Achieving Software Engineering Knowledge Items with an Unit Testing Approach

A.Nirmal Kumar, B.G.Geetha

Abstract*-- Classification makes a vital role to advancing knowledge in both science and engineering. It is a process of investigating the relationships between the objects to be classified and identifies gaps in knowledge. Classification in engineering also has a practical application. They can help maturing Software Engineering knowledge, as classifications constitute an organized structure of knowledge items. Till date, in existing system, there have been few attempts at classifying in test cases. In this research, we examine how useful classifications in Software Engineering are for advancing knowledge by trying to classify testing techniques. This paper presents a preliminary classification of a set of unit testing techniques. To obtain this classification, we enacted a generic process for developing useful Software Engineering classifications. The proposed classification has been proven useful for maturing knowledge about testing techniques. SE helps to: 1) provide a systematic description of the techniques,2) understand testing techniques by studying the relationships among techniques (measured in terms of differences and similarities), 3) identify potentially useful techniques that do not yet exist by analyzing gaps in the classification, and 4) support practitioners in testing technique selection by matching technique characteristics to project characteristics.

Index Terms*-- Classification, software engineering, software testing, test design techniques, testing techniques, unit testing techniques.

I. INTRODUCTION

In science and engineering, knowledge matures as the investigated objects are classified. Mature knowledge is not a sequential heap of pieces of knowledge, but an organized structure of knowledge items, where each piece smoothly and elegantly fits into place, as in a puzzle. Classification groups similar objects to form an organization. Examples are the classification of living beings in the natural sciences, diseases in medicine, elements in chemistry, architectural styles in architecture, materials in civil engineering, etc. The unit testing process is composed of three *phases* that are partitioned into a total of eight basic *activities* as follows:

1) *Perform the test planning*
 a) Plan the general approach, resources, and schedule
 b) Determine features to be tested
 c) Refine the general plan
2) *Acquire the test set*
 a) Design the set of tests
 b) Implement the refined plan and design
3) *Measure the test unit*
 a) Execute the test procedures
 b) Check for termination
 c) Evaluate the test effort and unit

The major dataflows into and out of the phases are shown in Fig A.

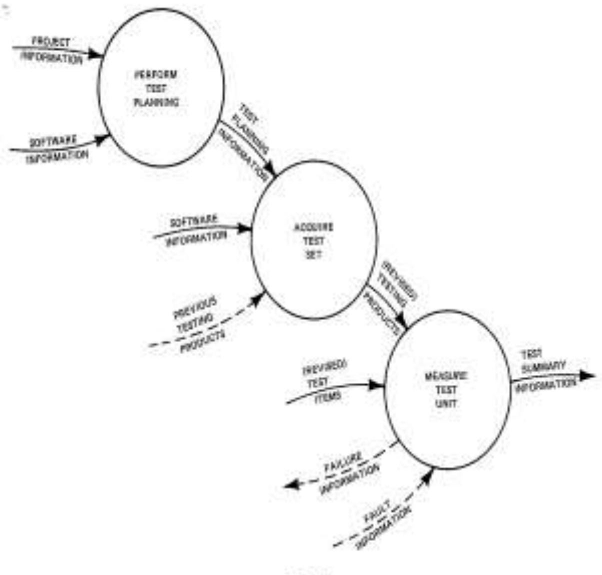

Fig A

Within a phase, each basic activity is associated with its own set of inputs and outputs and is composed of a series of tasks. The inputs, tasks, and outputs for each activity are specified in the body of this standard. When more than one unit is to be unit tested (for example, all those associated with a software project), the Plan activity should address the total set of test units and should not be repeated for each test unit. The other activities must be performed at least once for each unit.

II. UNIT TESTING ACTIVITIES

Under normal conditions, these activities are sequentially initiated except for the Execute and Check cycle
as illustrated in Fig 1. When performing any of the activities except Plan, improper performance of a preceding activity or external events (for example, schedule, requirements, or design changes) may result in the need to redo one or more of the preceding activities and then return to the one being performed.

Manuscript received February 16, 2012.

