Journal of Information Technology Management

ISSN #1042-1319

A Publication of the Association of Management

ENGAGING TESTERS EARLY AND THROUGHOUT THE SOFTWARE DEVELOPMENT PROCESS: SIX MODELS AND A SIMULATION STUDY

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

MICHAEL J. RACER
UNIVERSITY OF MEMPHIS
mracer@memphis.edu

SANDRA M. RICHARDSON
UNIVERSITY OF MEMPHIS
srchrdsn@memphis.edu

XIHUI ZHANG
UNIVERSITY OF NORTH ALABAMA
xzhang6@una.edu

ABSTRACT

Software testing is indispensable in ensuring software quality. Traditionally, testing has been viewed as a separate and distinct stage at the end of the software development process. However, testing activities have evolved from the "code and fix" process of executing a piece of software in an attempt to find coding errors, to a collaborative coordinated effort with testing activities embedded throughout the entire software development life cycle. The benefits of contemporary testing activities include: linking together of perspectives across the entire organization, development of a better software product with fewer errors, and reduced cost by avoiding or finding errors earlier in the development life cycle. In spite of an emerging view that testing activities should be included early and throughout the software development process, there is little research in the area of how this can be accomplished. This paper attempted to address this void by offering six models for engaging testers early and throughout the software development process. It also carried out a simulation study with the in-depth surveyed data from 13 software testing professionals, for the purpose of determining which of the six models would be best under different development environment circumstances.

Keywords: software development, software testing, model simulation

INTRODUCTION

As software is becoming critical to almost every organization, the art and complexity of software development has become a perennial topic of interest in both research and practice. The practice of software development has evolved steadily from its beginning half a century ago, and numerous methods and models (e.g., life cycle models and agile methods) have been proposed to enhance its efficiency and effectiveness. Royce [31] is widely recognized for introducing the first formal methodology for software development, now known as the waterfall methodology. Royce's waterfall model introduced a sequential process that emphasized systematic development and divided software development processes into separate and distinct phases, including requirements analysis, program design, coding, testing, and operations. Royce's model has been refined by organizations and researchers for decades, resulting in a vast number of waterfall model variations collectively referred to as the Software Development Life Cycle (SDLC) methodology.

SDLC models represent a structured approach to software development which has been praised for providing necessary order and control for large complex projects, and criticized for being inflexible and time consuming. Over the past four decades the SDLC models have been refined to cope with increasingly larger, collaborative, inter-organizational, and complex software development projects [2]. Today, new development models (e.g., progressive, iterative, and agile methods) have emerged and been proposed to tackle some of the criticisms of the SDLC methodology [36]. However, in spite of the seeming popularity of these new agile models and the persistent criticism of the SDLC, evidence suggests that the SDLC continues to be the preferred development methodology in contemporary organizations [2][21][22][30][36]. In fact, a recent Gartner study estimated that newer development methods such as agile development account for less than 15% of the software developed in organizations today [17].

Software testing is indispensable in ensuring software quality [7]. The inclusion of software testing activities in early waterfall models provided testament to the importance of testing a piece of software prior to its use within an organization. However, these early models placed software testing activities at the end of a sequential structured process relegating software testing activities to a "code and fix," or error finding approach [4]. Over time, it was recognized that the cost of finding an error after the development process had been completed was much more expensive than finding the error during the development process itself [3][4][5][13]. More recently, views on testing propose software testing activities as an integral part of the overall development life cycle [13][29][32] and view testing as more of a coordinated collaborative effort [6][19][20]. It has also been recognized that integrating software testing activities into earlier phases, if not all phases, of development provides benefits, including: appropriate time allocation and better scheduling of testing activities [28], more productive collaboration that links together all aspects of an organization that results in a better software product [6][20][27], improved project performance in terms of cost and cycle time [35], and cost savings by catching bugs earlier [3][5][37].

In spite of this recognition, software testing activities have been slow to move from the end of the development life cycle and into all phases of the development process. Relatively little research has been conducted in this area leaving a void in the current literature for the introduction and analysis of new formal models for engaging software testers early and throughout the software development life cycle. In this paper, we attempt to address this void. Specifically, we address a major research question: *How can testers be embedded in the software development life cycle to obtain the most beneficial testing results?* We address the question by first offering six formal models for engaging software testers early and throughout the software development process and then developing a simulation study with surveyed data from 13 software testing professionals, for the purpose of determining which of the six models would be best under different development environment circumstances.

This paper proceeds as follows. First, we provide a review of the software development and testing literature. Then, we propose six formal models for engaging testers early and throughout the software development process. Next, we describe the development and the analysis of a simulation study to test the models using surveyed data from software testing professionals. Finally, we offer a brief conclusion.

LITERATURE REVIEW

The software development life cycle has a rich history in the IS research literature. Research has illustrated the importance and evolution of the SDLC over the past four decades. Each of the SDLC's general phases (e.g., requirements, design, implementation, and testing) has evolved into individual research streams. However, while requirements, design and implementation each have been investigated extensively, research on testing activities has been less prevalent. While more contemporary SDLC models emphasize the importance of embedding testing activities early and throughout the development life cycle, little research has been conducted in this area.

In the following sections, we provide a review of the literature related to the SDLC and software testing activities to illustrate the co-evolution of both. This literature review reveals a void in the current literature related to the development and analysis of formal models for embedding testing activities early and throughout the entire software development life cycle.

