process of a *tested* unit of code [12] can complicate our understanding of a unit and a unit test. Let us assume, for example, that we have successfully tested and implemented an isolated portion of code with no external dependencies. Once all the tests pass, the developers must refactor their code. During the refactoring process, the previously identified *unit* of code can be extracted to multiple methods or classes. If the extracted classes are themselves considered units, then we have fundamentally undermined our original test. Even more complicated than method extraction, code that is migrated to an abstract parent class cripples our definition of a unit even further. Abstract classes cannot be tested because their code cannot be instantiated. Do we relate to the inheriting classes of the abstract class in order to test their shared parent?

Under ideal code conditions for a unit (isolation, atomic, etc.), we can address the aptitude of the programmers or testers for adequately understanding a given program’s requirements. A correctly written test can be executed on an isolated section of code and can pass, if, unfortunately, the developer did not accurately understand the necessary requirements. As a result, all the tests will pass although many of them did not actually validate the intended functionality of the code.

Regarding the degradation of code, all matter is subject to natural deterioration. The robustness of the matter largely influences the pace at which the deterioration occurs. Software, albeit abstract and intangible, is equally vulnerable to the deterioration and degeneration phenomena found in the physical world around us. More research is imperative if we are to better understand the underlying source and effects of software degradation (code rot) and the significance of unit tests in the creation of a more durable, more robust software product.

CONCLUSIONS

It is apparent that the definition of the term "unit test" is neither clear nor precise. Most of the literature we have reviewed tend to consider the structural aspect of the term – atomic, isolation, etc. – and relates the action to an X-unit testing infrastructure. About 24% of the sources define "unit test" more loosely and display a higher level of abstraction that does not restrict the definition, and which allows an integrative portion of the program to be included in the unit being tested.

- Unit testing – This is the act of testing an isolated, atomic, and code-related portion of the software (a unit). It is evident that the right candidates to perform this activity are the developers themselves.
- Component testing – This is the testing of a functional and more substantial portion of the program (a component). We claim that another set of skills and another kind of knowledge are needed to perform this portion of the work.

It is vital in our opinion to distinguish between the two aspects and to allocate the best resources for each assignment, or, alternatively, to train the developers and provide them with new skills and knowledge so that they can perform these two categories of testing.

As stated in the Introduction, the move to CICD shows shifting the center of testing activities into unit testing. Therefore, we can identify growing importance of the role of the unit testing level, which emphasizes the importance of the distinction between classical unit testing level and the integration (component) level. We recommend that the two aspects of testing be separated in the early stages of software development.

Threat to Validity

One of the possible threats to the validity of a survey's findings is related to the appropriateness of the data collection and sampling approaches. To attain optimal objectivity in our study, it was important to collect a representative, statistically sound sample (our full list of papers is included in Appendix 1). Another possible threat may stem from the fact that we had to subjectively interpret each definition and each use of the terms in question. To prevent discrepancies, we had two independent teams repeat the paper reviewing process.

The Next Step

To fully understand the impact to the quality of a software product, it is important to follow this research with a field study of the actual implementations of unit test as part of the software quality assurance during a CI/CD project. Since unit testing is considered an important link in a chain of quality activities aimed to improve organization outcome, understanding that will assist organizations in focusing their quality goals and recruitment needs.

Additionally, we can identify other future research directions:

- Checking whether the current tools enable the two activities – We propose a continuation of research on the existing testing tool with the aim of determining whether there are tools that enable both unit test definitions noted above.
- The present research study did not explore the conditions under which unit testing is enabled. It may be interesting to identify the terms of feasibility for the proper use of unit tests, or, in other words, to determine when unit testing cannot be performed.
- Another possible direction is to try to bridge the two definitions of unit testing by proposing a metamorphosis of unit test artifacts into component testing.
- The recognition of the open-source community as a mainstream constituent in global software development has produced new research motivations. GitHub, Stack Overflow, and Twitter, to mention but a few social profiles, are the main vehicles for individuals today to promote and publish their opinions, styles, and code. Testing and quality constitute core values in most open-source projects. The comparison and contrast of the definitions, perceptions, and implementations of quality in open-source community projects versus the competitive software industry could provide us with an even more profound understanding of testing and, more specifically, unit testing.

REFERENCES

[1] Arcuri, A., G. Fraser, and J.P. Galeotti. *Automated unit test generation for classes with environment dependencies*. in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014. ACM.

[2] Beizer, B. and J. Wiley, *Black Box Testing: Techniques for Functional Testing of Software and Systems.* Software, IEEE, 1996. 13(5): p. 98.

[3] Bell, J. and G. Kaiser. *VMVM: unit test virtualization for Java*. in *Companion Proceedings*

of the 36th International Conference on Software Engineering. 2014. ACM.