A.Nirmal Kumar, Assistant Professor,Department of CSE , Christian College of Engineering and Technology, Dindigul, TamilNadu, India. +91-9443998771, (e-mail: sa.nirmalkumar@gmail.com).
Dr.B.G.Geetha, Professor & Head, Department of CSE, KSR College of Technology, Tiruchengode, TamilNadu, India. (e-mail: geethaksrct@gmail.com).

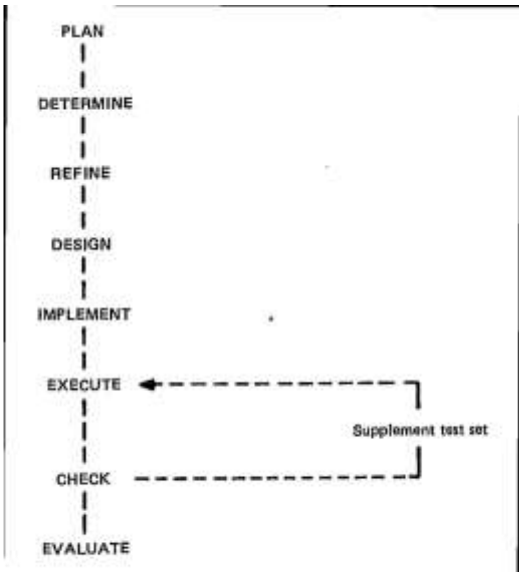

Fig 1
Unit Testing Activities

During the testing process, a test design specification and a test summary report must be developed. Other test documents may be developed. All test documents must conform to the ANSI/IEEE Std 829-1983 [2]. In addition, all test documents must have identified authors and be dated. The test design specification will derive its information from the Determine, Refine, and Design activities. The test summary report will derive its information from all of the activities.

2.1 Plan the General Approach, Resources, and Schedule

General unit test planning should occur during overall test planning and be recorded in the corresponding planning document.

2.1.1 Plan Inputs

1) Project plans
2) Software requirements documentation

2.1.2 Plan Tasks

(1)*Specify a General Approach to Unit Testing.*

Identify risk areas to be addressed by the testing. Specify constraints on characteristic determination (for example, features that must be tested), test design, or test implementation (for example, test sets that must be used).

Identify existing sources of input, output, and state data (for example, test files, production files, test data generators). Identify general techniques for data validation. Identify general techniques to be used for output recording, collection, reduction, and validation. Describe provisions for application Software that directly interfaces with the units to be tested.

(2)*Specify Completeness Requirements.*

Identify the areas (for example, features, procedures, states, functions, data characteristics, instructions) to be covered by the unit test set and the degree of coverage required for each area. When testing a unit during software development, every software feature must be covered by a test case or an approved exception. The same should hold during software maintenance for any unit testing. When testing a unit implemented with a procedural language (for example, COBOL) during software development, every instruction that can be reached and executed must be covered by a test case or an approved exception, except for instructions contained in modules that have been separately unit tested. The same should hold during software maintenance for the testing of a unit implemented with a procedural language.

(3)*Specify Termination Requirements.*

Specify the requirements for normal termination of the unit testing process. Termination requirements must include satisfying the completeness requirements. Identify any conditions that could cause abnormal termination of the unit testing process (for example, detecting a major design fault, reaching a schedule deadline) and any notification procedures that apply.

(4)*Determine Resource Requirements.*

Estimate the resources required for test set acquisition, initial execution, and subsequent repetition of testing activities. Consider hardware, access time (for example, dedicated computer time), communications or system software, test tools, test files, and forms or other supplies. Also consider the need for unusually large volumes of forms and supplies. Identify resources needing preparation and the parties responsible. Make arrangements for these resources, including requests for resources that require significant lead time (for example, customized test tools). Identify the parties responsible for unit testing and unit debugging. Identify personnel requirements including skills, number, and duration.

(5)*Specify a General Schedule.*

Specify a schedule constrained by resource and test unit availability for all unit testing activity.