Software Development Life Cycle

The art of software development is a perennial topic that both researchers and practitioners have grappled with for decades. Software development methodologies have evolved as a result. In the 1950's, there were only two steps in the software development process: an analysis step followed by a coding step [31]. Early systems development projects focused on hardware. Systems development was primarily dominated by engineers who tended to adopt linear processes that focused on hardware conservation, a philosophy consistent with the computing economics of the time. A decade later, in the 1960's, systems development processes began to change as people recognized that software was easier to modify than hardware and did not require expensive production environments to develop. The result was a shift toward a "code and fix" approach to development [4][25]. As organizational systems increased in complexity and numbers, the "code and fix" approach often resulted in unwieldy "spaghetti code" and frequent patches [3][4]. In reaction, the classic waterfall model was introduced by Royce in 1970 as an attempt to introduce structure and controls into the software development process.

The widespread adoption of the waterfall methodology resulted in the evolution of a framework for software development that incorporated important organizational issues into the development process, such as incorporating the stakeholder perspective, emphasizing requirements analysis, and the importance of testing activities [27]. New models divided the development process into well-defined phases, typically including analysis, design, coding, testing, and implementation [23]. Strict adherence and emphasis on the sequential nature of the SDLC phases persisted in the decades that followed, leading to a misinterpretation of the SDLC as an inflexible sequential process. This perspective was solidified in part by government process standards emphasizing a purely sequential interpretation of the model [4][24].

In the decades following its introduction, the waterfall methodology has continuously evolved as organizations have adapted it to meet individual and context specific needs [2]. However, the SDLC continues to be frequently identified as inflexible, time consuming, and costly [30]. As a result, many new models of software development have emerged, including modified SDLC models, progressive, iterative, and agile development methodologies. Interestingly, a search of the related research literature and professional journals would lead one to believe that the adoption of these newer methodologies were contributing to the quick demise of structured waterfall models [21]. However, current studies estimate that software developed using agile methods accounts for less than 15% of the software developed in organizations today [17], and that organizations continue to use traditional structured approaches for the majority of their development projects [2][21][22][30].

The stages of the SDLC have been refined over the past five decades. More recent models have tweaked the granularity of the initial models; however, most continue to include the basic phases of planning (problem and scope identification, financial implications, strategy), analysis (determine user needs), design (system specification), implementation (new system development, installation, integration, testing and training), and finally maintenance (ongoing operations and improvements to software). This general model is often refined and each of these phases made more or less granular, by breaking them into additional or fewer phases [18]. Many newer models break logical and physical design into two separate phases; some include testing as a separate phase. Other models emphasize business requirements, systems requirements, high-level logical design, detailed physical design, and implementation. However, most newer models continue to place testing activities near the end of the sequential life cycle.

The impact of software development teams on the quality and effectiveness of software products has been investigated in the research literature. Research related to the impact of team size on software quality and effectiveness has investigated coordination efforts and cost as well as communication issues. Results reveal that increasing team size does not necessarily increase costs and if effectively managed can improve outcomes [11][14][26]. Investigation into the coordination of project team members illustrated the need to create a standard understanding among developers, testers, and managers to achieve quality software [6][9]. Research also reveals that effective coordination, or management, of team members' expertise (knowledge management) positively impacts outcomes [10]. Early models proposed that effective quality controls for software development required a division of labor and responsibilities across developers and testers, reinforcing the notion that testing activities should be isolated and conducted at the end of the development process. Dahlbom and Mathiassen [8] supported this notion by proposing that independence is required between developers and testers in order to avoid self deception in having developers evaluate their own work. However more recently, it has been suggested that different individuals on a team can have different goals and responsibilities, resulting in mutual interdependence, and allowing for the benefits of collaboration among team members throughout the entire process [6]. More contemporary models suggest that the benefits of testing early and often place less emphasis on independence and division of labor [4][6][13][27][34][36][37]. In spite of this changing philosophy regarding testing activities, current research has not explored the development of new formal models for embedding testing activities throughout the life cycle. This void provides an opportunity for researchers to develop formal testing models that analyze the impacts of embedding testers over the development life cycle.

Software Testing

One of the most significant criticisms of traditional structured SDLC methodologies is the placement of testing activities at the end of the development life cycle. Early definitions of testing describe testing as simply being "the process of executing a program with the intent of finding errors" [25, p. 16]. The placement of testing activities at the end of the life cycle often results in increased cost associated with software development [3][34] as "finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements or design phase" [5, p. 135]. Relegating testing activities to a "hand-off" activity at the end of the development process can result in a number of problems, including: poor planning and tight schedules for testing work leaving too little time to fix code and design flaws [6][13], and negatively impacting decisions determining if software products meet requirements given there is not enough time to systematically search, judge requirements satisfaction, or determine when and how to stop testing activities [6][15].

Contemporary views of testing emphasize testing as prevention rather than correction. Hetzel [16] was the first to illustrate this trend with his definition of software testing as activities aimed at evaluating the capability of a system and determining if the system meets the documtented organizational and user requirements. In addition, Gelperin and Hetzel [13] captured the evolution of testing by classifying the history of testing activities, including: debugging-oriented testing (until 1956), demonstration-oriented (1957-1978), destruction-oriented (1979-1982), evaluation-oriented (1983-1987), and prevention-oriented (1988-). This classification illustrated a slow but continuous trend of software testing slowly emerging to more importance in the development process itself.