[4] Clark, P., et al. *Exploiting a thesaurus-based semantic net for knowledge-based search*. in *AAAI/IAAI*. 2000.

[5] Daka, E. and G. Fraser. *A survey on unit testing practices and problems*. in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014. IEEE.

[6] Ganesan, D., et al., *An analysis of unit tests of a flight software product line.* Science of Computer Programming, 2013. **78**(12): p. 2360-2380.

[7] Hunter, A., *Are unit test overused* in *Simple talk* 2012: **https://www.simple-talk.com/dotnet/net-framework/are-unit-tests-overused/**

[8] Lewis, W.E., *Software testing and continuous quality improvement*. 2016: CRC press.

[9] Marschall, Philippe. "Detecting the methods under test in java." *Bachelor thesis* (2005).

[10] Michael, H. 2016. Understanding and improving continuous integration. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE 2016). ACM, New York, NY, USA, 1066-1067. DOI: https://doi.org/10.1145/2950290.2983952

[11] Naik, K. and P. Tripathy, *Software testing and quality assurance: theory and practice*. 2011: John Wiley & Sons.

[12] Passier, H., L. Bijlsma, and C. Bockisch. *Maintaining Unit Tests During Refactoring*. in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 2016. ACM.

[13] Raymond, E.S., *The art of Unix programming*. 2003: Addison-Wesley Professional.

[14] Runeson, P., *A survey of unit testing practices.* IEEE software, 2006. **23**(4): p. 22-29.

[15] Sarma, G.P., et al., *Unit testing, model validation, and biological simulation.* arXiv preprint arXiv:1508.04635, 2015.

[16] Ståhl, D., and Bosch, J. "Modeling continuous integration practice differences in industry software development." Journal of Systems and Software 87 (2014): 48-59.

[17] Tillmann, N. and W. Schulte. *Parameterized unit tests*. in *ACM SIGSOFT Software Engineering Notes*. 2005. ACM.

[18] Verona, Joakim, Michael Duffy, and Paul Swartout. *Learning DevOps: Continuously Deliver Better Software*. Packt Publishing Ltd, 2016. page 88

AUTHOR BIOGRAPHIES

Hadas Chasidim is a faculty member at Shamoon College of Engineering's Department of Software Engineering. She holds B.Sc., M.Sc. and Ph.D degrees in Industrial Engineering from Ben-Gurion University (2013). Her research deals with software quality and testing, human-computer interaction, usable privacy and security. Hadas is the head of software quality development track at Software Engineering department in Beer Sheva campus.

Dani Almog is a senior researcher and lecturer on Software Quality and test automation. Contributing researcher and lecturer at Shamoon College of Engineering (SCE) and Ben Gurion University (BGU). Currently, the main topic of his studies are Software quality testing and fundamentals, addressing software engineering needs. Dani has very vast experience in the industry – former test automation managing director for Amdocs product development division. Dani is serving at the Advisory Board of ITCB (Israeli Testing Certification Board) and Leading the ISTQB academia research group stream.

Dov Benyomin Sohacheski is currently a graduate student and holds a B.Sc. in Software Engineering from SCE Shamoon College of Engineering. He is a researcher and an assistant lecturer at SCE. Dov has over 8 years of experience in the software development industry as a PHP and Python developer, working on projects from web-based software services to data-analytics and statistical platforms.

Mark L. Gillenson is Professor of Business Information and Technology in the Fogelman College of Business and Economics of the University of Memphis, USA. He received his B.S. degree in Mathematics from Rensselaer Polytechnic Institute and his M.S. and Ph.D. degrees in Computer and Information Science from The Ohio State University. He is an associate editor of the Journal of Database Management and has published in such leading journals as MIS Quarterly, European Journal of Information Systems, and Information & Management. His latest book is Fundamentals of Database Management Systems 2nd edition, 2012, John Wiley & Sons.

Dr. Robin Poston is the Director of the System Testing Excellence Program for the FedEx Institute of Technology at The University of Memphis, and she is a Professor and Dept Chair of Business Information and Technology at the Fogelman College of Business & Economics at The University of Memphis. She is holds the

Papasan Family Professorship for Exemplary Leadership, serves as the Interim Associate Dean of the Graduate School. Dr. Poston is a recipient of the Memphis Alumni Association Distinguished Teaching Award and she leads the annual International Research Workshop on Advances and Innovations in Software Testing attended by over a hundred academic and industry professionals.

Prof. Shlomo Mark currently the dean of the faculty of engineering at SCE College of Engineering - Ashdod campus is a professor of software engineering, and the Head of the NMCRC – the Negev Monte Carlo Research Center at SCE. His main research interests are Quality in software engineering, Agility and Agile life cycle, scientific computing and software Life Cycle for Scientific software product, Computational Modeling for Physical, Environmental and Medical Application.