2.1.3 Plan Outputs

(1) General unit test planning information (from 2.1.2 (1) through (5) inclusive)

(2) Unit test general resource requests if produced from 2.1.2 (4)

2.2 Determine Features To Be Tested

2.2.1 Determine Inputs

(1) Unit requirements documentation
(2) Software architectural design documentation if needed

2.2.2 Determine Tasks

(1)*Study the Functional Requirements.*

Study each function described in the unit requirements documentation. Ensure that each function has a unique identifier. When necessary, request clarification of the requirements.

(2)*Identify Additional Requirements and Associated Procedures.*

Identify requirements other than functions(for example, performance, attributes, or design constraints) associated with software characteristics that can be effectively tested at the unit level. Identify any usage or operating procedures associated only with the unit to be tested. Ensure that each additional requirement and procedure has a unique identifier. When necessary, request clarification of the requirements.

(3)*Identify States of the Unit.*
If the unit requirements documentation specifies or implies multiple states (for example, inactive, ready to receive, processing) software, identify each state and each valid state transition. Ensure that each state and state transition has a unique identifier. When necessary, request clarification of the requirements.

(4)*Identify Input and Output Data Characteristics.*
Identify the input and output data structures of the unit to be tested. For each structure, identify characteristics, such as arrival rates, formats, value ranges, and relationships between field values. For each characteristic, specify its valid ranges.
Ensure that each characteristic has a unique identifier. When necessary, request clarification of the requirements.

(5)*Select Elements to be Included in the Testing.*
Select the features to be tested. Select the associated, procedures, associated states, associated state transitions, and associated data characteristics to be included in the testing. Invalid and valid input data must be selected. When complete testing is impractical, information regarding the expected use of the unit should be used to determine the selections.
Identify the risk associated with unselected elements. Enter the selected features, procedures, states, state transitions, and data characteristics in the *Features to be Tested* section of the unit's Test Design Specification.

2.2.3 Determine Outputs

(1) List of elements to be included in the testing (from 2.2.2 (5))

(2) Unit requirements clarification requests; if produced from 2.2.2 (1) through (4) inclusive.

2.3 Refine the General Plan

2.3.1 Refine Inputs

(1) List of elements to be included in the testing (from 2.2.2 (5))

(2) General unit test planning information (from 2.1.2 (1) through (5) inclusive)

2.3.2 Refine Tasks

(1)*Refine the Approach.*
Identify existing test cases and test procedures to be considered for use. Identify any special techniques to be used for data validation. Identify any special techniques to be used for output recording, collection, reduction, and validation.
Record the refined approach in the *Approach Refinements* section of the unit's test design specification.

(2)*Specify Special Resource Requirements.*
Identify any special resources needed to test the unit (for example, software that directly interfaces with the unit). Make preparations for the identified resources. Record the special resource requirements in the *Approach Refinements* section of the unit's test design specification.

(3)*Specify a Detailed Schedule.*
Specify a schedule for the unit testing based on support software, special resource, and unit availability and integration schedules. Record the schedule in the *Approach Refinements* section of the unit's test design specification.

2.3.3 Refine Outputs

(1) Specific unit test planning information (from 2.3.2 (1) through (3) inclusive)

(2) Unit test special resource requests; if produced from 2.3.2 (2).

2.4 Design the Set of Tests

2.4.1 Design Inputs

(1) Unit requirements documentation

2) List of elements to be included in the testing (from 2.2.2 (5))

(3) Unit test planning information (from 2.1.2 (1) and (2) and 2.3.2 (1))

(4) Unit design documentation

(5) Test specifications from previous testing; if available

2.4.2 Design Tasks

(1) *Design the Architecture of the Test Set.* Based on the features to be tested and the conditions specified or implied by the selected associated elements (for example, procedures, state transitions, data characteristics), design a hierarchically decomposed set of test objectives so that each lowest-level objective can be directly tested by a few test cases. Select appropriate existing test cases. Associate groups of test-case identifiers with the lowest-level objectives. Record the hierarchy of objectives and associated test case identifiers in the *Test Identification* section of the unit's test design specification.