In the 1990's, new concepts of software testing as part of the overall software development life cycle started to emerge. Dalal et al. [9] captured this new view of embedding software testing as an integral part of every stage of the development life cycle, and considered software testing as a coordinated effort that is part of the overall software engineering process. Software testing was identified as an important part of a cooperative process where multiple actors (testers, designers, programmers, etc.) link together different parts of an organization to accomplish a collective set of tasks over the entire life cycle [6][20]. Software testing activities were no longer an error finding activity after the fact, but were seen as prevention activities that not only reduced cost but improved the quality of the software that was developed [35].

The introduction of formal software development methodologies, such as the waterfall method, established clearly defined phases of software development and testing activities were decidedly at the end of this sequential process [4][31]. The emphasis on coordinated efforts brought attention to this placement that often resulted in software testing actitives being the poorest planned part of the development process resulting in less effective results than could be realized if software testing were moved earlier in the development life cycle.

More contemporary views of testing emphasize the importance of integrating testing activities into each phase of the overall development life cycle [6][9][13][36]. Pyhäjärvi and Rautiainen [29] argue that testing is "an integral activity in software development" and recommend that "testing should be included early in software development" (p. 33). Schach [32] also suggests that "testing should be performed throughout the software life cycle" (p. 277) and predicts that "the future role of testing will be to prevent faults rather than to detect them" (p. 278). Several empirical studies have shown that engaging testers earlier and throughout the software development process is beneficial to a project team's performance. Waligora and Coon [35] present quantitative evidence that, by starting testing earlier in the development life cycle, project performance, in terms of cost and cycle time, is improved without sacrificing the overall quality of the end product. As today's organizational environment rapidly evolves into an increasingly networked environment of outsourcing relationships, alliances, and partnerships; the demand for cooperative, collaborative, and interpersonal testing activities is emerging [6][27].

Current research does not offer formal models for embedding testing activities early and throughout in the software development life cycle. In the following section, we describe the development of six formal models, based on our experiences and the existing literature related to software development and testing.

SIX MODELS FOR ENGAGING TESTERS EARLY AND THROUGHOUT THE SOFTWARE DEVELOPMENT PROCESS

As captured by the literature review described in the previous section, the SDLC has been widely used and researched over the past several decades. As a result, there is not a single SDLC model that serves as the standard model. Researchers and organizations have adjusted the granularity of phases and have also reorganized the activities included in each phase. Therefore, it is necessary for us to define the SDLC model that will be used in this project. We adhere to generally accepted SDLC phases and activities that have persisted in the literature across the decades. For the purposes of this project, we will define five software development process phases or stages, as follows:

- Business Requirements: The set of specifications of what the business unit expects the application to accomplish.
- Systems Requirements: The systems analysis stage in which the business requirements are translated into graphical formats that show processes and data flows.
- High-Level Design: The specification of the code modules and their functions, and the flow of data among the code modules.
- Detailed Design: The design of the functions within each code module.
- Implementation: The programming of each code module.

Having established a software development process framework with which to work, we now turn to the nature of the participation of the testers at each software development process stage. This concept can be broken down into two possibilities, as we noted earlier, one or both of which can be practiced. One, which is applicable at all of the stages, is the idea of testing the output of the stage. Do the requirements make sense, do they meet a set of accepted standards, and what are those standards? Do the diagrams that result from the systems analysis stage flow correctly and make sense? Have the systems analysis diagrams been constructed to meet accepted standards and what are those standards? Are the program design specifications and the database design acceptable?

The second possibility regarding the nature of the participation of the testers is directed toward the creation of systems that will lend themselves to being more easily and effectively tested. This would begin in the systems analysis stage and significantly impact the systems design stage. This could have profound implications in systems testing, in streamlining the process in general, and in specifics such as determining the test datasets to use.

There are a number of factors to consider in formulating a model to use in integrating testers into the application development team and process. An initial factor is whether a company believes there is value in embedding testers at all levels of the software development process. As we have stated, we believe that the arguments for doing so are compelling and so for our purposes we will assume that this is the case.

Assuming there is significant value in embedding testers earlier in the software development process, one factor in deciding which model to use is the skill set of the individuals in the testing department or organization. We will assume that any tester assigned to represent the testing organization in an application development project is skilled in testing in at least one of the application development stages. This can even be extended back into the education and training backgrounds of the individuals. It is as difficult to imagine a tester without a business background leading a business requirements review, as it is imagining a tester without a programming background leading a code review. Pursuing this further, the Tester Embedding Model chosen will also depend on whether the company's testers tend toward breadth of skill or depth of skill. Are the individual testers expected to have skills that range from business to technical skills or are their skill sets expected to be narrowly focused?

Another factor is the amount of resources the company is willing to invest in testing. This certainly will depend on the company's commitment to testing and on the size of the application development project. Generally, in this regard, more would seem to be better; however, even a company that takes testing seriously would not want to overwhelm the application development teams with testers.

With the previous discussion as background, we propose six "Tester Embedding Models" for embedding testers early and throughout the software development process.

