Volume 7, Issue 2
May 2014
ISSN: 1946-1836

JOURNAL OF INFORMATION SYSTEMS APPLIED RESEARCH

In this issue:

The **Journal of Information Systems Applied Research** (JISAR) is a double-blind peer-reviewed academic journal published by **EDSIG,** the Education Special Interest Group of AITP, the Association of Information Technology Professionals (Chicago, Illinois). Publishing frequency is currently quarterly. The first date of publication is December 1, 2008.

JISAR is published online (http://jisar.org) in connection with CONISAR, the Conference on Information Systems Applied Research, which is also double-blind peer reviewed. Our sister publication, the Proceedings of CONISAR, features all papers, panels, workshops, and presentations from the conference. (http://conisar.org)

The journal acceptance review process involves a minimum of three double-blind peer reviews, where both the reviewer is not aware of the identities of the authors and the authors are not aware of the identities of the reviewers. The initial reviews happen before the conference. At that point papers are divided into award papers (top 15%), other journal papers (top 30%), unsettled papers, and non-journal papers. The unsettled papers are subjected to a second round of blind peer review to establish whether they will be accepted to the journal or not. Those papers that are deemed of sufficient quality are accepted for publication in the JISAR journal. Currently the target acceptance rate for the journal is about 40%.

Questions should be addressed to the editor at editor@jisar.org or the publisher at publisher@jisar.org.

JOURNAL OF INFORMATION SYSTEMS APPLIED RESEARCH

Editors

JISAR Editorial Board

Taxonomy of Common Software Testing Terminology: Framework for Key Software Engineering Testing Concepts

Robert F. Roggio
broggio@unf.edu

Jamie S. Gordon
jamie.s.gordon@unf.edu

School of Computing
University of North Florida
Jacksonville, FL 32224, United States

James R. Comer
j.comer@tcu.edu
Computer Science Department
Texas Christian University
Fort Worth, TX 76129, United States

Abstract

Most accredited computing programs have at least a single course addressing a software development process. These courses typically include a discussion of fundamental concepts and terminology that includes software testing. While many key concepts are in common use, terms describing testing are often misunderstood, misused, and misguided. The purpose of this paper is to provide a framework for commonly used and misused terminology central to software testing, and also to demonstrate their application in three common classes of testing: static and dynamic testing, black box and white box testing, and verification, validation, and acceptance testing.

Keywords: software testing, static and dynamic testing, black box and white box testing.

- **SOFTWARE TESTING**

Background

The term, software testing, often evokes conflicting understandings of what is meant. What is being tested, what is a test, who performs the tests, and what is a "tester"? Additionally, what is the difference between a program having a fault, or error, or failure, or defect, and what are the various kinds of tests and what are their similarities and differences? The authors of this paper feel that a basic understanding of these principals is essential in order to provide a framework of terminology when software engineers – or, for that matter, any stakeholder, discusses the subject. Is it possible to talk about an essential activity, such as testing, such that all participants have a consistent understanding of the meaning? Sadly, rarely is this the case, as evidenced by Naik and Tripathy, Galin, and

others. (Niak & Tripathy, 2008) (Galin, 2004) (Juran, 2000) It seems as if one must define context before positive conversation may ensue. Thus, the effort to develop a common base of understanding appears to have merit.
Interestingly, the importance of a paper on essential concepts arose during development of another paper that sought to address differences between traditional testing procedures and object-oriented testing procedures. While discussing the subject of testing, the authors noted different understandings, and perceptions, of many commonly used terms. Humbling as it was, this was the reality that prompted the development of the current paper.

Definition of Software Testing
Software testing is a verification process for the assessment of software quality and a process for achieving that quality (Naik & Tripathy, 2013). Interestingly, software testing is used to support the interests of all stakeholders of an application. In particular, software testing is essential for:

- end-users to determine whether developed or otherwise maintained software meets specifications,
- developers to ensure that the code successfully implements a credible design,
- designers to ensure that their solution is one that meets specifications,
- and, to testers, to ensure that products to be delivered do indeed meet the client's needs.

Moreover, stakeholders include:

- customer service representatives who are often charged with responding to clients who 'call' to communicate a malfunction,
- and, to administration and finance individuals who may bill clients for software provided.

The list is endless and all have a vested interest in what is called - 'testing.'

Given this backdrop, it should be clear that different levels of testing need to be done by various stakeholders at different times (during or subsequent to development). To do so requires that procedures be designed to uncover issues - all with various views of outcomes. Thus, in order to frame this paper, the authors have limited the treatment of testing to those stakeholders whose main concern is the design, implementation (programming), and end user testing.
Please also note that while the categories are indeed different in many respects and hold different meanings for different stakeholders, there is considerable overlap. The specific workplace for software development will no doubt have its own vocabulary in addressing the world of software testing. To begin, it is important to establish a basic set of definitions.

- **TESTING TERMINOLOBY**

Terms
Four useful and related terms, are frequently encountered when dealing with events that occur when software fails to perform as expected (Niak & Tripathy, 2008). References to these terms: ***failure***, ***error***, ***fault***, and ***defect*** are common in the industry; yet, unfortunately, although their means are related, they have different interpretations among practitioners. As an overview:

- A ***failure*** is defined as a behavior exhibited by a system that does not match what has been described in specifications.
- An ***error*** is an incorrect system state which could lead to a failure.
- A ***fault*** is the cause of an error. In general a fault leads to an error which leads to a failure, although not strictly so (Naik & Tripathy, 2013).
- A ***defect***, also according to Niak & Tripathy, refers to a design issue that leads to faults, although this is not as strict a definition (Niak & Tripathy, 2008).

Similar to Niak and Tripathy's terminology framework may be found in Galin. (Galin, 2004) His approach is very similar to that of Niak and Tripathy. Stressing that as practitioners we are mainly interested in software failures that disrupt or interrupt the use of software, he asserts that we must examine the relationship between software faults and failures. (Galin, 2004)

Galin begins with the simplest term, **software error** and offers that this can be a simple grammatical error in a line of code or a logical error in carrying out one or more of the client's requirements. But, once stated, Galin continues to point out that not all software errors become **software faults**. A software error may indeed cause improper functioning of the software in general or in a specific application but in other instances, the error may not cause a problem in the software as a whole; sometimes "part of these cases … the fault may be corrected or "neutralized" by subsequent code lines." (Galin, 2004)

Galin goes on to assert that we are interested in the relationship between software faults and software failures. Recognizing that not all software faults end up as software failures, he points out that a **software failure** occurs only when it is "activated." Thus in many executions of a piece of software, the software fault is never discovered because specific software executions do not activate the software fault. Of course, then, in these instances, no software failure is discovered.

Galin captures his approach to software errors, faults, and failures nicely in Figure 1.

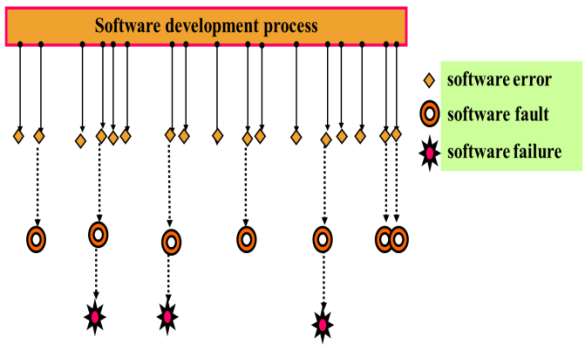

Figure 1 Software Errors, Software Faults and Software Failures (Galin, 2004)

Still others have different 'takes' on these terms. Walia and Carver state that an **error** is a mistake in the human thought process while trying to understand given information, solve problems or use methods and tools. (Wallia & Carver, 2012) **Software faults** are defined by IEEE as "an incorrect step, process, or data definition in computer programs." Favaro and her colleagues state that a **software failure** is "the inability of code to perform its required function within specified performance requirements." (Favaro, et al, 2013)

Down to Earth Examples

Let's consider a very simple example to illustrate these differences. Consider a specification that requires a very basic computation such as distance = rate * time. This is simple enough. This is a basic formula given in physical science 101. Algebraically, solving for rate would be defined as rate = distance / time. Applying this relationship to an automated solution designed to compute distance as a function of rate and time, we can address the standard definitions more closely.

Defect

Starting with design, perhaps the formula is erroneously misunderstood and designed as distance = rate + time (vice distance = rate * time). Clearly, if coded incorrectly, the resulting outputs would likely produce what might appear as a reasonable result; that is, until software testing is undertaken. A software developer, tester, end user, analyst, etc. might discover that the answers are incorrect in specific test cases. The defect is in the design. The formula is incorrect. The 'solution' to the requirement is incorrectly specified and designed, and although the program may well run to, end of job, the defect is (hopefully) clear.

Stutzke integrates treatment of these terms by defining a defect as "An observation of incorrect behavior caused by a failure or detection of a fault." (Stutzke, 2005) The failure in this case is an incorrect result (discovered during testing) and is the manifestation of a fault or incorrect result; Stutzke goes on to point out that the fault is an error that could cause a program to fail or potential failure. He defines error as the amount by which the result is incorrect.

Error

The failure was the production of an incorrect system state: the producing of an incorrect value. The state of the system is now incorrect. For the distance = rate + time, the resultant state of distance is incorrect.

Stutzke cites that an error can be the simple result of a misunderstanding. He cites the fault-tolerance discipline that addresses these terms: in Fault Tolerance the discipline distinguishes between human action (a mistake), the manifestation or result of the mistake (hardware or software fault), the specific result of the fault (a failure), and the amount by which the result is incorrect (the error). Again, the defect is the observation caused by the failure (event) or detection of a fault.

Fault

The fault is the cause of the error which was a design defect leading to this fault. A fault led to a failure, the incorrect result discovered by testing. The fault here is implementing the design defect (distance = rate + time) which manifests itself in the detection of a failure.

Failure
It is commonplace to say the fault (cause of the error) led to a failure, where the failure in our example is the behavior of the application (adding rate to time in lieu of multiplying rate by time) during run time to produce the expected result. The production of an unexpected result points out a fault.

Conclusion
While all this might at first glance appear to be unimportant, the differences between discovering errors in design as opposed to discovering failures in implementation are quite significant from a cost perspective. Thus, realizing that a software defect is a design issue vis' a vis' one associated with implementation can affect the overall development and testing processes and can negatively impact the understanding of what the engineering of software really means.

It is important for a software engineer to have a commonly accepted set of terminology for communications, which is central to modern software development practices. To successfully communicate, we need a common language. Precision in identifying root causes of software errors (design defect, implementation fault, etc.) is essential to good software development practices so that proper best practices can appropriately address the wide-ranging origins of software errors.

- **TYPES OF TESTING**

Software testing can be classified into many subcategories, often depending on one's perspective and often based on terms in common use in one's working environment. According to *Software Test Engineering @ Microsoft*, a number of test categories arises from the breaking down of work items in a workplace. This paper suggests a list that includes functional testing, specification testing, security testing, regression testing, automation testing and beta testing. The paper cites that the list is intentionally incomplete and requests supplements to the list. One response included unit/API testing, acceptance testing, stress/load testing, performance benchmark testing, and release testing. Still another response included performance testing, stress testing, interoperability testing, conformance testing, static testing, and maintainability testing. (blogs.msdn.com/b/chappell/archive/2004/03/24/95718.aspx) This diversity clearly supports that there are simply many types of testing, and that types of tests appear to be centered on one's focus or interest. Given this, the authors have taken liberty to divide software testing into a few different broad categories to include static and dynamic testing, white-box and black-box testing, and verification and validation testing.

Static and Dynamic Testing

Static Testing
In general, static testing can be performed on both documentation (specification documents, design documents, etc.) and source code (pseudo-code, source programs, scripts, etc.). (Johnson, 1996) Pressman discusses static testing tools as those that embody tools used to test code, specialized testing languages, and requirements-based testing tools. (Pressman, 1997) Code-based testing tools process source code (or a program description language) as the primary input and undertakes several analyses resulting in generation of test cases. They also identify a number of poor programming practices (identifiers defined and not used; incompatibilities between definition and use of attributes and more). Specialized testing languages enable a software developer to develop detailed test specifications and describe each test case and the logistics needed for its execution. Requirements-based testing tools inspect user requirements to suggest test cases or classes of tests to exercise the requirements. All of these are accommodated without any execution of code.

Certainly careful static analysis of documentation can reveal many issues. Defects may be discovered in the specification and/or design stages as well, without any need for any actual program development and subsequent execution. For example, Structure Charts for procedural development and many UML diagrams (class diagrams, object diagrams, subsystem and package diagrams, sequence and communications diagrams) are all candidates for testing without any 'program' execution. All of these may well lead to the discovery of defects by observing how, for example, a sequence of object responsibilities (methods) are invoked in a sequence diagram used to capture the procedural flow in a scenario captured from a use case. Such an analysis might lead to the movement of responsibilities from one object to another in the interests of good design.

Consider static analysis of requirements. Static analysis of requirements can take place by visually inspecting the specification document

and test for sufficiency, necessity, feasibility, completeness, and measurability. While indeed we are reviewing specifications, tests of this nature are static and do test the specifications.

Consider static analysis in design. Consider then a simple sequence diagram that is used to show the collaboration of objects and their method calls that are 'designed' to implement a scenario in a use case. In developing the sequence diagram, it is reasonably easy to discover that responsibilities assigned to an object, that is, methods, are poorly placed. For example, good cohesive design encourages the incidence of attributes and the methods that process these attributes to be located within the same object. In developing the sequence diagram, poor design can readily show that the methods and the data are not together. This kind of static test can easily result in modifying the object design so as to improve cohesion and hence provide for a better design. Again, this is a simple static test in tracing through a scenario in its accommodation in OO design. Additional static design tests include viewing, for example, UML diagrams to determine degree of coupling, object obsolescence, candidates for dividing and conquering complex objects and more.

Traditionally, static testing often addresses programming and deals with analysis of written code through walk-throughs and/or code inspections that result in algorithm analysis, and syntax or semantic checks (Nail and Tripathy, 2008). However, no actual execution is done in this stage as 'static' testing implies. It is purely investigation of the structure of code and hypothesizing what might happen at run-time. Many compilers and integrated development environments (IDE's) are designed to greatly assist programmers with this process. An example of static testing in programming is running a static analyzer looking for unreachable code, or 'dead code' that often arises in programs that have been modified over the years. In cases where programs have been maintained over a period of many years, they may have undergone many changes. Oftentimes a programmer must surgically delve into existing code to add features or correct errors without corrupting the existing functionality. Usually the programmer is given insufficient time to do a thorough analysis and must modify the program for a redeployment within often severely imposed time constraints. The programmer must react quickly and precisely and is not afforded the time he/she might need in order to undertake a thorough analysis.

A static view of code may reveal shortcomings via visual 'smells' that suggests the need for refactoring. Code smells, in and of themselves, are not bugs and do not necessarily lead to a non-functioning program. They may, however indicate weaknesses in design and may lead to code failure in the future. Long, multi-functional classes, methods with large numbers of parameters and options and many more smells suggested by Fowler may well suggest refactoring. (Fowler, 2012)

Dynamic Testing

In contrast to static testing, dynamic testing involves execution of a design or written code (m*ost dynamic testing is done on code).*

Pressman states that "dynamic testing tools interact with an executing program checking path coverage, testing assertions about the values of specific variables, and otherwise instrumenting the execution flow of the program." (Pressman, 1997) Niak and Tripathy state that dynamic testing involves analysis of behavioral and performance of the design and code (Naik and Tripathy, 2008), while Schulmeyer and MacKenzie cite that dynamic analysis methods involve the execution of a development activity designed to "detect errors by analyzing the response of a product to sets of input data." (Schulmeyer and MacKenzie, 2000) Clearly, desired outputs and/or ranges of output must be known ahead of time. Too, testing is the most frequent dynamic analysis activity. It is interesting to note that while dynamic testing is most often associated with code execution, dynamic testing can be applied during prototyping – especially during software requirements verification and validation. While the precise outputs are likely not always known, it can sometimes be determined that the system response to an input meets system requirements.

To show how broadly the principles of dynamic testing extend, Schulmeyer includes the running of static analysis tools as part of what he calls *Implementation Verification* and the running of dynamic analysis tools as part of *Validation*. We will concentrate on dynamic testing of code.
Dynamic Testing of Code represents a very large and encompassing set of tools for software testing. As an example of practical dynamic testing, consider the following real-world example that formed a part of dynamic testing of major programs.