(2) *Obtain Explicit Test Procedures as Required.* A combination of the unit requirements documentation, test planning information, and test-case specifications may implicitly specify the unit test procedures and therefore minimize the need for explicit specification. Select existing test procedures that can be modified or used without modification.
Specify any additional procedures needed either in a supplementary section in the unit's test design specification or in a separate procedure specification document. Either choice must be in accordance with the information required by ANSI/IEEE Std 829-1983 [2]. When the correlation between test cases and procedures is not readily apparent, develop a table relating them and include it in the unit's test design specification.

(3) *Obtain the Test Case Specifications.* Specify the new test cases. Existing specifications may be referenced.
Record the specifications directly or by reference in either a supplementary section of the unit's test design specification or in a separate document. Either choice must be in accordance with the information required by ANSI/IEEE Std 829-1983 [2].

(4) *Augment, as Required, the Set of Test-Case Specifications Based on Design Information.* Based on information about the unit's design, update as required the test set architecture in accordance with 2.4.2 (1). Consider the characteristics of selected algorithms and internal data structures. Identify control flows and changes to internal data that must be recorded. Anticipate special recording difficulties that might arise, for example, from a need to trace control flow in complex

algorithms or from a need to trace changes in internal data structures (for example, stacks or trees). When necessary, request enhancement of the unit design (for example, a formatted data structure dump capability)
to increase the test-ability of the unit. Based on information in the unit's design, specify any newly identified test cases and complete any partial test case specifications in accordance with 2.4.2 (3).

(5) *Complete the Test Design Specification.* Complete the test design specification for the unit in accordance
with ANSI/IEEE Std 829-1983 [2].

2.4.3 Design Outputs

(1) Unit test design specification (from 2.4.2 (5))

(2) Separate test procedure specifications; if produced from 2.4.2 (2)

(3) Separate test-case specifications; if produced from 2.4.2 (3) or (4)

(4) Unit design enhancement requests; if produced from 2.4.2 (4)

2.5 Implement the Refined Plan and Design

2.5.1 Implement Inputs

(1) Unit test planning information (from 2.1.2 (1), (4), and (5) and 2.3.2 (1) through (3) inclusive)

(2) Test-case specifications in the unit test design specification or separate documents (from 2.4.2 (3)
and (4)

(3) Software data structure descriptions

(4) Test support resources

(5) Test items

(6) Test data from previous testing activities; if available

(7) Test tools from previous testing activities; if available

2.5.2 Implement Tasks

(1) *Obtain and Verify Test Data.* Obtain a copy of existing test data to be modified or used without modification. Generate any new data required. Include additional data necessary to ensure data consistency and integrity. Verify all data (including those to be used as is) against the software data structure specifications. When the correlation between test cases and data sets is not readily apparent, develop a table to record this correlation and include it in the unit's test design specification.

(2) *Obtain Special Resources.* Obtain the test support resources specified in 2.3.2 (2).

(3) *Obtain Test Items.* Collect test items including available manuals, operating system procedures, control data (for example, tables), and computer programs. Obtain software identified during test planning that directly interfaces with the test unit. When testing a unit implemented with a procedural language, ensure that execution trace information will be available to evaluate satisfaction of the code-based completeness requirements. Record the identifier of each item in the *Summary* section of the unit's test summary report.

2.5.3 Implement Outputs

(1) Verified test data (from 2.5.2 (1))

(2) Test support resources (from 2.5.2 (2))

(3) Configuration of test items (from 2.5.2 (3))

(4) Initial summary information (from 2.5.2 (3))

2.6 Execute the Test Procedures

2.6.1 Execute Inputs

(1) Verified test data (from 2.5.2 (1))

(2) Test support resources (from 2.5.2 (2))

(3) Configuration of test items (from 2.5.2 (3))

(4) Test-case specifications (from 2.4.2 (3) and (4))

(5) Test procedure specifications (from 2.4.2 (2)); if produced

(6) Failure analysis results (from debugging process); if produced

2.6.2 Execute Tasks

(1) *Run Tests.* Set up the test environment. Run the test set. Record all test incidents in the *Summary of Results* section of the unit's test summary report.