Model 1: "The Single Tester Model"

As the name implies, in The Single Tester Model, one tester is assigned to an application development project and stays with it through all of its stages (see Figure 1). This has the advantage of continuity as one person begins learning about the project from the very beginning of the business requirements phase and continues building her project knowledge through each successive stage. The disadvantages of this model include a project over dependency on one person and the expectation that the one person must be well-versed in a breadth of skills ranging from requirements analysis to systems design and programming. Assuming you can find such a person, if she gets sick or leaves the company, the project is left in the lurch. And, if The Single Tester Model is attempted without a sufficiently broadly skilled tester, then clearly the principle of having strong testing expertise at each development stage will have been defeated.

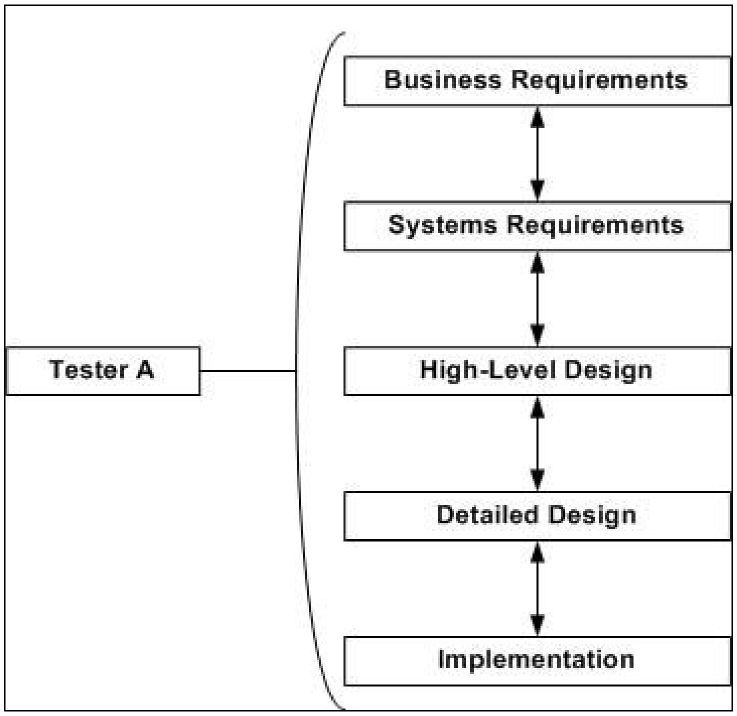

Figure 1: The Single Tester Model

Model 2: "The Specialist Model"

If The Single Tester Model is one extreme, then The Specialist Model is the other extreme. In The Specialist Model, a different, highly specialized tester works on the application development project in each of its stages (see Figure 2). Presumably, each tester is a true expert in the work being done at their particular development stage and thus the advantage is the level of testing expertise that can be applied at each stage. Conversely, each tester does not have to possess a broad skill base. However, there are some disadvantages as well. One disadvantage is that each Specialist must learn the nature and details of the project when they cycle onto the project. Another disadvantage is the lack of communication between the testers in the different stages. Indeed, it is this lack of communication that inspires certain aspects of the next four models.

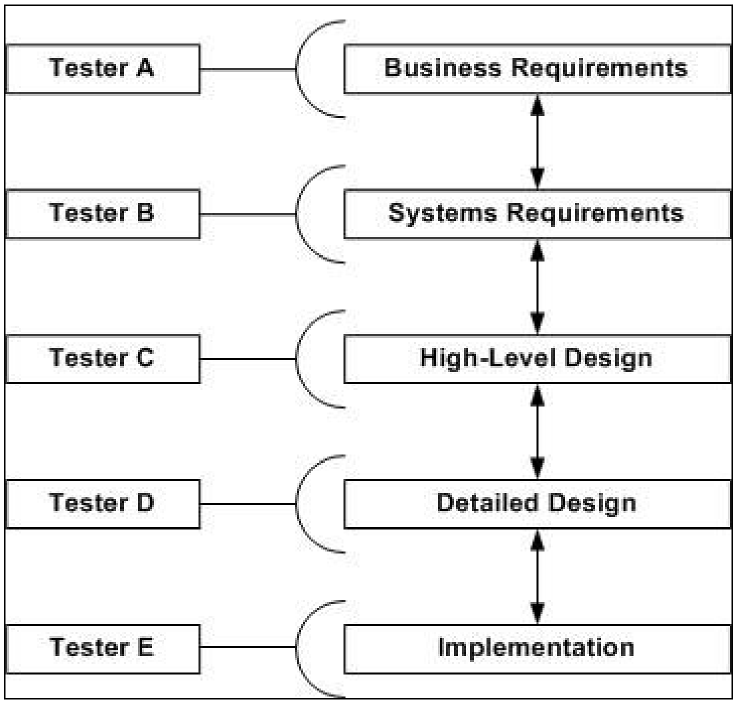

Figure 2: The Specialist Model

Model 3: "The Leapfrog Model"

The Leapfrog Model is designed to overcome some of the problems associated with both The Single Tester Model and The Specialist Model (see Figure 3). The Leapfrog Model begins with Tester A, who is a requirements testing specialist, working on the Business Requirements stage. Tester A continues working on the project in the Systems Requirements stage, where she is joined by Tester B, whose expertise is more geared towards systems analysis and high-level systems design. As both Testers A and B work on the Systems Requirement stage, Tester A is able to gradually transfer her project knowledge to Tester B. At the end of the Systems Requirements stage, Tester A leaves the project. At the beginning of the High-Level Design stage, Tester B is joined by Tester C, whose expertise is focused on both high-level and detailed systems design. Similarly, at the end of the High-Level Design stage, Tester B leaves the project and at the beginning of Detailed Design stage, Tester C is joined by Tester D, whose expertise is focused on detailed program design and programming. Tester C leaves the project at the end of the Detailed Design stage. An advantage of The Leapfrog Model includes having two testers working on each development stage except for the first and last stages. At each of the intermediate stages there is the opportunity for one tester to gradually transfer her project knowledge to the next tester. However, as with The Specialist Model, The Leapfrog Model assumes the availability of a stable of relatively specialized testers and, with two testers involved at each of the intermediate stages, it is even more resource intensive.