Consider a program called *Percent Execute*; a program used long ago in the U.S. Air Force. Its purpose was to monitor the run-time behavior of programs as part of the testing activities before deployment of the software. The purpose of Percent Execute was simply to discover how much (literally) of a program was actually executed given an input dataset. Given specific inputs (and several different sets of inputs), just what portions of a program were / were not executed? Clearly, different input data would cause different execution paths to be executed. The methodology called for a source program to be instrumented with source code probes (discussed later) that were inserted into every program unit (method, function, paragraph, module, etc.). Afterwards, the program was re-compiled and executed with carefully designed sets of input data to determine what parts of the program were being executed. Dynamic testing clearly (and often) revealed that key parts of the executable code were not exercised. This was disturbing given that essential edits were discovered to go unexecuted but were assumed to have been. For example, edits in financial programs to ensure financial and data integrity were sometimes simply not executed for some input data. Without dynamic testing, making this determination would have been very difficult and would have involved inspecting output files record by record – a very labor-intensive process. Running the instrumented program might reveal that 30% or 40% of a program was actually executed (specific code segments executed were reported). Naturally, all segments of the program were not expected to run for all input test data sets, as the program logic accommodated. But for specific sets of inputs, key parts of the programs were expected to run.

This is a great example of dynamic testing - run the program and monitor its run time behavior. Testing such programs dynamically pointed out serious defects (design issues implemented in code) causing errors (production of an incorrect state); the fault was the cause of the error (logical design resulting from poor design) and the resulting failure arose from the resultant behavior (manifestation of the fault(s) through reports generated by summary data produced by the instrumented program executed upon program completion).

(The source code 'probes' are merely integer counters in a single array. Each programming construct (function, paragraph, method, etc.) was instrumented to add 1 to a counter in the array that was associated with that construct. Upon conclusion of the program, the value of each array element represented the total number of times that construct was executed, ranging from zero to a higher number. A function was appended to the program and was executed just prior to normal program termination. This code accessed the array and displayed the numbers of times each programming construct was executed.)

Most modern IDEs offer the ability to monitor variables and their changing values during runtime. Students using Eclipse, NetBeans, or a number of other popular IDEs are familiar with these features that can track program execution allowing one to step through a program one statement at a time and observe how the values of attributes change. These are further examples of dynamic testing and support Stutzke's contention that dynamic analysis is the process of "…operating a system or component under controlled conditions to collect measurements to determine and evaluate the characteristics and performance of the system or component." (Stutzke, 2005)

Black-Box and White-Box Testing

Another grouping of test categories, not mutually exclusive from static and dynamic testing, is black-box and white-box testing. When creating test cases, various sources need to be considered such as specifications captured, perhaps, from use cases or user stories, design documents captured in structure charts or UML diagrams, and actual source code or pseudo-code, captured in a wide range of IDEs. Also, there is available documentation.

Pressman sums up the differences between black-box and white-box testing rather nicely: "Any engineering product (and most other things) can be tested in one of two ways: 1) knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational, at the same time searching for errors in each function; 2) knowing the internal workings of a product, tests can be conducted to ensure that 'all gears mesh.', that is, that internal operation performs according to specification and all internal components have been adequately exercised. The first test approach is called *black-box testing* and the second, *white-box testing."* (Pressman, 1997)

White-box Testing
(Sometimes called structural or glass-box) testing is done through examination and knowledge of source code. White-box testing examines execution flow through algorithms via 'coverage measures' such as examination of statement coverage, path coverage and branch coverage investigations. White-box testing, in its many forms, monitors the internals of a program and tracks and determines 'how' the program executes, how much of the code is being exercised. Also considered is how many tests through an algorithm are necessary to assure a minimal or acceptable level of testing, what constitutes a minimal set of tests needed to assure a high level of reliability, how 'robust' the program must be and similar tests.

White-box testing considers many program / system execution characteristics. Consider this more closely. Recognizing that one can never assert that a program is error-free, white-box testing addresses factors such as how many edits need to be included in the code to assure an acceptable level of reliability? In particular, is the program one that deals with safety-critical applications, aircraft or weaponry instrumentation, financial systems, or health systems? How much code must be added and tested to assure acceptable levels of reliability and how much reliability is really needed? These are a few of the factors whose answers are used to determine the degree to which edits and other checks are included in both the design and implementation to achieve desired levels of reliability, robustness, and fault tolerance. These are all execution time tests and are verified during run time.

In white-box testing, there needs to be some assurance that code that must be executed is indeed being executed via tests with specific input data. In a way, it is close to but involves both static and dynamic testing. In dynamic testing, test results can point out programming anomalies or areas not executed or time spent in program components (perhaps implying that these are candidates for optimization). But white-box testing (in the coding sense) goes deeply into the internals of the program, to the code itself. The testing yields significant analyses citing statements executed or branches not taken, or execution paths not executed and similar low level information to the developer. The critical thinking is that white-box testing involves the detailed execution analysis of the program's guts; that is, statements, branches, paths, function calls, method calls, and more.
While dynamic testing is used to collect measurements and evaluate characteristics and performance of a component, and can be seen as part of validation, white-box testing, on the other hand, is at the lowest level and is needed for the developers (analyst and programmers) to consider in assuring effective dynamic testing.

Black Box Testing
In contrast to white-box testing is black-box testing, sometimes referred to as end-user testing. In black-box testing, the internals of program execution are not an issue; rather, key concerns center on the production of the correct output given specific inputs. Are the results timely—and accurate? And are all of the requirements accommodated?

In black-box testing, the program is viewed as a black box. The program must read in the inputs, process the data, and check the outputs. While this sounds simple, it is not. Certainly running the test is easy, but the design of suitable test cases may well be an onerous task as a host of carefully designed sets of tests must be generated, oftentimes including boundary testing, stress testing, regression testing, functional testing, and other related black-box testing issues. All of these tests are designed to determine if the application produces the correct outputs given a variety of inputs that exercise / test both the functional and non-functional requirements (Kulak and Guiney, 2004).

Testing requires both functionality (outputs produced given inputs) and non-functional testing (system loading, reliability, robustness, scalability, portability, maintainability, security and more. Black-box testing is often done as part of validation by end users, hence the reason for it sometimes being referred to as end-user testing.

Black-box testing is done without knowledge of the internal workings of code (Turner and Robson, 1993) Instead test cases are derived from the specifications or design or any other documentation that implies functionality. In this way, black-box testing is only concerned with what can be generated from running the application. Defects are often discovered in black-box testing and may be traced back to design issues or perhaps implementation issues. Failures (behavioral issues; the producing of unintended results) may also be readily observed via black-box testing. In contrast, the cause(s) of

the error (fault) and the producing of an incorrect system state (error) are more typically discovered via white-box testing.

Verification, Validation and Acceptance Testing

These tests reflect still another category of testing – again, not mutually exclusive of static and dynamic testing and black-box and white-box testing - using terms often in common use with different stakeholders. Verification is often combined with another software engineering concept known as validation. These are two different types of testing with different goals in mind, unlike static and dynamic testing which both seek to find faults in code and defects in design on the development side.

Stutzke sums up the differences between verification and validation. He says that verification deals with evaluation of products in a given [development] activity "to determine both correctness and consistency with respect to the products and standards provided as input to that specific activity." Verification ensures that "you have built it right." In contrast, validation confirms that the product, as provided (or as it will be provided) will fulfill its intended use. Validation ensures that "you built the right thing." (Stuzke, 2005) In more detail, consider the following elaboration of these definitions.

Verification testing is the pursuit of establishing that a particular phase of a software system has satisfied the requirements which had been decided upon before embarking on that phase (Naik and Tripathy, 2013) Thus, verification testing is typically white-box testing but may also include black-box testing. Essentially, verification is done by the developers or maintainers of software to ensure that the software meets requirements This is often the activity undertaken by software developers typically during unit testing. It follows from this that although verification testing is generally white-box testing, clearly the developer is interested in producing correct outputs given specific inputs. Specifically, the product is built right.

Validation testing is done to assure that software meets the needs of those who intend to use it (Naik and Tripathy, 2013). Validation testing is, thus, often black-box testing and is concerned with ensuring functionality. Validation testing provides the customer confidence that the software system is adequate for its intended use. Essentially successful validation testing provides assurance to the user that their expectations have been met. Customers typically undertake validation exercises to ensure the right thing was built.

While verification testing is used to eliminate defects and faults that cause error states and visible failures, validation testing shows that there are no failures. Stated equivalently, in verification: programmer runs unit tests against specifications and eliminates defects and faults causing error states and visible failures; in validation: end user runs tests to determine if specific inputs result in specific outputs. Clients / end-users run tests to ensure no failures are experienced.

One sometimes sees the term, acceptance testing and acceptance criteria. Acceptance criteria are often defined by the designers in the hopes that satisfying the criteria adequately demonstrates to the user that their needs have been met. Also acceptance testing is designed to help the end-user gain confidence in the code.

4. CONCLUSIONS

The paper has provided definitions of fault, errors, failures and defects with specific examples to provide clarity in their use. While the paper did not propose a study to verify the approaches offered by researchers in the literature review, value lies in establishing a solid basis of definition and use of these commonly misunderstood and misused key definitions both in the workplace and in the classroom. Practitioners and students must use precise definitions when referring to defects, errors, faults, and failures.

The authors have also applied these terms to three major categories of testing: static and dynamic testing, white-box and black-box testing, and verification, validation, and acceptance testing. While there are other categories of testing that are often unique to specific software development methodologies, most of these categories can easily fit within a framework of the three testing categories provided.

5. REFERENCES

Favaro, Francesca, M., David Jackson, and Joseph Saleh, *Software Contributions to Aircraft Adverse Events: Case Studies and Analyses of Recurrent Accident Patterns and Failure Mechanisms*, Reliability Engineering & System Safety 113, May 2013.

Fowler, Martin, contributions by: Kent Beck, John Brant, William Opdyke, Don Roberts, *Refactoring: Improving the Design of Existing Code*, Addison- Wesley, 2012, ISBN: 0-201-48567-2.

Galin, Daniel, *Software Quality Assurance,* Pearson / Addison Wesley, 2004 (ISBN 0-201-70945-7), Chapter 2, What is Software Quality?

Juran, J.M. and A. Blanton Godfrey, *Juran's Quality Control Handbook,* 5th edition, McGraw-Hill, New York ISBN-13: 978-0071165396, 2000.

Kulak, Daryl and Eamonn Guiney, *Use Cases: Requirements in Context,* Addison Wesley, 2004, ISBN: 0-321-15498-3.

Morris S. Johnson, Jr., "A Survey of Testing Techniques for Object-Oriented Systems," Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '96).

Naik, K., & Tripathy, P., *Software Testing And Quality Assurance: Theory And Practice*, John Wiley & Sons, 2008. p. 7-27.

Pressman, Roger, Software Engineering: A Practitioner's Approach, McGraw-Hill, 1997, ISBN 0-07-052182-4.

Schulmeyer, C. Gordon and Garth R. MacKenzie, *Verification & Validation of Modern Software –Intensive Systems*, Prentice-Hall PTR, 2000 ISBN: 0-13-020584-2.

Stutzke, Richard D., Estimating Software-Intensive Systems, Pearson Education Inc., 2005 ISBN 0-201-70312-2.

Turner, C.D.; Robson, D.J., "The State-Based Testing of Object-Oriented Programming Conference on Software Maintenance, CSM-93, Proceedings, 1993 ISBN 0- 8186-4600-4.

Walia, Gursimran S., and Jeffrey C. Carver, *Using Error Abstraction and Classification to Improve Requirement Quality: Conclusions from a Family of Four Empirical Studies,* Springer Science + Business Media, LLC 2012.

Microsoft vs Apple: Which is Great by Choice?

James A. Sena
jsena@calpoly.edu

Eric Olsen
eolsen@calpoly.edu

Orfalea College of Business,
California Polytechnic State University
San Luis Obispo, CA 93407, USA

Abstract

We set out to examine the performance and practices of Microsoft and Apple since the Collins *Great by Choice* [GBC] study. In *Great by Choice*, Collins and Hansen developed an explanatory framework based on their comparative study of seven pairs of high performing companies and matched comparison companies. One of these pairs was Microsoft and Apple. For these two, we examined financial performance for the eleven-year GBC comparison period (1991 - 2001) and the analysis period (2002 - 2012). Using this financial analysis, we developed and examined research questions about whether Apple and Microsoft were or were not employing the GBC practices over our research period. Although GBC seemed to have sound advice for companies, our findings were mixed. During the research period, Apple went from under-performing to outperforming Microsoft. However, the causal relationship of the GBC practices to the financial reversal is not clear. Both Microsoft and Apple varied in their use of the GBC practices over the research period.

Keywords: Leadership, management best practices, practice versus performance, comparison case studies, *Great by Choice*, Apple, Microsoft

1. INTRODUCTION

The rivalry between Microsoft and Apple began when Microsoft chose to license its operating system to different computer manufacturers. This resulted in several different machines running Windows while Apple chose to keep its operating system to itself and to construct its own hardware. Today this rivalry is still evident in Apple and its Mac OS, and Microsoft and Windows 8. At Apple, the one-size-fits-all approach emphasizes a particular product. Microsoft has over 100 Windows 8 devices marketed. This exemplifies the strategies of Microsoft and Apple in a nutshell—Apple limits your choices; Microsoft multiplies them. For Microsoft, the level of support and technical help may suffer. Pros and cons aside, the contrasting strategies between the two companies will continue to define the significant differences between Microsoft's and Apple's business results (Gilbert 2012).

In a series of works by Collins, and then with Hansen, the authors sought to establish principles and practices that were unique to successful companies. In *Great by Choice* [GBC], they examined paired companies over an extended period until 2002. One of these pairs was Microsoft and Apple. Collins and Hansen identified Microsoft as one of the companies that chose to

be "great" by implementing the GBC practices they identified, whereas Apple did not. Their GBC principles and practices applied to companies within their period of analysis (up to 2002), but what about beyond? Collins makes the case that *falling* from greatness did not contradict his conclusions because during the dynastic period the companies were engaging in those practices while financially *great*. His assumption was that the companies are no longer "great" because they were no longer using the practices. In this paper, we examine Microsoft and Apple to determine if "great" performance is explained by the application of GBC practices or a reduction in performance is explained by discontinuing using those practices that purportedly made them "great." Perhaps the answer is somewhere in-between. We begin with a review of the conclusions and practices from Collins' previous works (Table 1).

Title	Reference	Objective
Built to Last	Collins, Jim and Porras, Jerry (2001)	Identify practices that enable the transformation from a mediocre (good) company to a great company.
Good to Great	Collins, Jim (2001)	Identify practices of great companies.
How the Mighty Fall	Collins, Jim (2009)	Identify mechanisms that cause once great companies to fail.
Good to Great and the Social Sector	Collins, Jim (2011)	Identify practices of great companies in the social sector.
Great by Choice	Collins, Jim and Hansen, Mortenson (2011)	Uncertainty, chaos luck -- why some thrive despite them all

Table 1: Quick reference to Collins and group series of books

In *Built to Last*, Collins described the practices of great companies. In *Good to Great,* Collins showed how "great" companies evolve over time and how long-term sustained performance could be engineered into the enterprise. He identified a set of elite companies that made the transition from mediocre to extraordinary results and sustained those results for at least fifteen years. After the transition, the good to great companies generated cumulative stock returns that beat the overall stock market by an average of seven times in fifteen years, better than twice the results delivered by a composite index of the world's greatest companies.

Subsequent to *Good to Great,* Collins and Hansen extended their research work in GBC by examining a set of companies that they refer to as "10x" cases. During the study period, these companies outperformed other companies in their industry by 10 times or more. One of the organizations that met their criteria was Microsoft.

These companies, specifically Microsoft in our study, started from a position of vulnerability, rose to become *great by choice* with outstanding financial performance. Microsoft did so in an unstable environment characterized by forces that were out of their control, fast moving, uncertain, and potentially harmful. Collins matched companies with firms that failed to become great in the same extreme environments, specifically Apple in our study. They used the distinction between winners and "also-rans" to uncover the distinguishing practices that allow some to thrive in uncertainty.

In this paper, we replicated the methodology presented in Collins and Hansen's GBC over the end of their period of examination (1991 – 2001) and extended it into a second period (2002 - 2012). Our goal is to determine if the practices developed and related performance that this particular pair of companies demonstrated in their dynastic period continued (or increased) or discontinued (or decreased) based on financial and practitioner research as formulated in GBC.

We set out to examine the financial performance and practices, Microsoft and Apple, from Collins' GBC study. We examined their financial performance for the eleven-year GBC comparison period (1991-2001) and the research period (2002 - 2012). We used these financial analyses along with the qualitative practice analysis to develop and evaluate research questions as to whether Apple and Microsoft were or were not employing the GBC practices. In the sections that follow, we describe our financial and qualitative practice analyses and conclusions.

2. FINANCIAL PERFORMANCE ANALYSIS

GBC Procedure

Collins and Hansen selected and compared companies based on financial performance from 1972 to 2002. They observed that the true test of a company's ability to handle a turbulent business environment was accomplished by comparing like companies operating in the same environment. Table 2 and Figure 1 (see Appendix) show the Total Price Return percentage for the GBC and comparison company (Microsoft and Apple) for the two periods: the last 11 years of the GBC period (1991-2001) and the 11 years since (2002-2012). The first test we performed was to verify that the Microsoft was still financially out performing Apple in the last eleven years of the GBC comparison period. We examined how the two companies performed in comparison to the

Standard and Poor's 500 (S&P 500) and to each other. Microsoft performed 12.8 times better than the S&P 500. Apple did much worse than the S&P 500. We looked at the Microsoft-Apple pairing. It showed that the "great" company, Microsoft, outperformed Apple by a factor of 42.7 in this period.