(2) *Determine Results.* For each test case, determine if the unit passed or failed based on required result specifications in the case descriptions. Record pass or fail results in the *Summary of Results* section of the unit's test summary report. Record resource consumption data in the *Summary of Activities* section of the report. When testing a unit implemented with a procedural language, collect execution trace summary information and attach it to the report. For each failure, have the failure analyzed and record the fault information in the *Summary of Results* section of the test summary report. Then select the applicable case and perform the associated

Case 1: A Fault in a Test Specification or Test Data. Correct the fault, record the fault correction in the *Summary of Activities* section of the test summary report, and rerun the tests that failed.

Case 2: A Fault in Test Procedure Execution. Rerun the incorrectly executed procedures.

Case 3: A Fault in the Test Environment (for example, system software). Either have the environment corrected, record the fault correction in the *Summary of Activities* section of the test summary report, and rerun the tests that failed OR prepare for abnormal termination by documenting the reason for not correcting the environment in the *Summary of Activities* section of the test summary report and proceed to check for termination (that is, proceed to activity 2.7).

Case 4: A Fault in the Unit Implementation. Either have the unit corrected, record the fault correction in the *Summary of Activities* section of the test summary report, and rerun all tests OR prepare for abnormal termination by documenting the reason for not correcting the unit in the *Summary of Activities* section of the test summary report and proceed to check for termination (that is, proceed to activity 2.7).

Case 5: A Fault in the Unit Design. Either have the design and unit corrected, modify the test specification and data as appropriate, record the fault correction in the *Summary of Activities* section of the test summary report, and rerun all tests OR prepare for abnormal termination by documenting the reason for not correcting the design in the *Summary of Activities* section of the test summary report and proceed to check for termination (that is, proceed to activity 2.7).

NOTE: The cycle of Execute and Check Tasks must be repeated until a termination condition defined in 2.1.2 (3) is satisfied (See Fig 3). Control flow within the Execute activity itself is pictured in Fig 2.

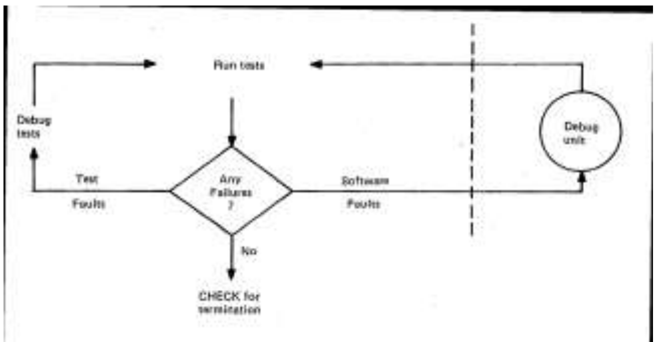

Fig 2
Control Flow Within the Check Activity

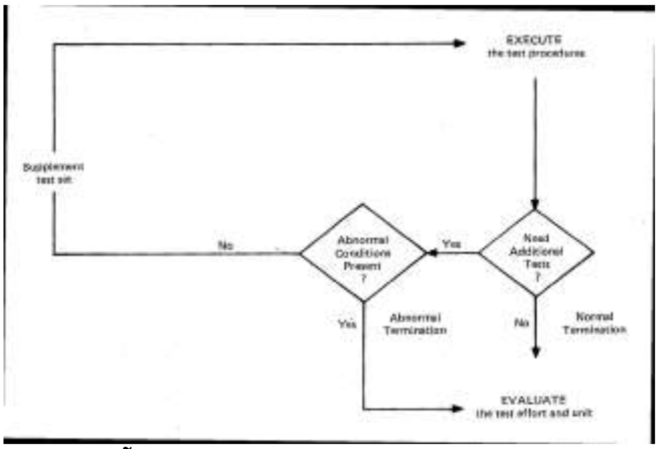

Fig 3 Ñ Control Flow Within the Check Activity

2.6.3 Execute Outputs

(1) Execution information logged in the test summary report including test outcomes, test incident descriptions, failure analysis results, fault correction activities, uncorrected fault reasons, resource consumption data and, for procedural language implementations, trace summary information (from 2.6.2 (1) and (2))

(2) Revised test specifications; if produced from 2.6.2 (2)

(3) Revised test data; if produced from 2.6.2 (2)

2.7 Check for Termination

2.7.1 Check Inputs

(1) Completeness and termination requirements (from 2.1.2 (2) and (3))

(2) Execution information (from 2.6.2 (1) and (2))

(3) Test specifications (from 2.4.2 (1) through (3) inclusive); if required

(4) Software data structure descriptions; if required

2.7.2 Check Tasks

(1) *Check for Normal Termination of the Testing Process.* Determine the need for additional tests based on completeness requirements or concerns raised by the failure history. For procedural language implementations, analyze the execution trace summary information (for example, variable, flow).