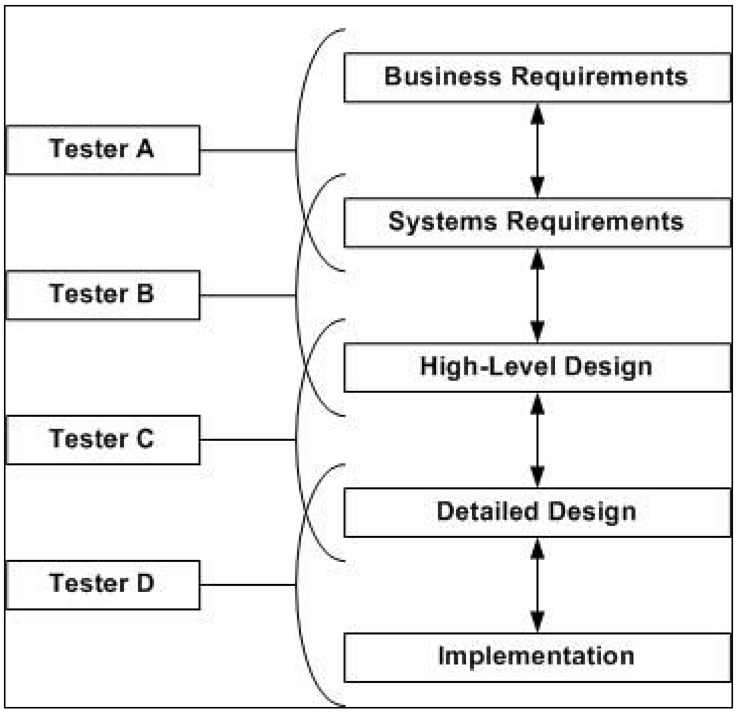

Figure 3: The Leapfrog Model

Model 4: "The Balanced Bifurcated Model"

In the Balanced Bifurcated Model, there are two testers, A and B (see Figure 4). Tester A has a broadly-based systems analysis background that extends to requirements on the one end and to high-level systems design on the other. Tester B has a programming background that includes the higher levels of systems design. In this model, Tester A begins at the Business Requirements stage and stays with the project through the High-Level Design stage, after which she leaves the project. Tester B joins the project at the High-Level Design stage and continues with it to its conclusion in the Implementation stage. With both Testers A and B working together in the High-Level Design stage, they have the opportunity to transfer project knowledge from A to B. Their skill bases must be broader than those of the testers in either The Specialist Model or the Leapfrog Model, but not as broad as the testers in The Single Tester Model. While there is a shift back toward the problem of over-dependence on individuals as in The Single Tester Model, there is also not as much of a resource drain as in either The Specialist Model or the Leapfrog Model.

Model 5: "The Top-Loaded, Unbalanced Bifurcated Model"

The difference between The Balanced Bifurcated Model and the two Unbalanced Bifurcated Models is the point of hand-off of responsibilities. The principle of the Top-Loaded, Unbalanced Bifurcated Model is that one tester, Tester A, will work with all aspects of the project through and including the Detailed Design stage (see Figure 5). Then, Tester B will join her in the Detailed Design stage and be responsible for testing in the Implementation stage. This model heightens the personnel dependency issue, plus Tester A must be very broadly based. The clear advantage of this model is it provides specialized testers whose sole purpose and total focus is to look at the detailed design and then work in the highly technical pursuit of code testing.