GBC-Redux. We looked at the 11-year update period 2002-2012. Apple went from being worse than the general market to 29.9 times better and 48.8 times better than Microsoft. To test Collins' and Hansen's proposition that GBC practices lead to "great" financial performance and the lack of these same practices leads to worse performance, we would expect that Apple should show evidence of using GBC practices during the update period. Microsoft should show a decrease in GBC practice usage due to their significantly decreased performance relative to the S&P 500 and Apple.

Another financial performance check we performed was to examine the companies' current ratio and debt-to-equity ratio. The data are included in Table 3 and Figures 2 and 3. This data is comparable to the data provided in GBC that concluded that the "great" companies hold current ratios better than comparisons 72% of the time and have better total debt-to-equity ratios 64% of the time. The analysis concurs with Collins and Hansen for the end of the GBC period. Microsoft outperformed Apple on both measures.

However, in the update period, Microsoft's average current ratio, though 13% better than Apple's, reduced by 23%, whereas Apple only reduced by 1%. Microsoft's debt-to-equity ratio was much worse in the subsequent 11 years. Apple's average debt-to-equity reduced by a factor of 30 and is now 6 times less than Microsoft's. This data provides the basis for the research proposition that Apple used GBC practice in the update period and Microsoft did not.

Based on the financial analysis we constructed two research propositions related to the GBC practices. These research propositions, shown in Table 4, depict expectations for GBC practice or lack of practice given our financial analysis of the update period.

3. PRACTICE OBSERVATIONS

In GBC, 10X leaders were both "disciplined" and "creative," "prudent" and "bold"—they went fast when they must, but slow when they could—they were consistent, yet open to change. According to Collins and Hansen, successful companies were often not as innovative as the control companies. In some cases, they were actually less innovative. Rather, they managed to "scale innovation," introducing changes gradually, then moving quickly to capitalize on those that showed promise. The successful companies were not necessarily the most likely to adopt internal changes as a response to a changing environment. "The 10X companies changed less in reaction to their changing world than the comparison cases" (Murray 2011). Table 5 presents the GBC practices.

Collins and Hansen began the process of identifying and further explicating the unique factors and variables that differentiate GBC companies. One of the most significant differences is the quality and nature of leadership. We used these practice descriptions, and those in GBC, to identify practice usage by Microsoft and Apple. To better understand the context and business environment we considered a number of other factors that complemented and correlated with the GBC practices. These included counts by year of acquisitions and divestitures; joint ventures; infrastructure incidents; significant personnel actions; philanthropic activity; litigation; financial announcements; and recognitions/presentations. These factors were particularly helpful in analyzing and assigning ratings in situations where there was considerable activity. Examples are litigation dealing with the acquisition activity of Microsoft and the personnel changes and leadership ratings of Apple.

We performed a comprehensive practice analysis of Microsoft and Apple depicted in Table 4. To verify the research questions we examined an comprehensive set of sources and references. Of note, there was neither uniform nor consistent availability of company data. For example, Wikipedia was somewhat useful for providing a ready supply of current links and sources. For Microsoft, the company websites overwhelmed us with data. We visited both company websites and examined their financial declarations for the period of study. There was much variability in the form and content of reporting. Media and press releases were quite useful—this involved sifting through two to three hundred references for each of the years. Another source we used was *Brint.com*, a specialized business search engine. This source allowed us to consider academic journals, business magazines and newspapers,

and industry publications while deploying various search filters.

Overall, we rated both companies as shown in Table 4 on the four practices: Fanatic Discipline, Productive Paranoia, Empirical Creativity, and Level 5 Ambition and noted whether the data supports or does not support the research proposition. We scored articles and incidents using GBC discussions and descriptions. The scores were converted into a 7-point scale ranging from "strongly disagree that the practice is being used" (1) to "strongly agree the practice is being used" (7). If the practice rating supports our research question on practice usage based on financial performance (Table 5), then our analysis supports Collins' and Hansen's work in GBC.

In the remainder of this paper, we present a case description of our analysis and conclusions with respect to GBC practice usage by Apple and Microsoft in the period from 2002 to 2013. At the conclusion of the paper, we summarize our findings and make recommendations for application and future research.

Research Question 1: Did Microsoft Stop Using GBC Practices?

Microsoft is the leading software producer worldwide (van Kotten 2011). As of 2012, they dominate both the PC operating systems and office suite markets. The company also produces a wide range of other software for desktops and servers. They are involved in areas including internet search (with Bing); the video game industry (with the Xbox and Xbox 360 consoles); the digital services market (through MSN); and mobile phones (via the Windows Phone OS). In June 2012, Microsoft announced that it would be entering the PC vendor market for the first time with the launch of the Microsoft Surface tablet computer**.**

The GBC study ended in 2001; in that period, Microsoft met the "great" criteria. In 2001, Microsoft was still firing on all cylinders. However, this was not always an accurate representation, especially in the latter part of the update period 2000 – 2012. Microsoft's fiscal year 2006 revenue was more than double Apple's FY '06 revenue: $44.3 billion to $19.3 billion. What has happened since? Apple's revenues have more than tripled while Microsoft's have grown by less than 50%. Microsoft still employs substantially more people than Apple does, although the size of Microsoft's workforce has dropped a bit, from 93,000 in 2009 to 89,000 in 2010. Apple's reported headcount has been rising, with a significant increase from 34,300 in 2009 to 46,600 in 2010. Apple's revenue per employee at the end of its 2010 fiscal year was substantially higher than Microsoft's: $1.4 million versus $702,000. Likewise, Apple's profits per employee were $300,429, compared with $211,236 for Microsoft (Machlis 2011).

Table 6 presents our compilation of the four practices for the update period along with other considerations that mitigate the practices ending in 2012 with respect to Microsoft. The compilation better clarifies by presenting chronologically as well as in summary form and introducing more granularity overall. Not all practices have a score for each year when there were no significant events.

Fanatic Discipline [Neutral]. To serve the needs of customers around the world and to improve the quality and usability of products in international markets, Microsoft localized many of their products. Localizing a product may involve modifying the user interface, altering dialog boxes, and translating text. Localization, although an attractive international strategy, can be a deterrent to consistency.

Microsoft has been active in acquisitions throughout its history. Over the past eleven years, they have acquired 64 companies. Table 6 showed the distribution over the eleven years of our study. Many of these acquisitions denote entries into new or developing marketing areas. Rarely is Microsoft a first mover. Microsoft often enters during the shakeout stage of the product life cycle. This is evidenced by their recent entry of a tablet into the crowded iPad/Samsung foray. Another example is their entry into the *cloud computing* market for Windows (Fried 2008) and their intent to open a chain of Microsoft-branded retail stores (Freid 2009). Over the past 20 years, Microsoft has exhibited discipline and endurance in its "not first mover" strategy.

Productive Paranoia [Somewhat]. Microsoft contracts most of their manufacturing activities to third parties. These include Xbox 360 and related games; Kinect for Xbox 360; various retail packaged software products and Microsoft hardware. Their products include some components that are available from only one or limited sources. Their Xbox 360 console and Kinect for Xbox 360 included key components supplied by a single source. The integrated central processing unit/graphics processing unit is purchased from IBM, and the supporting

embedded dynamic random access memory chips are purchased from Taiwan Semiconductor Manufacturing Company. However, they usually have multiple sources for raw materials, supplies, and components, and are often able to acquire component parts and materials on a volume discount basis (U.S. Securities Exchange Commission 2011).

As the smartphone industry boomed beginning in 2007, Microsoft struggled to keep up with its rivals Apple and Google in providing a modern smartphone operating system. As a result, in 2010, Microsoft revamped their aging flagship mobile operating system [OS], Windows Mobile, replacing it with the new Windows Phone OS. This was a change in strategy in the smartphone industry. Microsoft is now working closely with smartphone manufacturers to provide a consistent user experience. In May 2012, Microsoft released the next generation Windows 8 software designed to power devices ranging from tablets to desktop computers (AFP Relax 2012).

Empirical Creativity [Somewhat Agree]. Microsoft (Kate 2005) has long been known as a company that tightly controls all aspects of its marketing and communications with customers, business partners, analysts, and the media. In the mid section of our study, Microsoft made efforts to change its image and develop a more open marketing culture. The fact that they reached out to the media and analyst community to discuss the change was news in itself. Internally they changed the way engineering and marketing work together to create a more cohesive and seamless product development process. This process was initially used in three projects: new versions of Office, Visual Studio, and Exchange.

Most of Microsoft's software products and services are developed internally. Internal development allows them to maintain competitive advantages that come from closer technical control over their products and services (U.S. Securities Exchange Commission 2011). This also gives them the freedom to decide which modifications and enhancements are important and when they should be implemented. They strive to obtain information as early as possible about changing usage patterns and hardware advances that may affect software design. Before releasing new software platforms, they provide application vendors with a range of resources and guidelines for development, training, and testing.

Level 5 Ambition [Neutral]. When Bill Gates, Chairman of Microsoft, announced his intention to step down in July 2008, he stressed that he was not retiring but just making a transition (BBC News, 2006). Even though he no longer would be the chair in two years' time, as chairman he intended to maintain a key role in advising the firm. In 2008, he had assumed the title of chief software architect and stayed on as company chairman; Steve Ballmer took over as chief executive (U.S. Securities Exchange Commission 2011).

In the 1990s, critics began to assert that Microsoft used monopolistic business practices and anti-competitive strategies. This placed unreasonable restrictions on the use of its software. Both the U.S. Department of Justice and European Commission found the company in violation of antitrust laws. Many forms of litigation continued throughout the period of our study. There were eighteen separate incidents from the time period of 2002 to 2006.

One of Microsoft's business tactics, described by an executive as "embrace, extend and extinguish," initially embraces a competing standard or product; extends it to produce their own version which is incompatible with the standard; and, in time, extinguishes competition that does not or cannot use Microsoft's new version (Rodgers 2008). Various companies and governments sued Microsoft over this set of tactics, resulting in billions of dollars in rulings against the company. Microsoft claimed that the original strategy was not anti-competitive, but rather an exercise of its discretion to implement features it believes customers wanted.

In Research Question 1, we proposed that Microsoft stopped the use of GBC practices based on our financial analysis. However, our examination of the four practices did not provide enough evidence to confirm the proposition.

Research Question 2: Did Apple Start Using GBC Practices?

From the period of 2002 to 2012, we noted a steady progression of improvement in Apple's Fanatic Discipline and "Productive Paranoia" and a relatively stable set of "Empirical Creativity" activities. However, in "Level 5 Ambition" there was mixed evidence due to questions about Steve Jobs' performance, as well as the introduction of products such as the iPad. Table 7 depicts the four practices and the corresponding set of activities.

Fanatic Discipline [Somewhat Agree]. Apple's leadership has been pervasive (Mirchandani, The New Technology Elite: How Great Companies Optimize Both Technology Consumption and Production 2012). Traditional supply chain disciplines like managing an extended network of contract manufacturers and component suppliers are fully in force, but beyond the areas Apple has led in at least two vital ways. The first is in its advantage of the digital supply chain. By fostering the development of a secondary market in applications for its iPhone, the company has shown again (as with iTunes) that consumer product revenue growth with zero inventories is not only possible, but also repeatable. The other area in which Apple's supply chain leadership is increasingly relevant is in the retail experience. As one of a handful of extremely vertically integrated brands, Apple's retail chain achieves almost unimaginable success in its stores.

Productive Paranoia [Somewhat Agree]. Apple has built a retail store chain that is the envy of even long-time retailers (Mirchandani, The New Technology Elite: How Great Companies Optimize Both Technology Consumption and Production 2012) . It has built an elaborate global network of suppliers and contract manufacturers that has confused the traditional accounting that economists use to determine global trade. In addition to the elaborate physical supply chain, it has had to integrate the digital supply chain as iPhones are activated via iTunes at customer homes and via carriers. As it rolls out its iCloud, it has built one of the biggest data centers in the world. It has built an ecosystem of apps and games around its products at a never seen before scale. Admirably, it built its supply chain in a much more volatile industry than that of consumer products or chemicals. Of course, Apple has itself driven the high-tech industry volatility with its own pace of product introductions. Dell used to be regarded as a benchmark of efficiency with its "build-to-order" supply chain. It manufactured most of the order content and even paid in advance. Apple raised the bar by showcasing a new product, guesstimating likely demand, and tuning its supply chain day-by-day and hour-by-hour. It broke traditional rules of demand forecasting because there was little historical data from which to forecast for a version 1.0 iPod or iPhone or iPad. It balanced the risk of overproducing or increasing buffer inventory and taking write-offs versus under- producing and losing customers to the next competitive product. It took that risk time and again, and made the rest of the industry do the same. In addition, the risks are not insignificant when talking about three million iPads in their first quarter of introduction.

Empirical Productivity [Somewhat Agree]. One example of Apple's creativity was the introduction of the Apple store. Apple is the most successful retailer in history, with an incredible $50,000 in sales per square foot in their best stores (there is no close second) and roughly $13 billion in revenue in ten years. For the Apple stores to succeed, they had to convey the Apple ideal of creative exploration and self-expression. That meant that stores had to look beyond just moving product to changing customers' lives by actively helping them express their creativity. The stores were envisioned as places where consumers could test-drive Apple products and learn the "digital arts" of using those products; where they could join Apple retail employees and other consumers in a real-life, brick-and-mortar, non-virtual community. Steve Jobs saw the stores as places that could best succeed—really, could only succeed—if they strove to inspire greatness in everyone who walked through the door.

According to Collins (J. Collins, The Most Creative Products Ever 1997) if you want to build an enduring great company, don't make the mistake the leaders of Apple Computer made in the late 1980s. After the remarkable success of the Macintosh computer and the departure of Steve Jobs, Apple's leaders spent their time trying to come up with the next insanely notable innovation. Instead, they should have spent their time being social inventors, designing an environment that would be the seedbed for many insanely significant innovations over decades to come. Upon his return to Apple, Steve Jobs changed both himself and ultimately Apple. He focused on what to do when your current product line becomes obsolete, and building a unique culture that could not easily be copied. Ultimately, he experimented with social inventions. Apple was fast becoming part of the next wave of enduring great companies being built not only by technical or product visionaries but by social visionaries—those who see their company and how it operates as their greatest creation and who invent entirely new ways of organizing human effort and creativity.

Level 5 Ambition [Somewhat Agree]. Steve Jobs famously refused to release a new Apple product, or even a product enclosure, until it was as close to perfection as possible. Yet, no one

allowed perfectionism to paralyze Apple's creative processes. Depending on the form it takes, perfectionism is not necessarily a impediment to creativity. A growing body of research in psychology has revealed that there are two forms of perfectionism: healthy and unhealthy. Characteristics of what psychologists see as beneficial perfectionism include striving for excellence and holding others to similar standards, planning, and strong organizational skills. Healthy perfectionism is internally driven in the sense that it is motivated by strong personal values for things like quality and excellence (Steve Jobs). Conversely, unhealthy perfectionism is externally driven. External concerns come up over perceived parental pressures, need for approval, a tendency to ruminate over past performances, or an intense worry about making mistakes (not Steve Jobs). Healthy perfectionists exhibit a deep concern for these outside factors.

Leaders who excel despite an uncertain environment tend to turn first to "empirical evidence, empirical experience, and empirical data rather than immediately seeking what experts or others advise them to do," Collins says. This hands-on approach "often leads 10Xers to highly creative outcomes, since the outcomes are based on empirical validation" (Grams 2011). He points to Apple founder Steve Jobs, who risked much of his company's success on the iPod. "You'd think it was this big creative thing that came out of nowhere," says Collins. "It was not. ... The MP3 was already out in the world, and [Apple employees had] made an iPod for themselves. The company fired what we call 'bullets' in taking small empirical steps to verify the concept, and then they went big with it."

In Research Question 2, we determined that Apple started using the GBC practices based on our financial analysis. Our assessment of Apple's use of the four practices confirmed the proposition.

4. CONCLUSIONS

Overall, we conclude that GBC has sound advice for companies. Given the life cycles of organizations, products and industries there is an ebb and flow that is evident in the financial bottom line. However, in GBC Collins and Hansen attempted to explain what some of these ingredients might be in the form of practices. Our approach to the study replication and extension was rigorous and required extensive subjective analysis. In our selection of Apple and Microsoft, we focused on a single pair in a dynamic industry.

There is a tendency among academicians to dismiss whitepapers, practitioner publications, and web-based articles as not meeting the rigorous standards required for academic journals. Collins' works demonstrate the value of combining financial and practitioner analysis.