If additional tests are *not* needed, then record normal termination in the *Summary of Activities* section of the test summary report and proceed to evaluate the test effort and unit (that is, proceed to activity 2.8).

(2) *Check for Abnormal Termination of the Testing Process.* If an abnormal termination condition is satisfied (for example, uncorrected major fault, out of time) then ensure that the specific situation causing termination is documented in the *Summary of Activities* section of the test summary report

together with the unfinished testing and any uncorrected faults. Then proceed to evaluate the test effort and unit (that is, proceed to activity 2.8).

(3) *Supplement the Test Set.* When additional tests are needed and the abnormal termination conditions are not satisfied, supplement the test set by following steps (a) through (e).

(a) Update the test set architecture in accordance with 2.4.2 (1) and obtain additional test-case specifications in accordance with 2.4.2 (3).

(b) Modify the test procedure specifications in accordance with 2.4.2 (2) as required.

(c) Obtain additional test data in accordance with 2.5.2 (1).

(d) Record the addition in the *Summary of Activities* section of the test summary report.

(e) Execute the additional tests (that is, return to activity 2.6).

2.7.3 Check Outputs

(1) Check information logged in the test summary report including the termination conditions and any test case addition activities (from 2.7.2 (1) through (3) inclusive)

(2) Additional or revised test specifications; if produced from 2.7.2 (3)

(3) Additional test data; if produced from 2.7.2 (3)

2.8 Evaluate the Test Effort and Unit

2.8.1 Evaluate Inputs

(1) Unit Test Design Specification (from 2.4.2 (5)

(2) Execution information (from 2.6.2 (1) and (2))

(3) Checking information (from 2.7.2 (1) through (3) inclusive)

4) Separate test-case specifications (from 2.4.2 (3) and (4)); if produced

2.8.2 Evaluate Tasks

(1) *Describe Testing Status.* Record variances from test plans and test specifications in the *Variances* section of the test summary report. Specify the reason for each variance. For abnormal termination, identify areas insufficiently covered by the testing and record reasons in the *Comprehensiveness Assessment* section of the test summary report. Identify unresolved test incidents and the reasons for a lack of resolution in the *Summary of Results* section of the test summary report.

(2) *Describe Unit's Status.* Record differences revealed by testing between the unit and its requirements documentation in the *Variances* section of the test summary report. Evaluate the unit design and implementation against requirements based on test results and detected fault information. Record evaluation information in the *Evaluation* section of the test summary report.

(3) *Complete the Test Summary Report.* Complete the test summary report for the unit in accordance with ANSI/IEEE Std 829-1983 [2].

(4) *Ensure Preservation of Testing Products.* Ensure that the testing products are collected, organized, and stored for reference and reuse. These products include the test design specification, separate test-case specifications, separate test procedure specifications, test data, test data generation procedures, test drivers and stubs, and the test summary report.

2.8.3 Evaluate Outputs

(1) Complete test summary report (from 2.8.2 (3))

(2) Complete, stored collection of testing products (from 2.8.2 (4))

III. CONCLUSIONS

Testing entails attempts to cause failures in order to detect faults, while debugging entails both failure analysis to locate and identify the associated faults and subsequent fault correction. Testing may need the results of debugging's failure analysis to decide on a course of action. Those actions may include the termination of testing or a request for requirements changes or fault correction. Progressively more detailed information about the nature of a test unit is found in the unit requirements documentation, the unit design documentation, and finally in the unit's implementation. As a result, the elements to be considered in testing may be built up incrementally during different periods of test activity.