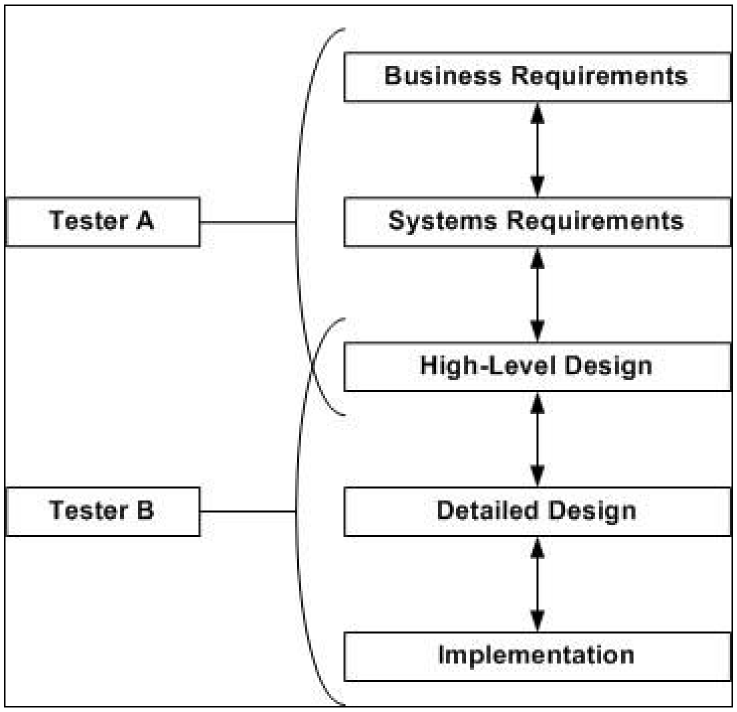

Figure 4: The Balanced Bifurcated Model

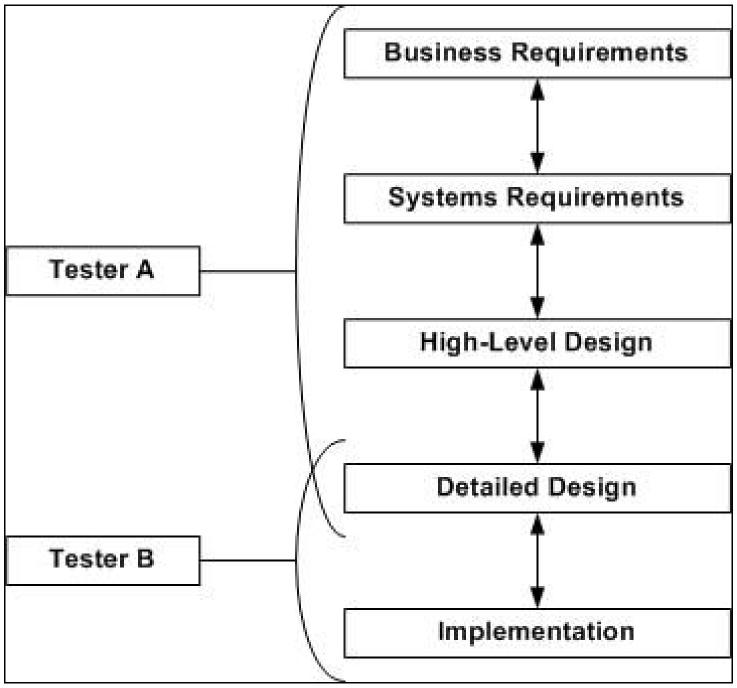

Figure 5: The Top-Loaded, Unbalanced Bifurcated Model

Model 6: "The Bottom-Loaded, Unbalanced Bifurcated Model"

In the Bottom-Loaded, Unbalanced Bifurcated Model, the point of hand-off is the Systems Requirements stage (see Figure 6). That is, Tester A handles the Business Requirements stage and the Systems Requirements stage. He is joined in the Systems Requirements stage by Tester B who begins working in this stage and then follows the project to its conclusion in the Implementation stage. In many structural respects it is similar to The Top-Loaded, Unbalanced Bifurcated Model, except that now, one tester, Tester A, is focused on business requirements, and the other tester, Tester B, begins with system requirements and then proceeds through both of the design stages and the implementation stage.

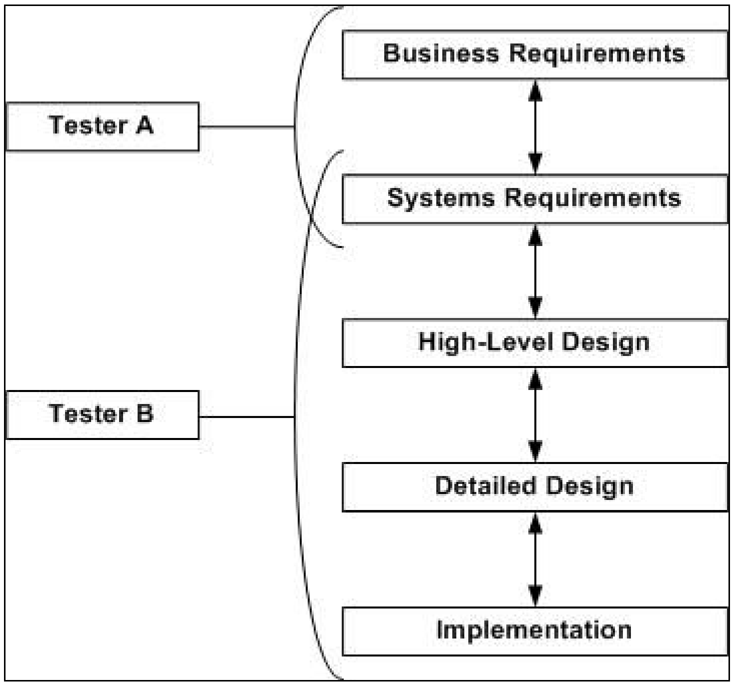

Figure 6: The Bottom-Loaded, Unbalanced Bifurcated Model

There are two possible considerations that could lead to variations in some or all of Models 1-6. One is that as a practical matter, depending on the size of the IT organization, the scope of the development project, and the company's dedication to testing, the testers described in any of the six models may well serve as "test leads" and bring additional testing personnel into the application development process as required. In all cases, this could obviously be simply a matter of handling the volume of work at hand. Also, in all cases, this decreases the dependency on only one or two people at any given development stage. Naturally, it also increases the amount of resources expended in testing. In addition, in Model 1, The Single Tester Model, and to a lesser extent in Models 4, 5, and 6, the three bifurcated models, it could involve calling in specialists to supplement the skills of the test leads in those models.