In our paper, we applied Collins' and Hansen's techniques to see if the practices they identified apply beyond the dynastic period of identification and to companies who adopt the practices. Does the momentum continue, or as in the case of Apple verses Microsoft, does performance and practice change over time. One final caveat: eleven years is a long time in the technology industry. Collins did examine the companies in his study on a year-by-year basis but summarized/coalesced his findings in a binary fashion. Our practitioner analysis attempted to replicate this process wherein we showed a succession of significant events that tempered our determinations.

Microsoft reduced their use of GBC practices. The decline of Microsoft may be based on moving away from GBC practices. For example, the change in leadership or perhaps the proliferation of products, many of which were cannon balls being shot after the battle was almost over (e.g. the entry of Bing into the search engine wars dominated by Google) cost Microsoft over $2 billion in losses. For Apple, that started using the practices, their performance improved. The adoption of GBC practices for an organization is best depicted by the resurgence of Apple. Isaacson (Isaacson 2011) narrates the ebb and flow of Steve Jobs from his formation of Apple, the release and success of the Macintosh, the deviation from fanatic discipline, the learning at Pixar, and the return and re-vitalization in the four-product business plan.

Apple had changed (Arthur 2012). From just under 10,000 full- and part-time staff in September 1998, it has grown to being 50,000 strong, with around 30,000 in its retail store chain. The core of the company remains small and relatively tight-knit. On August 9, 2011, Apple's market capitalization briefly rose to $341.5 billion, edging it just ahead of Exxon, until that morning the highest-valued company in the world. The company Steve Jobs had co-created assembling computers, the one that Michael Dell had suggested shutting down 14 years earlier

because it had no future, was now worth more than any other. The stock fell back by the end of the day, but it had made its mark; the transformation of Apple from financial basket case to ruler was complete. At the end of the day, it was worth $346.7 billion; Microsoft was worth $214.3 billion (Elmer-Dewitt 2013).

The rivalry with Microsoft still flickers occasionally, but strategically they virtually ignore each other. AppTle has won in music. Its position in phones and tablets has pushed Microsoft to playing catch-up, yet Microsoft can still rely on its sheer heft of 1.5 billion PC installations to ensure a stream of replacements and new sales for Office. Apple's reputation has been transformed from a put-upon, also-ran PC maker to world-spanning design brand.

5. REFERENCES

Achido, B. (2012, June 25). Microsoft's Yammer deal may cost too much, come too late. Retrieved from USA Today: http://www.usatoday.com/tech/news/story/2012-06-25/microsoft-yammer-aquisition/55811172/1

AFP Relax. (2012, May 21). Microsoft launches social network So.cl, claims not to compete with Facebook. Retrieved from Yahoo OMG: http://ph.omg.yahoo.com/news/microsoft-launches-social-network-cl-claims-not-compete-110708921.html

Anonymous. (2010). The New Breed of Leadership. Pharmaceutical Executive.

Apple Inc. (2011, October 18). Apple Reports Fourth Quarter Results. Retrieved from Apple Press Release: http://www.apple.com/pr/library/2011/10/18Apple-Reports-Fourth-Quarter-Results.html

Apple Inc. (2011, July 19). Apple Reports Third Quarter Results. Retrieved from Apple Press release: http://www.apple.com/pr/library/2011/07/19Apple-Reports-Third-Quarter-Results.html

Apple Inc. (2012, January 24). Apple Reports First Quarter Results. Retrieved from Apple Press Release: http://www.apple.com/pr/library/2012/01/24Apple-Reports-First-Quarter-Results.html

Apple Inc. (2012, May 21). Apple Reports Second Quarter Results. Retrieved from Apple Press Release: http://www.apple.com/pr/library/2012/04/24Apple-Reports-Second-Quarter-Results.html

Arthur, C. (2012). Apple vs. Google vs. Microsoft: Battle for digital supremacy. Kogan, Page.

BBC News. (2006, June 15). Bill Gates: A timeline. Retrieved from new.BBC.co.uk: http://news.bbc.co.uk/2/hi/business/5085630.stm

Bloomberg BusinessWeek. (2012, September 4). Bloomberg BusinessWeek Financials. Retrieved from Bloomberg BusinessWeek: http://investing.businessweek.com/research/stocks/financials/financials.asp?ticker=PGR

Clarke, K. (2012, February). The No-Excuses Guide to Greatness. Retrieved from Association Now: http://www.asaecenter.org/Resources/ANowDetail.cfm?ItemNumber=144608

Collins, J. &. (2001). Built to Last: Successful Habits of Visionary Companie. HarperCollins.

Collins, J. &. (2011). Great by Choice: Uncertainty, Chaos, and Luck--Why Some Thrive Despite Them Al. Harper Collins.

Collins, J. (1997, May). The Most Creative Products Ever. Inc.

Collins, J. (2001). Good to Great: Why Some Companies Make the Leap...And Others Don't. Harper Collins.

Collins, J. (2009). How The Mighty Fall: And Why Some Companies Never Give In. HarperCollins.

Collins, J. (2011, September 30). How to manage through chaos. Fortune.

Collins, J. (2011). Good To Great And The Social Sectors: A Monograph to Accompany Good to Great. HarperCollins.

Collins, J. a. (2011). Great by Choice. HarperCollins.

Darrell, R. (2012, May). Microsoft vs. Apple: The History Of Computing [Infographic]. Retrieved 5 9, 2013, from Geek: http://www.bitrebels.com/geek/microsoft-vs-apple-the-history-of-computing-infographic/

Elmer-Dewitt, P. (2013, May 6). Apple Cracks the Fortune 10. Fortune.

Fontana, J. (2007, July 23). Microsoft has Plenty to Change. Network World.

Freid, I. (2009, February 12). Microsoft follows Apple into the Retail Business. Retrieved from CNET CBS Interactive.

Fried, I. (2008, October 27). Microsoft launches Windows Azure. Retrieved from CNET CBS Interactive.

Fubini, D. P. (2006). Mergers: Leadership, Performance and Corporate Health. Insed Business Press.

George, B. (2007). True North: Discover Your Authentic Leadership. John Wiley & Sons.

Gilbert, J. (2012, 10 25). Microsoft vs. Apple In Two Photos: The Battle Over The Many And The Few. Retrieved 05 08, 2013, from Hufington Post: http://www.huffingtonpost.com/2012/10/25/microsoft-vs-apple_n_2017801.html

Google. (2012). Google Finance. Retrieved from Google.com: http://www.google.com/finance?q=NASDAQ%3AMSFT&fstype=ii&ei=zaFQUMmRHISSiALMKA

Grams, C. (2011, October 10). A review of the new Jim Collins book "Great By Choice". Retrieved from Dark Matter Matters: http://darkmattermatters.com/2011/10/26/a-review-of-the-new-jim-collins-book-great-by-choice/

Griffith, T. (2011). The Plugged-In Manager: Get in Tune with Yur People, Technology and Organization to Thrive. John Wiley and Sons.

Heritage, C. (2006). Microsoft Innovation through HR's Partnership. Strategic HR Review, Mar/Apr 24-27.

Iansiti, A. M. (2009). Intellectual Property, Architecture, and the Management of Technological Transitions: Evidenced by Microsoft. Journal of Production and Innovation Management, Vol. 26.

Isaacson, W. (2011). Steve Jobs. Simon and Schuster.

Iskowitz, M. (2011). Harrison & Star. Medical Marketing and Media.

Kate, M. (2005, October 100). Microsoft Opens the Lines of Communication. B to B, pp. 1-42.

Machlis, S. (2011, April 19). Apple vs. Microsoft by the numbers. Computerworld.

Meyrick, M. (2001). In Search of Excellence. British Journal of Administrative Management, 24-25.

Mirchandani, V. (2010). The New Polymath: Profiles in Compound Technology Innovations. John Wiley and Sons.

Mirchandani, V. (2012). The New Technology Elite: How Great Companies Optimize Both Technology Consumption and Production. John Wiley and Sons.

Murray, A. (2011). Turbulent Times, Steady Success. Wall Street Journal, Oct.

Peters, T. J. (1989). In Search of Excellence: Lessons from America's Best-Run Companies. Harper Collins.

Porter, M. (1985). Competitive Advantage: Creating and Sustaining Superior Performance. Simon & Schuster.

Porter, M. (1998). Competitive Strategy: Techniques for Analyzing Industries and Competitors. Simon & Schuster.

Rodgers, W. (2008, August 18). Intel exec: MS wanted to 'extend, embrace and extinguish' competition. ZDNet.

Snyder, B. (2010, April 22). Who's buying Microsoft's outsourcing excuses? Info World.

Solomons, D. (1983). In Search of Excellence: Lessons from America's Best-Run Companies. Journal of Accountancy.

Stengel, J. (2011). Grow: How Ideals Power Growth and Profit at the World's Greatest Companies. Random House.

Tu, J. (2011, October 8). Microsoft named best multinational workplace. Seattle Times.

U.S. Securities Exchange Commission. (2011). Microsoft 10-K Report. Washington, DC: U.S. Government.

van Kotten, M. (2011, August 23). Global Software Top 100 Edition 2011: Highlights. Retrieved from SoftwareTop100.org: http://www.softwaretop100.org/global-software-top-100-edition-2011

Wingfield, N. a. (2011, January 4). Microsoft Alliance With Intel Shows Age. Wall Street Journal.

Wunker, S. (2011). Capturing New Markets: How Smart Companies Create Opportunities Others Don't. McGraw-Hill.

Editor's Note:

This paper was selected for inclusion in the journal as a CONISAR 2013 Distinguished Paper. The acceptance rate is typically 7% for this category of paper based on blind reviews from six or more peers including three or more former best papers authors who did not submit a paper in 2013.

Appendix

	Total Price Return % and Times Better					
	GBC (1991-2001)			Update (2002-2012)		
	Percentage Change Last 11 Years GBC	Times better than S&P 500	Times better than Comparison Company	Percentage Change	Times better than S&P 500	Times better than Comparison Company
S&P 500 Index	**319**			**54**		
Microsoft	5280	12.8	42.7	-6	0.6	
Apple	26	0.3		4510	29.9	48.8

Table 2. Total Price Return Percentage Comparison

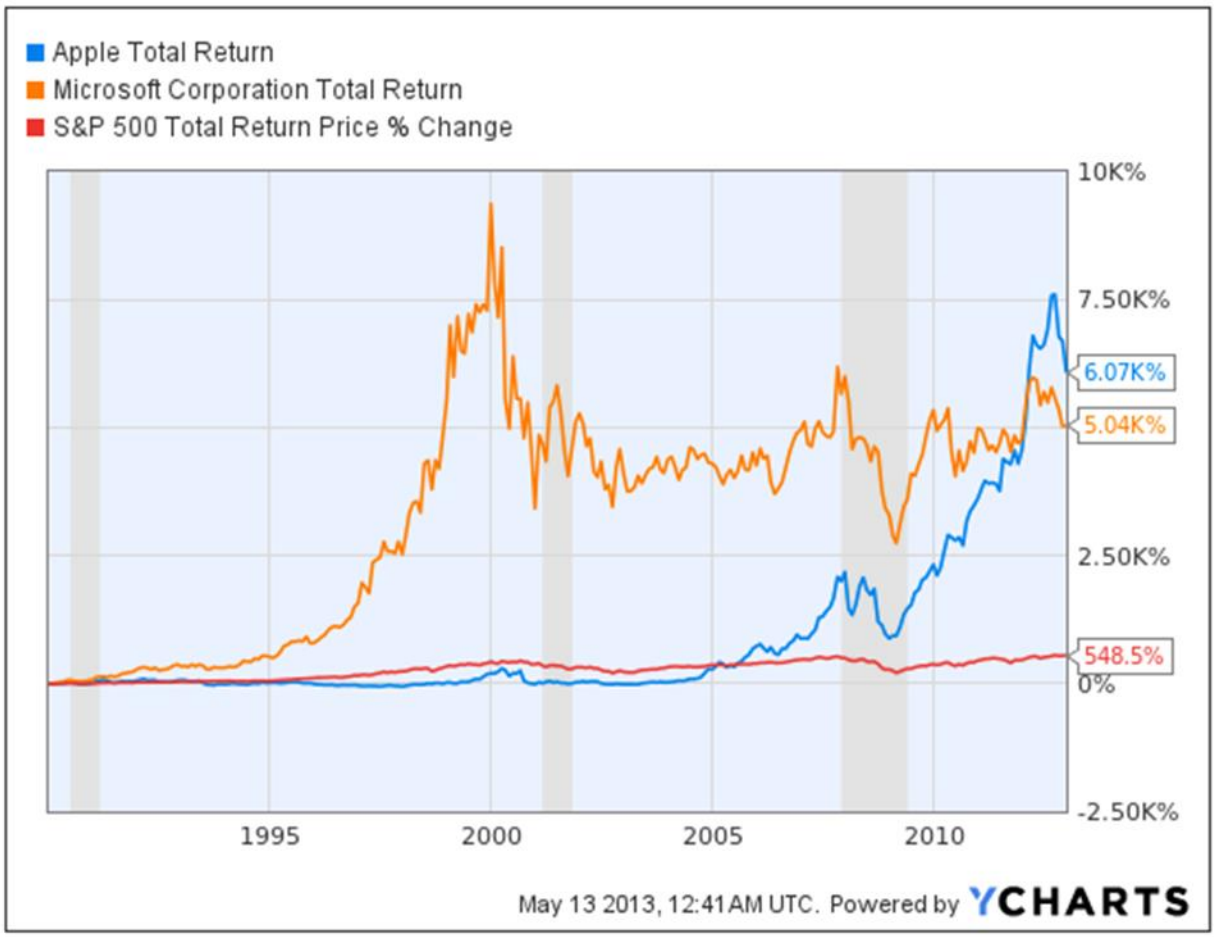

Figure 1. Total Return Microsoft vs Apple

Current Ratio and Debt/Equity Ratio

	GBC (1991-2001)		Update (2002-2012)	
	Avg Current Ratio	Avg Debt/Equity Ratio	Avg Current Ratio	Avg Debt/Equity Ratio
Microsoft	3.57	0.00	2.74	0.06
Apple	2.42	0.30	2.39	0.01