For procedural language (for example, COBOL) implementations, element specification occurs in three increments. The first group is specified during the Determine activity and is based on the unit requirements documentation. The second group is specified during the Design activity and is based on the unit design (that is, algorithms and data structures) as stated in a software design description. The third group is specified during the Check activity and is based on the unit's code. For nonprocedural language (for example, report writer or sort specification languages) implementations, specification occurs in two increments. The first is during the Determine activity and is based on requirements and the second is during Design and is based on the nonprocedural specification. An incremental approach permits unit testing to begin as soon as unit requirements are available and minimizes the bias introduced by detailed knowledge of the unit design and code.

REFERENCES

[1] V.R. Basili, F. Shull, and F. Lanubile, "Using Experiments to Builda Body of Knowledge," Proc. Third Int'l Performance Studies Int'l Conf., pp. 265-282, July 1999.

[2] L. Bass, P. Clements, R. Kazman, and K. Bass, Software Architecturein Practice. Addison-Wesley, 1998.

[3] A. Bertolino, SWEBOK: Guide to the Software Engineering Body of Knowledge, Guide to the Knowledge Area of Software Testing, 2004 version, chapter 5. IEEE CS, 2004.

[4] R. Chillarege, "Orthogonal Defect Classification," Handbook of Software Reliability Eng., chapter 9, Mc Graw-Hill, 1996.

[5] R.L. Glass, Building Quality Software. Prentice Hall, 1992.

[6] R.L. Glass, "Questioning the Software Engineering Unquestionables," IEEE Software, pp. 119-120, May/June 2003.

[7] R.L. Glass, I. Vessey, and V. Ramesh, "Research in Software Engineering: An Analysis of the Literature," Information and Software Technology, vol. 44, no. 8, pp. 491-506, 2002.

[8] SWEBOK: Guide to the Software Engineering Body of Knowledge, 2004 version, IEEE CS, 2004.

[9] M. Knight, "Ideas in Chemistry," A History of the Science, Athlone Press, 1992.

[10] N.A.M. Maiden and G. Rugg, "ACRE: Selecting Methods for Requirements Acquisition," Software Eng. J., vol. 11, no. 3, pp. 183-192, 1996.

[11] R.M. Needham, "Computer Methods for Classification and Grouping," The Use of Computers in Anthropology, I. Hymes, ed., pp. 345-356, Mouton, 1965.

[12] D.E. Perry, A.A. Porter, and L.G. Votta, "Empirical Studies of Software Engineering: A Roadmap," Proc. Conf. Future of Software Eng., pp. 345-355, May 2000.

[13] V. Ramesh, R.L. Glass, and I. Vessey, "Research in Computer Science: An Empirical Study," J. Systems and Software, vol. 70,nos. 1/2, pp. 165-176, 2004.

[14] P.N Robillard, "The Role of Knowledge in Software Development,"Comm. ACM, vol. 42, no. 1, pp. 87-92, Jan. 1998.

[15] S. Vegas, "A Characterisation Schema for Selecting Software Testing Techniques." PhD thesis, Facultad de Informa´tica, Universidad Polite´cnica de Madrid, http://grise.ls.fi.upm.es/docs/Sira_Vegas_PhD_Dissertation.zip, Feb. 2002.

[16] S. Vegas and V.R. Basili, "A Characterization Schema for Software Testing Techniques," Empirical Software Eng., vol. 10, pp. 437-466,2005.

[17] S. Vegas, N. Juristo, and V.R. Basili, "A Process for Identifying Relevant Information for a Repository: A Case Study for Testing Techniques," Managing Software Engineering Knowledge, chapter 10, pp. 199-230, Springer-Verlag, 2003.

[18] I. Vessey, V. Ramesh, and R.L. Glass, "A Unified Classification System for Research in the Computing Disciplines," Information and Software Technology, vol. 47, no. 4, pp. 245-255, 2005.

[19] W.G. Vincenti, What Engineers Know and How They Know It. The Johns Hopkins Univ. Press, 1990.

[20] C.R. Woese, "Bacterial Evolution," Microbiological Rev., vol. 51,pp. 221-271, 1987.

[21] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," ACM Computing Surveys, vol. 29, no. 4, pp. 366-427, Dec. 1997.