The other consideration has to do with the overlap between the testers at the various intermediate stages of development. Models 1 and 2 have no overlap. If we eliminate the overlap in Model 3, The Leapfrog Model, it effectively reverts to Model 2, The Specialist Model. The issue is whether in Models 4, 5, and 6, the three bifurcated models, the overlap could be eliminated. This has the advantages of reducing the expenditure of resources and of not requiring the testers to be quite so broad in their skill sets. On the other hand, there will clearly be a cost in losing the transfer of project knowledge from one tester to the next that is provided by the overlap. Perhaps the elimination of the overlap is most appealing in Model 5, the Top-Loaded, Unbalanced Bifurcated Model. Eliminating the overlap in Model 5 would isolate Tester B who is responsible for code testing. Historically, this responsibility has in many organizations been the only real job for testers. Even in those instances in which testing is embedded early and throughout the software development process, the nature of code testing and its specific and highly technical techniques make it a candidate for being largely separable from the other software development process stages. In fact, as a practical matter in today's IT environment, code testing can be looked upon as a candidate for outsourcing, which would fit the variation that we might now call the Unbalanced Bifurcated Model Without Overlap.

SIMULATIONS OF THE SIX TESTER EMBEDDING MODELS

Simulation Background

One of the most widely-used tools in analysis today is simulation. Simulation is "the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system" [33, p. 7]. Some have argued that simulation is indispensable as a means of approaching and solving real-world problems [1][12]. The fundamental goal in simulation is to mimic the essential elements involved in a process. A simulation model is very user-friendly, in the sense that the components of the simulation reflect actual components of the process being modeled. Consequently, simulations are extremely robust. Simulations allow for the collection of data at all stages of the process, so hidden factors might be easily revealed. And the most appealing attribute of simulation is the "what if..?" capability. That is, the user has the ability to alter conditions in the model, identify how the changes influence the outcomes, and project that same behavior to the real-world. Once the fundamental model is in place, the potential for investigating these alternative scenarios is practically limitless.

Within the context of software testing, the progress of the development of a piece of software from conception to completion is a good example of the use of simulation. From the model development standpoint, there may be components in the simulation that mimic contributors at every stage (business requirements, systems requirements, high-level design, and detailed design) and processes developed that show how these various factors interact and influence the quality of the final implementation. From the quality standpoint, we may capture the testing layer, and show how the performance of the various testers embedded throughout product development ultimately and collectively influence the final product.

The "what if?" standpoint is often the most significant element of a simulation. For example, a simulation model can be developed to show how a change in testing strategy (e.g., single-tester vs. specialist) can influence both costs and product quality. If we can understand the system better, we can make decisions on the front end of the suitability of a particular tester framework in a particular set of development environment circumstances.

While econometric and statistical models have capabilities to capture and explain some very basic relationships, they lack the capability to address detailed relationships within the process. For either of these models, it is extremely difficult to reveal the nature of interactive relationships (e.g., "How do certain business requirements influence the quality of the product when a bifurcated testing system is used?"). Compounding this, as econometric and statistical models increase in complexity, their comprehensibility declines - the model becomes less understandable to the original user, and consequently, less defensible.

In summary, the fundamental value of a simulation lies in these three characteristics:

- Dynamic: a simulation truly mimics the real-world process itself, allowing for in-depth, yet understandable modeling.
- Interactive: a simulation allows the user to capture the impact of complex relationships.
- "What if?": a simulation allows the user to "witness" the impact of a change in conditions, within the computer, rather than an expensive, impractical, real-world test.

The key to simulating the alternative testing environments is understanding the system components and their attributes, and how those attributes interact. Some of those basic components and their attributes are the software length, complexity, and number of programmers required. In addition, there is the experience and expertise of the tester, and the nature of the testing environment (i.e., the variations expressed in the six tester engaging models). Finally, there are the issues surrounding the errors to be found in the testing, such as their location, their severity, and the period at which they were introduced (business requirements, systems requirements, high-level design, detailed design, and implementation.)

Scenario Analysis

Empirical data on the employment and evaluation of software testing is sparse. As a result, we elected to rely on a combination of survey considerations and special case analysis. Given the phased structure of software development, in order to do some comparative analyses, we chose three test scenarios:

- Error rates based on surveyed expectation
  - This scenario considers user experiences, and hence provides a reasonable basis for an initial evaluation
- Error rates decreasing by phase
  - This scenario captures the worst-case environment, particularly if such errors are not recognized immediately
- Error rates evenly distributed over each phase
  - By assuming errors occur evenly throughout the development process, this scenario gives an "average-case" study, the results of which can be analyzed against the other two scenarios

Given an error within the code, the next step in the process is to discern the point at which the error is caught and corrected. This is the point at which the various testing strategies will affect performance. Again, survey results were used to determine the likelihood of recognizing a particular type of error at a given test stage. With this combination of scenarios, we will be able to show how the various testing schemes influence the development and testing process, as well as the final product quality and cost.

A Survey of Software Testing Professionals (Error Rates): An initial data set is required for all simulation programs. We supplemented information from the literature with data collected from a survey of 13 software testing professionals in order to develop an initial data set for the simulation. The addition of surveyed data provides a more robust initial data set for our simulation than relying on the literature alone.

In developing the initial data set, we surveyed 13 software testing professionals. Information related to the individuals who completed the survey was kept anonymous and the results of the surveys were kept confidential. Each of the 13 individuals who completed the survey had over 10 years of testing experience.