Table 3: Median Current Ratio and Debt-to-Equity Ratio

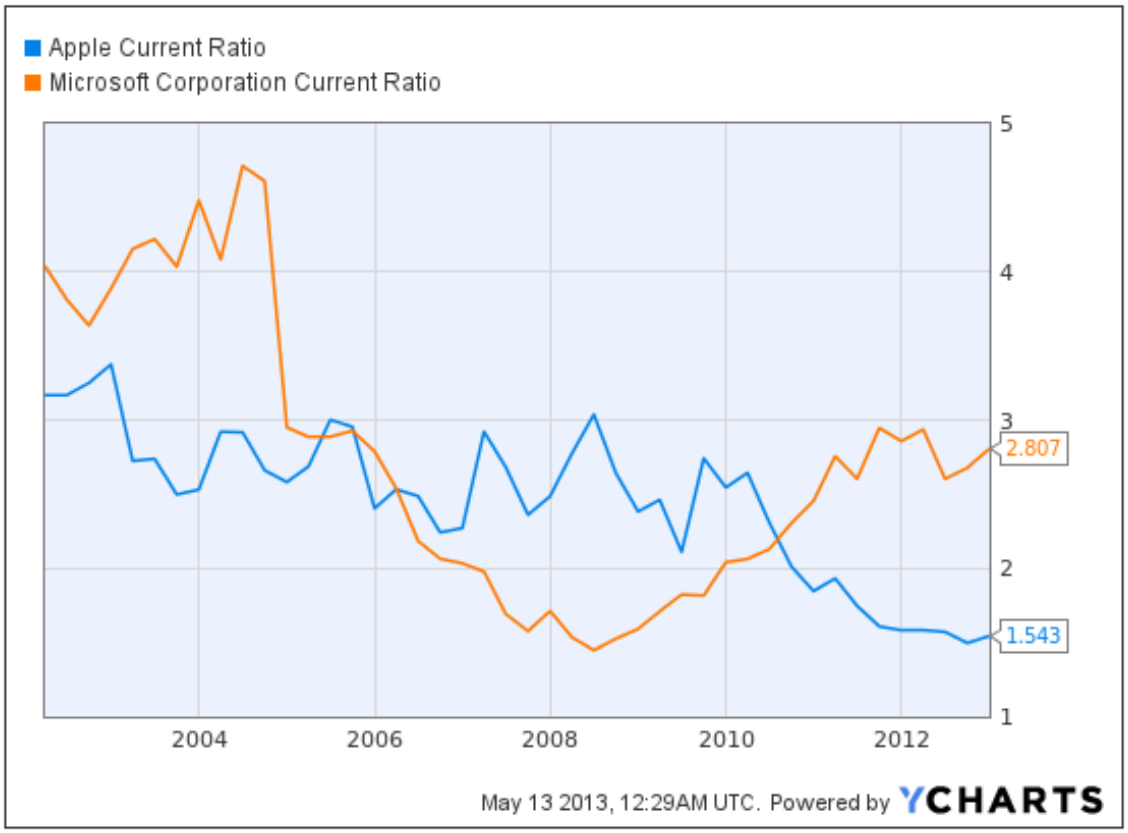

Figure 2. Current Ratio Microsoft vs Apple

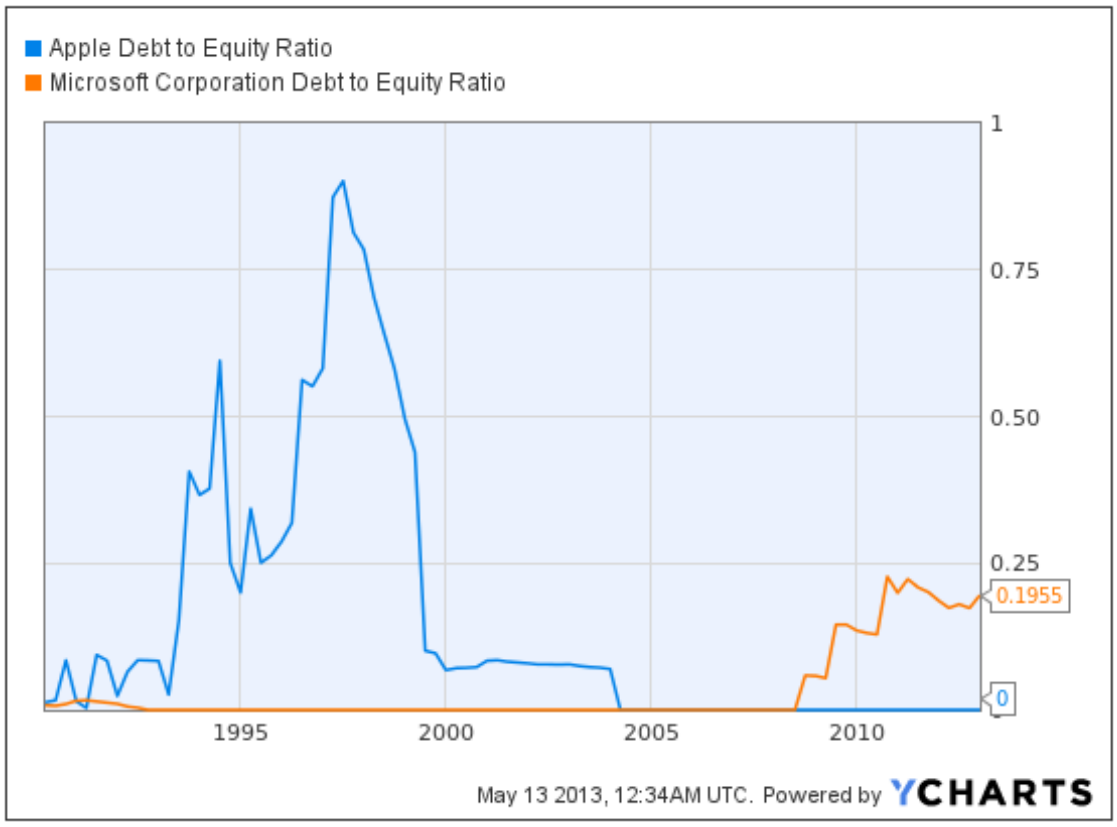

Figure 3. Debt to Equity Ratio Microsoft vs Apple

	Financial Observations		Practices Observations (according to Literature)					
Research Question	**Company**	**GBC Practices (per financial analysis)**	**Fanatic DISCIPLINE**	**Productive PARANOIA**	**Empirical CREATIVITY**	**Level5 AMBITION**	**Summary Practices**	**GBC practices (per Literature)** (Agree/Disagree)
1	**Microsoft**	Stopped using	4.4	3.3	5.3	3.6	4.2	Neutral
2	**Apple**	Started using	5.0	5.4	5.4	4.9	5.2	Somewhat Agree

1	Strongly disagree
2	Disagree
3	Somewhat disagree
4	neutral
5	Somewhat agree
6	agree
7	Strongly agree

Table 4. Proposed GBC practice usage update period 2002-2012

Practice	Analogy	Description
Fanatic Discipline	[The 20 Mile March]	Consistent execution without overreaching in good times or underachieving in bad times. (1) the discomfort of unwavering commitment to high performance in difficult conditions, and (2) the discomfort of holding back in good conditions. GBC leaders and companies demonstrate the discipline to
Productive Paranoia	Leading above the Death Line	Learning how to effectively manage risk so that the risks your organization takes never put it in mortal danger.GBC leaders continuously scan the environment "zoom out" mode and then "zoom in". This puts specific plans and resources in place to cover lower probability eventualities if the effect is potentially devastating
	Return on Luck	"The critical question is not whether you'll have luck, but what you do with the luck that you get.
Empirical Creativity	[Firing Bullets, Then Cannonballs]	Unique ability to collect and analyze their own data. GBC companies are data driven - testing concepts in small ways and then making adjustments rather than placing big, unproven bets. But then placing big bets when you have figured out exactly where to aim.
Level 5 Ambition		Ambition for the success of the organization rather than self -- many of those classified in this group displayed an unusual mix of intense determination and profound humility; often having a long-term, personal sense of investment in the company and its success, cultivated through a career-spanning climb through the company's ranks. Personal ego and individual financial gain are not as important as the long-term benefits to the team and the company

Table 5. *Great by Choice* practices.

Year	Acquisitions	Infrastructure	Personnel	Philanthropy	Litigation	Financial	Recognition/ Presentations
2002	4				4		
2003	2	1	0	0	3	0	0
2004	2	0	0	0	5	0	0
2005	7	1	0	0	3	0	0
2006	11	2	2	0	3	0	0
2007	8	1	3	1	3	3	3
2008	16	1	0	0	0	1	0
2009	6	1	0	0	0	1	0
2010	3	2	1	0	0	0	1
2011	3	0	1	1	0	3	2
2012	2	1	0	1	0	1	0
	64	**10**	**7**	**3**	**21**	**9**	**6**

Year	Fanatic DISCIPLINE	Productive PARANOIA	Empirical CREATIVITY	Level5 AMBITION
2002	10.0	7.3		
2003	7.5		7.0	
2004	7.0	7.0		
2005	9.0	6.0	8.0	8.0
2006		5.3	8.5	10.0
2007		5.0		
2008		5.0	9.0	5.0
2009	7.0	7.0	8.0	5.0
2010	8.0	5.5		
2011	4.7	7.6	10.0	5.0
2012	6.3	6.3		5.0
	74%	**62%**	**84%**	**63%**

Table 6. Microsoft Four Practices and Considerations

Year	Acquisitions	Infrastructure	Personnel	Philanthropy	Litigation	Financial	Recognition/ Presentations
2002	2	4	2	1	1	6	4
2003	0	2	2	2	0	5	3
2004	0	3	2	0	0	4	2
2005	0	2	1	1	0	5	1
2006	0	3	2	0	1	4	1
2007	0	2	2	0	0	4	1
2008	0	5	2	0	0	4	2
2009	0	0	3	0	0	5	1
2010	0	5	2	0	1	4	2
2011	0	0	3	0	1	5	3
2012	0	1	2	0	1	3	2
	2	**27**	**23**	**4**	**5**	**49**	**22**

Year	Fanatic DISCIPLINE	Productive PARANOIA	Empirical CREATIVITY	Level5 AMBITION
2002	6.67	7.68	8.93	
2003	8.5	8.4	8.7	8.0
2004	6.3	7.1	7.6	7.0
2005	8.4	7.3	7.8	6.0
2006	8.8	8.1	8.5	
2007	6.6	7.8	7.8	9.5
2008	8.2	8.8	7.7	
2009		8.8	9.2	
2010	10.0	9.4	9.6	10.0
2011	10.0	9.8	8.8	5.0
2012	10.0	9.8	9.7	
	84%	**84%**	**86%**	**76%**

Table 7. Apple Four Practices and Considerations

Information Security in Nonprofits: A First Glance at the State of Security in Two Illinois Regions

Thomas R. Imboden
timboden@siu.edu
Southern Illinois University
Carbondale, IL 62901

Jeremy N. Phillips
Jphillips2@wcupa.edu
West Chester University
West Chester, PA 19383

J.Drew Seib
jseib@murraystate.edu
Murray State University
Murray, KY 42071

Susan R. Fiorentino
sfiorentin@wcupa.edu
West Chester University
West Chester, PA 19383

Abstract

Information security is a hot button topic across all industries and new reports of security incidents and data breaches is a near daily occurrence. Much is known about recent trends and shortcomings in information security in the public and private sectors, but relatively little research examines the state of information security in nonprofit organizations. The underlying missions of nonprofit organizations, composition of their workforce, and their reliance on grants and donations for revenue generation streams set nonprofits apart from private business. These facts warrant an examination of information security of nonprofit organizations separate from private or commercial groups. This paper examines the state of information security in nonprofit organizations with results obtained by surveying volunteers or employees at nonprofit groups in two areas of Illinois. A qualitative discussion using observations gained from direct analysis of the security status of three organizations as part of student service learning projects is presented as well.

Keywords: Information Security, Nonprofit, Information Technology

1. INTRODUCTION

Today, organizations thrive on information. Often the success of an organization depends upon the quantity and quality of the data collected and their ability to employ the data as a resource. Collecting information comes with a cost, however. As data collection becomes more prevalent so does the need to protect and secure this data. To date, researchers have focused heavily on how for-profit and governmental organizations use and protect information. To a large extent, research on how the nonprofit sector protects information is lacking. This void is unfortunate considering the size of the nonprofit sector, the increasing reliance on the nonprofit sector to deliver services traditionally provided by governments, and the push within the nonprofit sector to strategically gather information to increase organizational capacity. Nonprofits may be required by law to maintain employee or client information containing medical data, or other personally identifiable information such as social security numbers, credit history, and criminal background check information. Failure to maintain the confidentiality of this information can result in legal liability.

This paper proceeds as follows. First, the authors survey the literature on nonprofit organizations and information security. Next, the authors provide an overview of the research methodology of the study, an electronic survey of employees at nonprofits in Illinois and an in person analysis of technical and operational security protections at three organizations. Then, the authors present the results of this mixed methods study. The results illustrate that there are significant areas where information security can be improved in nonprofit organizations. A set of four nontechnical and operational recommendations are presented to assist nonprofits in improving their security posture. Finally, the future goals of the authors' work in the area will be shared.

2. BACKGROUND

The need for nonprofit organizations to pay attention to information security issues is ever growing. According to Kolb and Abdullah (2009), the FBI and the Privacy Rights Clearinghouse report that nonprofit organizations are highly susceptible to identity theft due to their strong web presence and use of electronic information. The rise of technology and use of digital information can be attributed to the push for nonprofit organizations to increase their use of strategic information technology, which includes making more data driven decisions and using technology to maximize growth (Hackler & Saxton, 2007).

Encouraging nonprofit organizations to employ strategic application of information and information technology will require nonprofit organizations to collect more information on constituents and the public (Kolb & Abdullah, 2009). Additionally, employing technology to maximize growth means that nonprofit organizations must use technology for focused marketing and fundraising, such as donations by credit card purchases and via direct bank withdrawals, often over the Internet. All of this information (personal information, medical records, credit information, etc.), as well as other organizational data are typically kept electronically on network servers and processed online and require organizations to take proactive steps to protect the integrity of the data through strong information security polices (Donohue, 2008).

The push for democratic governance heightens the need for nonprofit organizations to employ technology, gather data, and share data. First, the increase in the privatization movement means that nonprofits are increasingly taking on governmental roles (Alessandrini, 2002). Additionally, there is a push for more networked forms of governance, where organizations in a policy domain work together to tackle a particular issue. This means highly sensitive information will need to be transferred between organizations (Kolb & Abdullah, 2009). Finally, nonprofits are also turning to the idea of e-governance and accountability through accessible mediums such as the Internet. Thus, they are relying on technology as a means of communicating with the public, increasing the likelihood of exposure of sensitive data and communications (Smith & Jamieson, 2006). If the sensitive information that nonprofit organizations collect is ever exposed, there may be disastrous effects for the nonprofit organization including financial loss, loss of reputation, damage to employee morale, donor disenchantment and loss, and litigation (Kolb & Abdullah, 2009).

Carey-Smith et al. (2007) find that many organizations do not maintain an atmosphere that is conducive to information security. Many organizations do not promote strong security awareness or monitor behavior that could increase risk. Burns, Davies, and Beynon-Davies

(2006) find that several organizations note a "lack of time and knowledge" as the greatest obstacle to employing sound security policies. They surmise that such barriers may be easily overcome by providing a strong information security policy template that organizations can adopt. Carey-Smith et al. (2007) echo this sentiment, "[w]here resources are scarce, every dollar invested in information security can be perceived as a dollar not spent in direct support of the organizational mission." These findings are also consistent with Imboden et al. (2013) who find that the size of nonprofit's budget is the primary factor predicting whether an organization has an information security policy. This study builds on Imboden et al. (2013 and seeks to better understand to what extent nonprofit organizations employ effective policies and practices to protect their organization's data.

For many organizations, the creation of an information security policy is a challenge due to management's lack of understanding of security concerns and issues. Often a policy is seen as unnecessary as minimal technical safeguards such as antivirus software and firewalls are erroneously viewed as protecting an organization. One method for approaching security and creating an improved security posture for an organization is to begin with the creation and adoption of a formal information security policy (SANS). The information security policy provides the organization with a set of expectations to be met regarding information security as well as outlining consequences for not meeting these expectations (SANS). The policy requires compliance and functions as an internal "law" for the organization. The System Administration, Networking and Security Institute (SANS), a leader in information security education and research, publishes a guide and many examples of security policy documents that organizations can freely use to create their own information security policy documents. This resource may be useful in guiding an organization through the first and arguably most cost effective step towards improving the security for many organizations.

3. RESEARCH METHODOLOGY

This study uses a mixed methods approach to identify attitudes and practices related to information security and policies for nonprofit organizations in two regions of Illinois. The first part of this study utilizes a survey instrument administered to nonprofit organizations in the two regions. The survey provides an overview of how nonprofits use and handle sensitive information, as well as a general understanding of the steps that nonprofit organizations take to adopt formal polices to deal with sensitive information. The second part of the study conducts an in-depth security analysis of three nonprofit organizations identified from the original survey. The purpose of the security analysis is two-fold. First, the in-depth analysis provides support for the results obtained from the survey. Second, and more importantly, the security analysis provides detailed information regarding the security practices of nonprofit organizations that cannot be obtained through a survey. Additionally, this qualitative approach provides the participant group with tangible and actionable recommendations to improve information security.

For initial data collection, the authors developed a survey consisting of 39 open and closed ended questions hosted on a web site for participants to complete electronically. Prospective respondents were identified from publicly accessible databases of nonprofit organizations; however, their participation was anonymous. Participants for this study were solicited via email. Two specific areas were targeted: the Chicago metropolitan region and southern Illinois. While the Chicago region consisted of a primarily urban and suburban population, the southern Illinois region encompassed rural areas in addition to the predominantly suburban Illinois area of metropolitan St. Louis, Missouri. During the approximately one month survey response period, 154 surveys were started by prospective participants, of which 78 were completed.

The survey instrument was designed to gather data on the composition of information technology and security hardware and software, resources available to the nonprofit, general group demographic and employee makeup of the organization, employee attitude and experience regarding information security, and the types of potentially sensitive or personally identifiable data their organization stores or processes on their information systems.

A small group of nonprofits located within the local area of one researcher were identified and solicited for participation in the analysis of technical and operational information security policies and protections. Participants were asked to complete the existing information security survey (but not included in the results of the previous portion), provide the researchers copies

of any organizational policies or similar documents that referenced information security or related topics, and allow the researchers to access the organization's technology assets to perform a basic security evaluation of the hardware, software, and operational activities of the organization. Students from a volunteerism-focused, student organization from one author's school with an interest or work experience in information security were identified as research assistants and assisted in the organizational analysis. As motivation for the nonprofits' participation, the student volunteers and the authors agreed to document any security concerns or inadequacies discovered at the nonprofits and, if desired, assist with remediation of potential problems.

In addition to the completion of the original survey by administrators at the local nonprofits, a second list of technical and operational security questions were developed from industry and governmental best practice documents. These questions aimed to determine whether common security best practices were followed at the organizations. As an example, the questions were designed to elicit data regarding, but not limited to, the following:

- Does the organization have a formal information security policy and are members aware of its existence?
- Are common information security protections such as antivirus, firewalls, and operating system and third party software updates implemented and kept current?
- Has the organization experienced incidents that presented potential risks to information security?
- What does the nonprofit view as potential risks from poor information security?

Finally, a follow up survey was sent to the organizations that provided documents that governed organizational procedures or activities related to information security. The survey was designed to discover employee knowledge of and adherence to the provisions of the adopted policy. These surveys were administered to staff and volunteers of the respective organization.

4. RESULTS

When examining the data as a whole, we see the organizations in the sample are very diverse, ranging from operations comprised of no full time employees and no formal information security budget to organizations that devoted a substantial amount of formal resources to information security. Table 1 provides average demographic data on organizations that took part in the electronic survey. As noted in the table, on average, organizations dedicated more than $23,000 dollars to information technology and security and nearly half of the organizations stated they had an employee with formal responsibilities devoted to overseeing information security in the organization.

Characteristic	Mean
Budget	$1,331,352
IT budget	$23,408
Number of employees	19.5
Employees dedicated to IT	46.80%

Table 1 - Size of Nonprofits

Table 2 illustrates the types of personally identifiable information that nonprofit organizations collect. Nearly all organizations collect some type of personal information, with 20-30% of organizations collecting what can be considered sensitive information that could be costly for both the organization and constituents if the information were compromised.

Type of Data	
Names	97.80%
Addresses	94.70%
Phone Numbers	89.50%
Birth Dates	53.70%
Social Security Numbers	31.60%
Health Records	20.80%
Criminal Records	11.50%
Income	27.40%

Table 2 - Types of Data Handled

Given that nonprofit organizations are collecting sensitive information, do they take appropriate steps to protect the information? The authors define "appropriate steps to avoid loss of sensitive information" to mean organizations adopting a formal information security policy that meets the security needs of the organization as well as utilizing programs and procedures, such as antivirus programs and ensuring that such programs are up-to-date, to mitigate information loss. While these are certainly not the only steps required to protect sensitive data and information

systems, the authors believe it a foundation for security to be built upon.

Table 3 details the percentages of organizations in the sample that have a formal policy that governs information security. Additionally, this table provides information on the origin of such policies.

Have formal security policy	**56%**
Developed by employees	39%
Developed by board of directors	33%
Template found online	30%
Created by legal counsel	27%
Provided by parent organization	13%
Provided by another organization	12%
Provided by insurance company	6%
Combination of the above sources	44%

Table 3 - Nonprofit Adoption and Development of Information Security Policies

As noted in Table 3, 56% of organizations in the sample had a formal policy governing the use of information technology and security. Of the organizations identified as having a formal information security policy, the origins of such polices are derived from a variety of places. For example, 30% of organizations with information security polices constructed it from a template found online. Very encouraging is that 44% of organizations with information security policies used two or more sources to develop their information security policy. This suggests that nearly half of nonprofit organizations are thinking broadly when developing their policies. For example, an organization may initially acquire an information security policy from a template, but then consult employees, legal counsel, and/or their board of directors to tailor the policy to fit the needs of the organization.

Also promising is that nonprofit organizations communicate their information security polices to employees and require employees to acknowledge the content of such policies. As detailed in Table 4, 84% of nonprofit organizations with policies formally require their employees to acknowledge policies that govern technology use. What is more, Table 5 illustrates that nonprofit organizations are institutionalizing their technology polices through employee training and inclusion in the organization's employee handbook. A combined 65% of nonprofit organizations hold group or individual trainings, 58% distribute the policy to their employees, and 69% include the policy in their employee handbook.

Required to acknowledged policy	84%
Not required to acknowledge policy	16%

Table 4 - Formal Employee Acknowledgement of Security Policy

Group training sessions	33%
Individual training sessions	32%
Distributed by paper	29%
Distributed electronically	29%
In the employee handbook	69%

Table 5 - How Nonprofits Communicate the Security Policy

In addition to adopting polices to help mitigate threats to security, some nonprofit organizations are also employing appropriate security technologies to help reduce risk. Table 6 provides information on the types of technologies used by nonprofit organizations including antivirus programs, firewalls, and blocking of unauthorized websites and downloads. A large portion of organizations protect all computers in the organization. The data reveal that 80% of organizations have antivirus programs installed on all computers owned by the organization. Additionally, 61% of organizations stated they have firewall programs. There are still a large percent of organizations that are not universally protecting their infrastructure. Less used are web blocking programs that restrict employees from visiting potentially dangerous or prohibited websites.

While nonprofit organizations are using appropriate technologies, our data shows that these organizations are ignoring another risk by not automatically updating software. Recently, malicious attacks have targeted out-of-date versions of operating systems as well as third party applications such as Java, Adobe Reader, and Adobe Flash (Kaspersky Lab, 2012). Table 7 shows that less than half the organizations in the sample use automatic settings to update operating systems and programs.

	Antivirus	Firewall	Web Block
All computers	80%	61%	23%
Some computers	11%	17%	34%
No computers	4%	10%	30%
Unsure	5%	12%	13%

Table 6 - The Use of Antivirus, Firewall, and Web Blocking Programs

Automatic checks	48%
Manual checks	24%
Systems are not checked	17%
Unsure	11%

Table 7 - Maintenance of Operating Systems and Software

Employing information security polices and technologies to reduce organizational risk appear to be born out of real and perceived risk. Table 8 highlights the percentage of organizations in the sample that have experienced specific threats to information security. 43% of the sample notes that they have experienced issues with a virus, spyware, or malware. Roughly a quarter of the sample reports hardware or software malfunctions. And 14% of the sample notes human error leading to an issue with security.

Virus, spyware, and/or malware	43%
Data theft	3%
Hardware theft	10%
Hardware failure	29%
Software failure	24%
Website defacement	3%
Employee error	14%
Employee misuse/vandalism	3%

Table 8: Types of Incidents That Have Occurred

Table 9 suggests that nonprofit organizations are aware of the potential risks of an information breech. In addition to concerns affecting organizational efficiency and effectiveness such as data loss or productivity, organizations are also aware of threats to the organization's reputation and potential legal action that may come for an information breach.

Data loss	80%
Loss of productivity	60%
Hardware damage	32%
Identity theft	33%
General decrease in company security level	31%
Loss of reputation	48%
Legal action	30%

Table 9 - Perceived Consequences of an Information Breach

Security Analysis of Selected Groups

Of the groups solicited for a more in-depth look at their information security policy, employee attitude towards security, and security status, three within one author's locality volunteered for additional focus and participation. Organization 1 (ORG1) is focused on victim advocacy and recovery. Organization 2 (ORG2) serves children in an educational capacity. Finally, Organization 3 (ORG3) serves the community with arts programming. One author has worked with each organization directly and with the support of student volunteers during the course of this project. For each of the three organizations, the administrators responsible for decisions regarding technology or information security were asked to complete the original electronic survey in paper format.

Analysis of Organization 1

The first nonprofit organization studied was found to have an information security posture that given the size, mission, and resources dedicated to information technology, impressed the authors. ORG1's information security practices were deemed strongest of the three nonprofits analyzed. ORG1 employed nearly seventy staff and volunteers, had a budget of over $1.25 million, and served over one thousand clients during the past year. They reported a dedicated information technology budget of $8,700 and owned approximately thirty desktop and three laptop computers.

A formal interview with ORG1 administrative respondents illustrated a wealth of useful data regarding the state of information security at their nonprofit. An in-person observation and evaluation of their procedures and information systems proved to be even more illustrative of the link between policy, accountability, and the security posture of the organization.

While ORG1 did not employ any staff with information technology or security background or training, the authors believe that the assignment of technical and security responsibilities to one of the administrative staff served to directly influence the security posture of the organization's information systems and assets. This nonprofit had the highest number of technology assets, staff and volunteers, and annual operating budget. As illustrated below, the authors believe this employee's implementation of several non-technical and basic security protections was the key factor in increasing the security status of the nonprofit. As an example, a "cheat sheet" on safe computing practices is found next to each computer and serves as a reminder to be cautious and vigilant when using the PCs. While room for improvement exists, the organization was found to be performing more of the most common security tasks and best practices, despite the relative size and number of assets, than the other two organizations. More on the steps taken by this employee will be discussed at the end of this section.

ORG1's policy regarding the acceptable use of computing resources was approved six months prior to the authors' examination of the document. As an example, it referenced employee password standards, prohibited the use of personal email for official business, and outlined enforcement and consequences of breaking the policy. Employees were surveyed regarding the policy and its integration into the organization and its culture. These questions sought to determine the following:

1. Are employees aware of the existence of the information security policy?
2. How is the information security policy communicated to employees?
3. Are employees asked to acknowledge their receipt and adherence to the organization's security policy?
4. Have employees received information security training at their current or previous employers?

The results of the employee survey of the above questions are shown in Table 10. Eighteen employees that routinely used computers and technology were solicited for participated in this survey. Nearly 90% of those surveyed were aware of the existence of an information security policy, while only 16% reported being asked to acknowledge the policy either written or verbally.

Have Policy	Yes	No	Unsure
	16	1	1
Communicated	Email	Meeting	Paper Copy
	1	5	12
Acknowledged	Yes	No	Unsure
	3	12	3
Security Training	Yes	No	
	3	15	

Table 10 - ORG1 Employee Security Policy Survey

The nonprofit serves victims of crime, and is mandated by state law to protect the privacy of their clients. As is likely the case with administrators in many nonprofits, one individual "wore many hats", and supporting and administering technology and security was one secondary duty assigned to them. In certain circumstances, inappropriate or unauthorized disclosure could lead to misdemeanor criminal charges. While the administrator possesses no formal background in security or information technology, they took it upon themselves to learn about and take steps to improve the security at the organization by ensuring employees were aware of a few basic activities to protect their computer use and actions.

Student volunteers were also given permission to examine the desktop and laptop computers at ORG1 in order to assess the status of several common applications and operating system settings that affect the system's security and, in-turn, organization security. Specifically, students observed and assessed the following:

- Operating system version
- Status of operating system updates and patches
- Status of antivirus application and associated definitions
- Status and version of Java
- Status and version of Adobe Reader
- Status and version of Adobe Flash
- Screensaver lock and idle delay
- Status of operating system firewall
- Account permissions given to users

The complete results of this analysis will be presented in future work, but an overview found a few common themes.

- Older systems that were performing slowly were more likely to be missing

operating system updates and running out of date third party applications.
- While the security policy required use of time delayed screensaver locks, a majority of the systems did not implement them.
- Overall, systems were running recent versions of third party applications with few exceptions.
- Surge protectors were supplied and used for most workstations.
- Antivirus software was running, updated, and virus scans ran regularly.
- Most computers contained files in their My Documents folder that their users were responsible for backing up. The type or importance of these files was not examined.
- A majority of the user accounts logged in when students performed their security analysis were operating with full administrative privilege.

Analysis of Organization 2

The second organization (ORG2) was substantially smaller than ORG1 in terms of the number of employees, budget, and clients served. The annual budget was reported at $650,000, of which none was allocated for information technology and security. Approximately twenty-five employees and volunteers worked with the nonprofit over the last year. Of these, three are considered managers with the power to make decisions regarding information technology; however, technology purchases must be approved by board members.

ORG2 reported that an information security policy did not exist. They reported a lack of expertise as well as a lack of an industry or legal requirement as factors contributing to lack of a policy. The managers acknowledged storing or processing potentially personally identifiable information on their systems.

ORG2 owns two desktop computers, which are primarily used by the management staff to keep track of financial information, communicate with clients, and to create operational paperwork. It was originally observed that of the two computer systems, one was completely nonfunctional and had been for months, creating a burden on the organization. During the course of discussion with this group, the second PC suffered a hardware malfunction, rendering the organization unable to perform several regularly required operational duties via their standard procedures. It was found that data, including some which was critical to the groups operation, had not been recently backed up on either of the two failing computers. A volunteer was solicited by the organization to assist and two replacement PCs were purchased, configured, and installed. A data recovery firm was contracted to restore the data lost during the system hardware failures. It was also noted that other instances of virus infection, hardware failure, and software or data corruption had previously affected the nonprofit. No employee was responsible for information technology and security at ORG2. Antivirus software and firewalls were running on the computers, but operating system and third party applications were out of date and not routinely updated. The organization was also unaware that their Internet router created an unneeded and unused wireless network access point.

Analysis of Organization 3

The smallest organization in terms of budget was ORG3. They reported an annual budget of $25,000, of which none was allocated for information technology and security. ORG3 is unique in that while only employing one paid staff member, approximately 120 volunteers supported the organization and made use of the four desktop computers used by ORG3 to help serve the community and fulfill the group's community arts mission. Like ORG2, it was reported that a security policy did not exist and that a lack of perceived need and lack of expertise required to create one was behind this fact. Again, like ORG2, it was reported that a recent incident caused by employee misuse resulted in the loss of mission critical donor related files from a storage device. Recreating the files took over forty hours of volunteer time. Unlike ORG2, it was reported that antivirus software was not used but common third party applications and operating system updates were regularly checked and maintained. Personally identifiable information for volunteers and donors is stored or processed on ORG3's computers.

Common Themes from Direct Organization Observations

There were several common characteristics or shared themes found across the nonprofits. All three organizations reported loss of data due to hardware or software failure, employee misuses or error, or similar circumstances. In two cases, it was reported that the missing data had been backed up at one time, but when attempting to recover the data from backup copies, they were found to be too old to be useful or corrupt. In one

circumstance one group paid a specialized data recovery firm $500 to recover data critical to the organization. In a second case, a volunteer had to recreate customized files crucial to donor and underwriting activities taking over forty hours to do so.

A second common theme was the lack of a dedicated information technology support staff member or even consultant who regularly provided guidance and assisted with maintenance of information systems. All the organizations reported having at times paid for help from local technology businesses as needed, often only when an emergency need arose. Contrasting this with the need to regularly perform software updates and other types of routine maintenance to improve security, it was expected that these tasks were neglected, putting individual and organization wide systems at higher risk. As ORG2 and ORG3 reported no budget funds allocated for information technology, it would stand to reason that paying outside help to fix technology issues would be a last resort. Secondly, given the need for nonprofits to rely on volunteers, it was found that each group relied on the information technology help and skills of volunteers trained in or working in IT positions.

Another common theme that is evident, given the examples of data loss and hardware failure, is the lack of redundancy in business critical hardware and applications, and the absence of regular and reliable backup technologies and processes.

Lessons Learned

Several key actions or themes that were believed to contribute significantly to the positive security stance of an organization were identified.

1. **Have an Information Security Champion –** Identify a single employee who can be charged with leading the effort for improved security. Understanding and implementing even the most basic security practices such as maintaining operating system and third party application updates will help decrease incidents.
2. **Create a Policy -** A basic policy addressing information security will help employees understand that information security is important to the organization and will provide a level of expectation regarding their use of technology.
3. **Train and Talk –** While it is unreasonable to expect volunteers and employees to become security experts, several basic tasks and activities can contribute to improving security. A regular discussion, whether in the form of formal meetings or as an informal email reminder of security tips, serves to open dialogue on the subject and keep it fresh in their minds.
4. **Develop Organization Specific Materials –** Create posters reminding users to think before they click and provide security checklists such as a "Do's and Don'ts" for safe computing to keep next to computers. This can serve as yet another illustration that the organization is concerned with security.

5. FUTURE WORK

The information presented in this paper is simply a first glance at the state of information security in nonprofit organizations. The authors intend to increase data collection efforts to expand to diverse regions across the United States. Results from a larger population will help to determine even further where deficiencies in information security practices and policies exist and provide researchers with a foundation for the development of resources that may help nonprofits. Those with minimal resources and expertise in information technology and security certainly could use help to improve their security posture and use their technology safely and efficiently.

6. REFERENCES

Alessandrini, M. (2002, October) A fourth sector: The impact of neoliberalism on non-profit organizations. Paper presented to Australasian Political Science Association Jubilee Conference, Canberra, Australia.

Burns, A., Davies, A., & Beynon-Davies, P. (2006, November) A study of the uptake of information security policies in small and medium sized businesses in Wales. Paper presented at Global Conference on Emergent Business Phenomena in the Digital Economy, Tampere, Finland.

Carey-Smith, M., Nelson, K., & May, L. (2007). Improving information security management in nonprofit organizations with action. Proceedings of 5th Australian Information

Security Management Conference (pp. 38-46), Perth, Australia: School of Computer and Information Science Edith Cowan University

Denhardt, J. V., & Denhardt, R. B. (2011). The new public service: Serving, not steering. New York: ME Sharpe.

Donohue, M. (2008) States push to encrypt personal data. The Nonprofit Times. Retrieved from http://www.thenonprofittimes.com/news-articles/states-push-to-encrypt-personal-data/.

Hackler, D., & Saxton, G. D. (2007). The strategic use of information technology by nonprofit organizations: Increasing capacity and untapped potential. Public Administration Review, 67(3):474–487.

Hrywna, M. (2007). Nonprofits and data breaches. The Nonprofit Times. Retrieved from http://www.thenonprofittimes.com/news-articles/nonprofits-and-data-breaches/.

Imboden, T. R., Phillips, J. N., Seib, J. D., & Fiorentino, S. R. (2013). How are nonprofit organizations influences to create and adopt information security policies? Issues in Information Systems, 14(2): 166-173.

Kaspersky Lab. (2012). Oracle Java surpasses Adobe Reader as the most frequently exploited software. Kaspersky Lab Corporate News. Retrieved from http://www.kaspersky.com/about/news/virus/2012/Oracle_Java_surpasses_Adobe_Reader_as_the_most_frequently_exploited_software.

Kolb, N., & Abdullah, F. (2009). Developing an information security awareness program for a non-profit organization. International Management Review, 5(2):103–108.

SANS. SANS Security policy project. Retrieved from http://www.sans.org/security-resources/policies/.

Smith, S. and Jamieson, R. (2006). Determining key factors in e-government information system security. Information Systems Management, 23(2):23–32.

A Comparison of Software Testing Using the Object-Oriented Paradigm and Traditional Testing

Jamie S. Gordon
jamie.s.gordon@unf.edu

Robert F. Roggio
broggio@unf.edu

School of Computing, University of North Florida
Jacksonville, FL 32224 United States

Abstract

Software testing is an important part of any software development. With the emphasis on developing systems using modern object oriented technologies, a critically-sensitive issue arises in the area of testing. While traditional testing is reasonably well understood, object oriented testing presents a host of new challenges. This paper focuses on what differentiates the two in test cases, testing levels, and OO features affecting testing.

Keywords: Object-oriented testing, traditional testing, testing levels

1. INTRODUCTION

Object-oriented testing is based not only on both the input and output of an object's methods, but also how that input and output may influence the object's state. Many of the positive features touted by object-oriented languages can map directly into increases in testing complexity. While the many beneficial features of the object-oriented paradigm are important, the increases in program complexity (sometimes in unintended and unseen ways) often negatively impacts testing in terms of effort and time.

Traditional testing involves the viewing of input and output of a program in a procedural manner. Both types of testing still involve tried and true testing types. In fact, many of the differences show up in white-box testing because the two types of programming can often solve the same problems using the same input and output.

This paper seeks to determine how testing is different in an object-oriented paradigm versus that of a traditional (procedural) program.

2. LITERATURE REVIEW

Research done on object oriented testing has changed over the years. Many early papers written on the subject lamented the inability of researchers to address the differences between object-oriented programs and procedural testing. As Turner and Robson pointed out, "the vast majority of research conducted into the testing of object-oriented programs fails to address the difference between the object-oriented and more traditional programming techniques," (Turner & Robson, 1993).

At around that same time, Hayes wrote a paper identifying some aspects of object-oriented programs and how that may affect systems (Hayes, 1994). The paper also described a testing methodology that the author believed

should be recommended for OO-programs. Hayes was successful at identifying the problems involved with inheritance, but did not discuss many other issues in much depth, such as polymorphism and dynamic binding. This was a problem with many early papers on the subject, which primarily focused on objects' states and inheritance. It became increasingly evident that many more features of object-oriented programming need to be considered for testing by looking at more recent papers, such as that of Jain, "Testing Polymorphism in Object-Oriented Programming," which described how advanced polymorphism makes it difficult to understand all the possible interactions among classes (Jain, 2008).

Gu, et al., wrote a paper detailing three processes to select test data and for evaluating the coverage of those tests. They were the flow-graph-based approach, graph-based class testing, and the ASTOOT approach, which used algebraic specification to determine test cases. These methods derive test data, for example the flow-graph-based process uses the flow of control from method to method to model test data while the graph-based process models transitions between different states of an object. These focused on program flow and state changes and ignored other features of OO as well. The authors also discussed how the testing of object-oriented programs must differ from traditional testing methods (Gu, et al., 1994).

By 1996, there was already enough literature for Johnson to report on the different testing levels and techniques proposed by researchers (Johnson, 1996). However, there was not much of a consensus at that point for a standardized system of testing. For example, there was a disagreement on unit-testing, in that some authors disagree that it should be involved in object-oriented testing at all.

As time went on, the differences between object-oriented and traditional (procedural) testing became more evident. For example Khatri, et al., described many features – encapsulation, inheritance, polymorphism, etc. – of object-oriented programming and how they made it more difficult to decide how test should be done (see *Object-Oriented Features that Affect Testing* below) (Khatri, et al., 2011). However, that paper did not describe in detail how testing should be done. Bhadauria described the same features, but also gave a sequence of testing levels and what kinds of tests should be run in each (Bhadauria, 2011). Some authors described design metrics that may help programmers determine beforehand how difficult to test their design may be (Badri, 2012). This sort of empirical view of object-oriented testing is another useful area of study. Authors have discussed both new and older metrics for measuring testability, and which are the most valuable to object-oriented programming (Yeresime, et al., 2012).

3. BACKGROUND: OBJECTIVES OF TRADITIONAL AND OBJECT-ORIENTED TESTING

There are many people with vested interests in the testing process, including programmers, testers, program managers, and end-users. These people are some of the stakeholders in the system, those that are impacted by the system or influenced by its behavior. Individuals or groups of individuals acting in these roles are those who depend on testing to show the systems performs as intended. The major objective in testing is to discover as many faults, errors, and defects as possible with minimum effort and cost (Khatri, Chillar, and Sangwan, 2012). According to Johnson (Johnson, 1996), "Testing is the process of executing a program with the intent to yield measurable errors." Testing is not about showing that there are no errors – effective testing comes from creating effective test cases that can coerce out errors and failures (Naik & Tripathy, 2013). This can help designers find 'defects' (term attributed to design) and programmers find 'faults' (term normally attributed to programming). Given this backdrop, however, what constitutes an effective test is quite different when contrasting traditional (procedural) testing and object-oriented testing (Dechang, Zhong, & Ali, 1994)

4. TEST ADEQUACY AXIOMS

Elaine Weyuker defined eleven axioms to determine the adequacy of a test set (Hayes, 1994). Some are less interesting as they apply equally to both testing paradigms, such as the applicability axiom (Every program has an adequate test set), the monotonicity axiom (It is possible to create a set of test cases that is larger than is necessary), the renaming axiom (If P is simply a renaming of Q, and T is adequate for Q, then T is adequate for P), or the non-exhaustive applicability (Program P is adequately tested by T, where T is a non-exhaustive test set). Below

is a list of the axioms to consider when discussing the difference between traditional and object-oriented testing. (Table 1)

Axiom	Description	Traditional	Object-Oriented
Complexity	For all n, there is a program that is adequately tested by a test set of size n, but not by a test set of size n-1	There is a minimum set of inputs that must be tested	There is a minimum set of inputs and object states that must be tested
Anti-extensionality	There are programs P and Q that compute the same functions (semantically similar), where T is adequate for P but not for Q	It cannot be assumed that the same test cases can be used for different programs that accomplish the same things	It cannot be assumed that the same test cases can be used for functionally similar programs, this can be extended to mean that just because one state is correct for one program, that does not mean that is correct for a similar program
General Multiple Change	There are programs P and Q that are syntactically similar, where T is adequate for P but not for Q	Syntax does not tell you what needs to be tested	The syntax of two programs does not determine the test sets, this also means that if two programs use the same classes, the test cases should be different because the messages sent between them may be different
Anti-decomposition	There is a program P and component C where T is adequate for P and T' is the subset of T that can be used for Q, but T' is not adequate for Q.	A component of a program (say a method) can be adequately tested for use within one program, but not necessarily on its own.	"When a new subclass is added (or an existing subclass is modified) all the methods inherited from each of its ancestor super classes must be retested."
Anti-composition	There exist programs P and Q and a test set T where T is adequate for P and the subset of T that can be used for Q is adequate for Q, but T is not valid for P;Q (the composition of P and Q).	Two programs (or methods) can be adequately tested on their own, but once combined or used in another class; they may no longer be adequately tested.	"If only one module of a program is changed, it seems intuitive that testing should be able to be limited to just the modified unit. However, [this] states that every dependent unit must be retested as well."

Table 1 Test Adequacy Axioms (Hayes, 1995)

5. TEST CASES

Traditional Test Cases

Test cases are often based on the traditional model of processing. The traditional Von-Neumann model of processing is in Figure 1.

Figure 1 Von-Neumann Model of Processing (Labiche, Tosse, Waeselynck, & Durand, 2000)

This model works well for the procedural paradigm where the input dictates what the

output of a program is. In accordance with this model, a test case ignores the processing aspect and focuses on input and output. One may thus view a test case as an ordered pair: <input, expected output> (Naik & Tripathy, 2008). This can be done because the processing is only dependent on the input into the application, as there is no program 'state' to consider, necessarily.

The expected output of a system would normally be described as either values produced by the program or messages to the user based on the input (Naik & Tripathy, 2008). The rationale behind this assertion is justifiable, as the output is program-generated and defined structurally rather than behaviorally (Johnson, 1996). This also implies test cases may be derived from static analysis for dynamic testing (discussed ahead).

Object-Oriented Test Cases

Test cases in the object-oriented paradigm are more complex. The traditional testing model is insufficient, because objects in a program have their own states which may well be impacted by the processing of input parameters (Turner & Robson, 1993). In addition, these state changes may not at all be evident from the output of a program. For example, consider a program that has objects of this class:

Student
-name: String -grades: int[]…
+addGrade(int grade): void +sortedGrades(): int[]…

Figure 2 Student Class Examples

In this example, only the important methods are listed. Suppose a test case is developed for sortedGrades(), where sortedGrades() is supposed to sort the grades array and then return the sorted values. Consider the following test case:

<add grades: (100, 50, 75),
expected output of sortedGrades(): 50, 75, 100>

Figure 3 Test Case with Output Results

These tests might pass with the traditional test model. However, without examining the state of the Student object, it is unknown whether the grades array has actually been altered, or if the sortedGrades() method simply returns a sorted array of integers without actually altering the grades array. The method sortedGrades() is designed to return the grades[ ] array as a sorted list without affecting the grades[ ] array itself. The reason for this is so that the user may specify in the interface that they want grades in ascending order by percentage. However, they may also want the grades in the order that they were entered, so it is important to preserve the original state. This means that not only should the expected output of the method be tested, but the expected state of the class should also be included in the test case. (Figure 4)

<add grades: (100, 50,75),
expected output of sortedGrades(): 50, 75, 100},
expected state of grades[]: {50, 75, 100}>

Figure 4 Test Case with Output and State

Due to this trait of the object-oriented model, the Von-Neumann model needs to be changed to accommodate the state of the objects involved in processing. Robson and Turner suggest the following adaption (Figure 5):

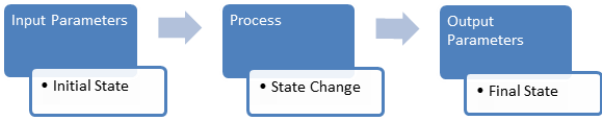

Figure 5 Von Neumann Model with Added State Changes (Turner and Robson, 1993)

Indeed, Dechang, et al., 1994 agree that an effective test case involves both the changing class state and the sequence of operations. The object-oriented paradigm is based on objects as instances of classes; therefore programming is inherently state-based. Not only that, but an object's internal values are not the only thing to consider when developing test cases. The associations between objects through method calls, inheritance, polymorphism, etc. make object-oriented test case generation much more complex (Johnson, 1996) (discussed ahead). For now, it is important to note that there is no strict input-process-output correspondence in object-oriented programming. For more advanced testing, where a method chain is involved for example, it is recommended that a few more items are inserted into the test case: (1) a list of messages and operations that either will or may be executed by the test, (2) any exceptions that may or are expected to occur, and (3) any

environmental setup external to the program (Bhaudaria, et al., 2012). This is in addition to any supplementary information deemed necessary.

6. TESTING LEVELS

There are many between object-oriented testing levels and traditional testing levels. While object-oriented programming provides functionality not afforded by procedural programming paradigm – such as data encapsulation and reuse of objects – the ease of writing object-oriented programs does not translate to testing. In fact, many researchers have observed that testing programs written in an object-oriented language increases the effort required for adequate testing (Jain, 2008). In this section, four levels of testing are described. Typically, in traditional testing, there is unit and system testing. With object-oriented testing it is necessary to include two new levels, class testing and integration testing.

Unit Testing

Unit testing can be used in both object-oriented and traditional testing. Methods and routines are tested independently of each other in unit testing (Johnson, 1996). The defects or faults of other classes and functions should not impact unit testing (Roggio & Gordon, 2013). In traditional testing, tests can be made by defining inputs and observing to see if the output of a method or set of methods matches the expected outputs of the function. These functions or methods need to be independent units, units which do not call other methods or use common global data (Hayes, 1994). However, in object-oriented unit testing, the method cannot interact with other classes or be dependent on its class's methods. Testing individual methods is significantly more difficult. In fact, some authors state that unit testing cannot be deduced from one object's operations because (when isolated) one may not see the object's relations to other methods, the class's state, and other classes (Labiche, et al., 2000). Instead, it is suggested that unit testing be combined with integration testing (Hayes, 1993).

To actually accomplish unit testing on individual methods, several additional items must be tracked. The first is any attributes of the class that may be changed by calling the method. The second is that other methods in the class called by a particular method are determined to be correct. The third is that objects of other classes used by the method must be first tested and determined to be correct. This means that the testing levels are not in a linear order, and have to be determined from a different method. There are many different ways of determining levels such as the flow-graph-based and graph-based techniques mentioned earlier (Dechang, et al, 1994).

Another way of dealing with dependencies when trying to unit test is simulating the dependent classes. This is an extension of a testing technique known as writing drivers or stubs. Traditionally, drivers and stubs were written as "dummy" methods for dependent methods. In object-oriented testing, this is extended to entire classes. A driver is written when a class is dependent on another for data to process. A driver is usually used with a lower layer in a hierarchical development model. A stub is a method written that is handed data to process when the module that processes data has not been written yet, or when the module that has been written has not been tested. Stubs are often written when testing higher classes in a hierarchical design.

Class Testing

This version of testing involves testing methods as they relate to and interact with one another. Of course, because this is "class testing," it is only involved in object-oriented testing (Johnson, 1996). Some authors consider this to be object-oriented testing's version of unit testing (Johnson, 1996) (Labiche, et al., 2000). The reasoning behind this is that testing a class's methods in isolation, without any relation to other methods, is not actually useful for any nontrivial task. Methods are meant to interact. In any event, the purpose of class testing is to test how a single class's methods interact with one another. Again, this means that any classes referred to by an object's methods need to be tested thoroughly beforehand, or the dependent classes need to be simulated in some way.

Cluster Testing

Cluster testing involves extending class testing to verify that a group (cluster) of cooperating classes interacts correctly. According to Johnson**,** (Johnson, 1996), a cluster of classes is a group of classes that are dependent and cooperate with one another directly. Traditional testing does not appear to have a clear comparison. In order to do this, the cooperating classes must have previously been tested individually, through class and unit testing if possible. (See next paragraph)

Integration & System Testing
In traditional testing, integration testing tests methods together. This will include methods that are dependent on other methods or dependent on common global data (Hayes, 1994). For object-oriented programming, integration testing is an extension to cluster testing. In integration testing, the testing is extended to the system as a whole. The clusters are combined into the total system, which is then tested as a whole, with all the dependencies intact. Another, more specialized, case of integration testing is system testing which is running the whole system based on normal customer usage scenarios as close to the customer's environment as possible (Johnson, 1996).

7. OBJECT-ORIENTED FEATURES THAT AFFECT TESTING

There are many positive features of object-oriented programming, and although they make the paradigm very effective, these features make it more difficult to test. The seven factors described below have been mentioned by different authors as factors that affect the amount of effort needed for adequate testing (Khatri, Chillar, & Sangwan, 2011) (Badri & Toure, 2012) (Jain, 2008) (Yeresime, Jayadeep, & Ku, 2008). The seven factors are encapsulation, inheritance, polymorphism, cohesion, coupling, dynamic binding, and abstraction. As described below, each contributes to greater difficulty in designing tests (other than cohesion that eases it) over traditional testing. The feature will be described and then its effect on testing over traditional testing will be given.

Encapsulation
Encapsulation is used to restrict access to some of an object's attributes and methods. When a program is written procedurally, then this is not as much of an issue because programs are typically full units whose private or protected methods are not modified by outside programs. In object-oriented programs, it can become difficult to observe object interactions with encapsulation, especially when variables and methods are not visible outside a class (Khatri, et al., 2011). This restricted visibility means that it might be more difficult to be aware of an object's state, which is important because private fields and methods can be affected, such as with getters or setters. Testing is therefore more difficult when the state of the object is important for a class test case and strong encapsulation is used (Bhadauria, Kothari, & Prasad, 2011). Of course, it is also important for class and cluster testing because a class's or other classes' methods may influence an object's state. If part of that state cannot be observed, then it will be difficult to not only design test cases, but also to observe testing results.

If it is the case that an object is strongly encapsulated, it is important to find a way to verify that private fields are correct if they are modified by other classes (Jain, 2008). The ability to control a test's input may also be difficult because the initial state cannot be determined, either (Badri & Toure, 2012). This might mean creating new methods to display a class's state, which may or may not go against the class goals as designed (Badauria, et al., 2011). Perhaps the attribute was designed to be invisible to all objects due to security concerns.

Inheritance
It does not make much sense to talk about inheritance in the case of a procedural program. The only near comparison is the reuse of methods or structures, but this should not be as complex as in object-oriented programs. On the other hand, inheritance is a method of sharing attributes and behavior from pre-existing classes to other subclasses. When one class is a subclass of another, it does not guarantee that all the inherited methods are still correct if they have been verified in the superclass (Khatri, et al., 2011). The superclass being well tested will not mean that all the classes that inherit it will be correct. Any new methods or attributes that have been added to the subclass may affect properties inherited by the subclass. Yeresime et al (Yeresime, et al., 2012) describe an empirical measure of inheritance, Depth of Inheritance Tree or DIT. DIT refers to the maximum length of a path from a class to the root class in an inheritance tree (Badri & Toure, 2012). The deeper a class in the tree, the higher the number of methods that can be inherited; this makes its behavior more complex, more difficult to predict, hence more difficult to design effective test cases (Yeresime, et al., 2012).

These issues result from invisible dependencies between parent and child classes. A child cannot then be tested without its parent class because errors in behavior might easily propagate down the inheritance tree (Badauria, et al., 2011). Another issue may arise when an inherited method is changed in the subclass, but the

subclass has an untouched, inherited method that uses the changed method. The overridden method and untouched inherited method need to be tested. In the example ahead, getNum() of Child may be called, but it will return a value that would be unexpected by examining Parent. (See Figure 6)

```
public class Parent{
   int num;
   public void setNum(int n){
         num = n;
   }
   public void getNum(){
         return mult();
   }
  public int mult(){
         return num * 2;
   }
}// end class

public class Child extends Parent{
   public int mult(){
         return num * 4;
   }
}// end class
```

Figure 6 Inheritance Example

Yeresime et al also define a metric for measuring a different kind of complexity, Number of Children (NOC) (Yeresime, et al., 2012). NOC is the number of immediate sub-classes in a class hierarchy and is meant to be a measure of the influence a class may have over the system as a whole (Badri & Toure, 2012). This would be used as part of cluster and system testing to determine how much emphasis should be put into testing that particular class.

These two metrics, DIT and NOC, are taken from the Chidamber and Kemerer metric suite. They can be used to determine the overhead involved in testing. The advice given by Yeresime is that if DIT is greater than six, then design complexity is high and testing overhead can be large. If the NOC is similarly high, then the design of abstract classes is diluted. The abstraction of classes is not utilized or the designed abstract classes are too general. Yeresime et al, also state that these metrics are untrustworthy for determining faults themselves and cannot be used to measure fault-proneness (Yeresime, Et al., 2012). It is difficult to assign inheritance a precisely measurable metric at this point, as inheritance comes in many forms, and inheritance trees can become very complex. It is therefore important– while still using inheritance effectively – to try to keep inheritance as simple as possible.

Polymorphism

Procedural languages do not have a very good comparison to polymorphism. Polymorphism allows attributes of an object to take multiple forms or data types. In addition, an operation may return more than one type of data or may accept more than one type of data for parameters (Khatri, et al., 2011).

Polymorphism is crucial to object-oriented programming and helps make it versatile and reusable (Bhadauria, et al., 2011). But all the different forms an object may take should be tested. A class or group of classes should be designed well enough so that the overhead required to test is low. For example, a class such as shown in the next figure is not advisable:

```
public class Foo{
        Object o1;
        Object o2;
 }// end class
```

Figure 7 Object can Morph into any other Class.

The reason for this cautionary note is that *o1* and *o2* can take almost any form because Object is the superclass of all objects in the Java language. This would make testing a Herculean task as *o1* and *o2* could become almost any data type. Polymorphism should still be used, but the attributes of a class should be more limited and well-defined, in regards to both design and testing. As stated by Hayes (1993), "testing should be used to ensure that data abstraction and value restriction are implemented properly." In Figure 8, Shape is a class that has three subclasses, Triangle, Square, and Circle. There are still nine possible combinations that *shape1* and *shape2* can take, but this is much more manageable than the first example. The testing of attributes should be done in unit and class testing. Other testing concerns include methods with return values that are polymorphic as well as parameters that are polymorphic. This would more readily be accommodated in cluster or system testing.

```
public class Bar{
        Shape shape1;
        Shape shape2;
  …
 }// end class
```

Figure 8 Well Defined, Limited Attributes

According to Jain (Jain, 2008), the first major type of polymorphism is called "ad hoc polymorphism." This type of polymorphism is considered completely syntactical, that is, entities are polymorphic only because they share a name and do not have to be behaviorally linked. This can best be explained by examining its first subtype, overloading. Overloading refers to separate methods which share a name. These methods may have completely different parameters and method bodies. A group of overloaded methods can be treated as completely separate methods from one another during testing without any extra effort. The second type of ad hoc polymorphism, coercion, is a conversion from one class or data type to another during code execution. This is also fairly easy to test because the conversion type can be determined statically from code. For example, when 0.50 * someInt + 6.9 is executed, the integer someInt will be converted to a float or double to coincide with the data type of 0.50.

A second major type of polymorphism is called universal polymorphism. This is considered to be "true" polymorphism, and refers to an object being able to become many different data types (Jain, 2008). This is an umbrella for two subtypes, inclusive and parametric polymorphism. Inclusive polymorphism is polymorphism where a subclass can be used in place of a superclass (see *Dynamic Binding*). In parametric polymorphism a method or object can be written in a generic manner through parameters which are given a class value when the object is instantiated. In this way parameterized types are not "written in stone," which implies they are not dependent on any one class. An object or function can be used in different contexts without any conversion or run-time testing needed in this type.

These types of polymorphism should be taken into account for testing. Indeed, understanding all the interactions that can result from the polymorphic nature of some objects can be very difficult (but necessary) to keep in mind when developing test cases (Jain, 2008). Ad hoc polymorphism is less testing intense because the tests can be derived statically. Universal polymorphism, as the name might imply, is more difficult to test because the forms an entity can take may be wide-ranging.

Cohesion

Cohesion is a measure of the degree to which the methods of a class create a single, well-defined class (Khatri, et al., 2011). In procedural programming, cohesion refers how well a module of code (typically a file) belongs together as a single unit. Most of the rest of this discussion talks about how a class is cohesive in terms of instance variables. Procedural programs do not have instance variables, but instead information is passed between methods as parameters. Therefore, cohesion in the procedural realm is concerned with methods dealing with similar parameters and functionality. In OO, if a class is cohesive (its methods contribute to the class as a single unit) then the class is reusable, more reliable, and more easily understood. Cohesion is related to coupling; if there is high cohesion, there is low coupling and vice-versa (Khatri, et al., 2011). High cohesion means that the methods within a class are similar in the variables used and the tasks they perform. This means test data are easier to create and more easily understood. Low cohesion means that there are many different types of data that need to be defined for a specific class (Yeresime, et al., 2013). This complexity in design leads to higher costs of testing, and renders testing itself more error-prone.

Another defined metric for this testing factor is the Lack of Cohesion in Methods (LCOM). LCOM is defined as the mathematical difference between the number of methods whose instance variables are completely dissimilar, and the number of methods whose instance variables are shared (Yeresime, et al., 2012). See Figure 9.

Consider the three sets of instance variables for a class with three methods: 1: {a, b, c, d, e}l 2:{a, b, e}, and 3: {x, y. z}

Figure 9 Method Cohesion

Methods 1 and 2 have shared instance variables, and, therefore, have cohesion. However, 1 and 3 and 2 and 3 have no shared instance variables, and therefore no cohesion. In this case, the LCOM would be one (2 non-cohesive method pairs – 1 cohesive method pair). If LCOM is high, it means that a class is not cohesive (and might be a candidate for refactoring into two classes. At the testing stage, a class will need to have different testing sets for the different methods rather than one testing set for the entire class. This leads to confusion and overall complexity of the testing process.

LCOM is found in the same suite as DIT and NOC (the Chidamber and Kemerer metric su*ite*) (Yeresime, et al., 2012). The authors Badri and

Fadel (Badri & Toure, 2012) found that LCOM and lines of code (LOC) were the most predictive testing metrics over DIT, NOC, (both previously discussed) and CBO which is described below.

Coupling

Coupling is a measure of the dependency between modules. Strong coupling is undesirable for many reasons, chiefly of which is that it prevents the change of components independently of the whole. (Also, many feel strong coupling cohesion is the antithesis of highly-valued high cohesion, which arguably results in low coupling) – Strong coupling means that all (or many) of the methods coupled together need to be understood as a set, instead of each class operating as its own unit (Khatri, et al., 2011). Strong coupling negatively contributes to testing, because it implies that unit testing cannot be done effectively. In good design, coupling is kept to a minimum especially for a large or complex system where coupling could result in cluster and system testing absorbing the majority of testing resources (Badauria, et al., 2011).

An additional Chidamber and Kemerer metric is Coupling Between Objects (CBO) (Yeresime, et al., 2012). CBO is a count of the number of classes to which a class is coupled (Badri & Toure, 2013), and represents still another measure of complexity. High CBO leads to less reliability, and the higher interoperability between classes causes unit testing to be difficult (Yeresime, et al., 2012). However, some interoperability is generally required for object-oriented programming as objects need to be able to communicate in some way. This implies the necessity of cluster testing.

Other metrics for software complexity are efferent coupling (Ce) and afferent coupling (Ca). These come from the R. C. Martins metric suite (Yeresime, et al., 2012). Efferent coupling occurs between packages and is the measure of all the classes external to a package that are used within the package (See Figure 10). In contrast, afferent coupling between packages counts all the classes external to a package that are dependent on the classes within a package. (See Figure 11) In conjunction, these two help measure the stability of a package as a whole (Yeresime, et al., 2012), where stability is measured $I = \frac{Ce}{Ce+Ca}$ (Scale 0 to 1 with 0 absolute stability; 1 absolutely unstable.) Stability, in a sense then, is a measure of how well a package can adapt to change. In a testing sense, stability can imply how changes in one particular package might impact other classes. If this impact is high due to high instability, then

much regression testing must be done in other packages as part of cluster or system testing.

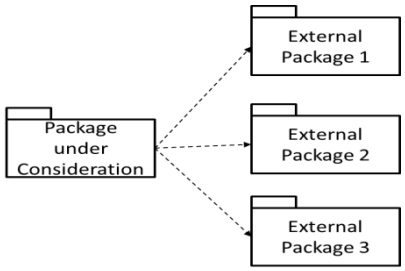

Figure 10 Efferent coupling (Ce)

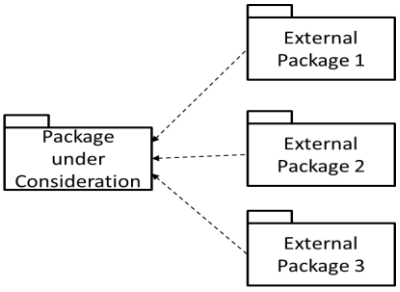

Figure 11 Afferent coupling (Ca)

Dynamic Binding

Dynamic binding is a result of either inclusive polymorphism or type parameterization polymorphism in some languages. For example, in Java the return type of a function or even the types of some fields can be decided at run-time rather than compile time: (Figure 12) Dynamic binding introduces concerns when deciding how to design test cases, because the exact data type of attributes cannot be known statically (Khatri, et al., 2011) .

```
public class Foo <E> {
   E field;
   public E getField(){
      return field;
   }
   public void setField (E field){
      this.field = field;
   }
}
```

Figure 12 Dynamic Binding is Determined at Runtime

The class in Figure 12 may be instantiated in many ways and with many data types through parameterization. Thus when designing test cases in parameter polymorphism, it may only be necessary to test based on how other classes in the program will instantiate Foo. Those data types that are used for parameterization in other classes must be considered for testing cases, but others need not be. This would be a part of cluster or system testing. Unit testing or class testing would be difficult to accomplish because it would not be known (without looking at the system as a whole) how Foo might be instantiated. The behaviors and properties of E might be incredibly varied. Class and unit testing should therefore probably not be done in this case of dynamic binding, due to its complexity as a unit. With the whole system, it can most likely be determined which data types E might take.

Inclusive polymorphism**,** in contrast, may be a simpler form of dynamic binding to test (Jain, 2008). This is because it is often known what classes inherit a superclass. In this way, it can be known what types an object may be bound to at run-time. All of these dynamic bindings must be included in a test case.

Abstraction

An abstract class is a type of class that cannot be instantiated. An interface is used as a template for other classes. If the class is just abstract and not an interface, it provides useful methods as well as variable fields (Khatri, et al., 2011). However, these defined methods cannot be tested directly and analysis is done from their subclasses, because one may not instantiate an abstract class or an interface in most languages (Badauria, et al., 2011). … This can lead to major overhead. If an abstract class is inherited by more than one class, how many of those child classes should be tested? Even if the child classes have not overridden the inherited methods, all would need to be tested because the abstract class itself cannot be tested, directly.

An R.C. Martins metric described by Yeresime et al is abstractness (Yeresime, et al., 2012). Abstractness is a ratio of the number of abstract classes/interfaces to the total number of classes in a package. This measure is used with the instability measure (see *Coupling*) to create a line graph (Yeresime, et al., 2012). (See Figure 13) These points along the "main sequence" line are considered to be balanced between abstraction and stability. These are well designed and more easily tested. This means that more abstract and unstable, the more difficult to test.

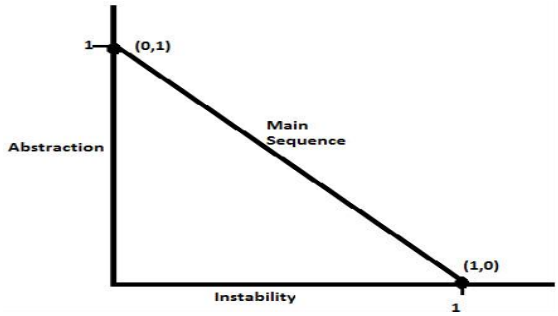

Figure 13 A-I Graph (Yeresime, et al., 2012)

8. CONCLUSION

Object-oriented testing is based not only on both the input and output of an object's methods, but also how that input and output may influence the object's state. . While the many beneficial features of the object-oriented paradigm are important, the increases in program complexity (sometimes in unintended and unseen ways) often negatively impacts testing in terms of effort and time. Some of these features, like cohesion help lower the amount of testing required, but others cause testing efforts to rise.

Traditional testing involves the viewing of input and output of a program in a procedural manner. Test cases tend to be one dimensional. In object-oriented testing, however, test cases are two dimensional, because changes in an object's state must be considered. Traditional testing involves both unit and system testing, while object-oriented testing requires class testing (for how the methods of a single object work together) and cluster testing (for how coupled objects change each other's' states). Thus, it is important to note that verification testing (the testing done by the developers) has been truly changed by the object-oriented paradigm, while validation (that done by the end-user) has not.

Moving forward, it will continue to be important to define more and better ways for testing object-oriented programs. Some already exist, but they are wide-ranging and there has been no major consensus as to what the best way to test is or what factors are most important in testing. Most focus on the fact that order to test object-oriented modules is not as definite as in traditional programs, where the order of tests follows a procedural path. In object oriented testing, an

object may send a message to another object at any time.

9. REFERENCES

Badri, Mourad, and Fadel Toure, "Empirical Analysis Of Object-Oriented Design Metrics For Predicting Unit Testing Effort Of Classes," Journal Of Software Engineering & Applications 5.7 (2012): 513-526.

Bhadauria, Sarita Singh, Abhay Kothari, and Lalji Prasad, "A Full Featured Component (Object-Oriented) Based Architecture Testing Tool" International Journal Of Computer Science Issues (IJCSI) 8.4 (2011): 618-627.

Gu, Dechang, Yin Zhong, and Sarwar Ali. "On Testing of Classes in Object-Oriented Programs" Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research

Hayes, Jane Huffman. "Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach," Notes in Computer Science, Vol. 858 (1994): 205-220.

Jain, Ajeet K. "Testing Polymorphism in Object-Oriented Programming." ICFAI Journal of Computer Sciences 2.4 (2008): pages 43-53.

Johnson, Jr., Morris S. A Survey of Testing Techniques for Object-Oriented Systems, Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative research (CASCON '96)

Khatri, Mrs. Sujata, Chhillar Dr. R. S., and Sangwan Mrs. Arti "Analysis Of Factors Affecting Testing In Object-oriented Systems," International Journal On Computer Science And Engineering 3 (2011): 1191.

Labiche, Y., Thevenod-Fosse, P., Waeselynck, H., and Durand, M.-H, "Testing Levels for Object-Oriented Software," Proceedings of the 22nd International Conference on Software Engineering, pages 136-145

Naik, Kshirasagar and Priyadarshi Tripathy Software Testing And Quality Assurance: Theory And Practice. John Wiley & Sons, 2008 pages 7-27

Turner, C.D. and Robson, D.J. "The State-Based Testing of Object-Oriented Programs," Software Maintenance, 1993 CSM-93, Proceedings., Conference on Software Maintenance pages 302-310, 27-30 Sep1993

Yeresime, Suresh, Pati Jayadeep, and Rath Santanu Ku "Effectiveness of Software Metrics For Object-Oriented System," Procedia Technology, Volume 6, 2nd International Conference on Communication, Computing &Security [ICCCS-2012] 420-42