One of the survey questions we asked the participants was the likely location of software errors. This provided a basis for a "representative" environment to investigate. In this scenario, the likelihood of an error occurring at each stage was estimated to be:

- Business Requirements: 56%
- Systems Requirements: 19%
- High-Level Design: 9%
- Detailed Design: 4%
- Implementation: 12%

Our simulation involved 250 replications. That is, based on the above distribution, we created 250 software "cases," with each containing an error at some level of the development.

The simulation then proceeded to identify the error. The likelihood of identifying an error at each phase was a function of that phase, and the phase in which the error was created. This leads to a matrix of values, P, where P_{ij} is the probability of finding an error created in phase i during phase j, when $j \geq i$; 0, when $j < i$.

The matrix P was created, based on the survey of 13 software development experts. Note in particular that the P matrix is a function of the testing environment. Each of the six models evaluated would promulgate a different value for the P_{ij}'s. This is the essence of our analysis – to investigate the effectiveness of error recognition for each of three test scenarios, given a particular testing scheme. The diverse nature of the three scenarios, along with the varied impacts of testing schemes will be revealed in the simulation results.

The evaluation framework is at two levels: Macro View and Conditional View. The Macro View indicates, for each Model, the percentage of errors identified at each phase. Note that this then is a function of both the error generation scheme as well as the Model. As such, we may compare Model performances within a given error generation environment, and provide analysis on the relative merits of each Model. Since the analysis is performed on three variant error generation schemes, one may also recognize the manner in which the anticipated error environment might influence the choice of a tester Model.

The Conditional View of evaluation answers a slightly different, more focused question: Given an error occurs in Phase i, what is the expected proportion discovered in Phase j? Again, this is a function of the Model itself, and provides a deeper insight into the value of the various tester set-ups. In particular, a cost-benefit analysis could be applied to determine the preferred strategy to employ.

The baseline case, which used only data provided by those surveyed results in Table 1, indicates the probability of errors located in each phase, for each of the six models tested.

Table 1: Distribution of Recognition Location by Testing Model - Baseline Study

		Testing Model					
		MODEL1	MODEL2	MODEL3	MODEL4	MODEL5	MODEL6
Phase of recognition	1	**0.144**	0.14	0.104	0.096	0.096	0.104
	2	0.268	0.276	0.236	0.268	**0.352**	0.324
	3	0.296	0.268	0.328	**0.348**	0.264	0.324
	4	0.212	0.228	**0.244**	0.196	0.228	0.188
	5	0.08	0.088	0.088	**0.092**	0.06	0.06

Note in particular the highlighted values, which indicate the maximum value in each row. This suggests that MODEL1 was more likely to recognize an error in the first phase, while MODEL5 was more likely to identify those problems in phase 2. Such information would be very important in deciding which testing model to implement. This would primarily involve a cost-benefit analysis. If the cost of not recognizing an error until later phases were very high – indicating significant re-work or redevelopment, then MODEL1 or MODEL5 would be preferable, since they identify errors earlier in the process. Note that each of the models was evaluated with respect to the same 250 simulated replications.

In addition to the summary table above, we also collected the following statistics, indicating the proportion of errors found in each phase, conditioned on the location in which the error originated (see Table 2). Note that this information takes into account the surveyed distribution of where the error is located along with the *P* matrix mentioned above – which was also survey-generated.

Such information allows us to further pinpoint the efficacy of each testing Model, with respect to the error profile. For example, suppose that our company is prone to making errors at Phase 1, Business Requirements. (This is the profile of this simulation, in which 59% of all errors originate in Phase 1.) MODEL3 tends to identify these errors prior to Implementation, with only 1% of our trials not being recognized beforehand. All other models allowed two to three times as many cases to reach Implementation. This suggests that MODEL3 might well be the testing model of choice if it is vital to avoid problems at Implementation, resulting from Business Requirements errors.

Table 2: Detailed Phase-based Error Recognition - Baseline Study

MODEL		Probability (error is recognized in…)				
MODEL1		1	2	3	4	5
Given error made in…	1	0.23	0.30	0.28	0.16	0.02
	2	0.00	0.24	0.39	0.31	0.07
	3	0.00	0.00	0.33	0.50	0.17
	4	0.00	0.00	0.00	0.20	0.80
	5	0.00	0.00	0.00	0.00	1.00
MODEL2		1	2	3	4	5
Given error made in…	1	0.21	0.38	0.23	0.15	0.02
	2	0.00	0.18	0.38	0.29	0.15
	3	0.00	0.00	0.17	0.58	0.25
	4	0.00	0.00	0.00	0.17	0.83
	5	0.00	0.00	0.00	0.00	1.00
MODEL3		1	2	3	4	5
Given error made in…	1	0.18	0.32	0.32	0.17	0.01
	2	0.00	0.21	0.33	0.35	0.11
	3	0.00	0.00	0.28	0.44	0.28
	4	0.00	0.00	0.00	0.00	1.00
	5	0.00	0.00	0.00	0.00	1.00

MODEL4		1	2	3	4	5
Given error made in…	1	0.19	0.35	0.28	0.17	0.02
	2	0.00	0.24	0.40	0.29	0.06
	3	0.00	0.00	0.41	0.36	0.23
	4	0.00	0.00	0.00	0.60	0.40
	5	0.00	0.00	0.00	0.00	1.00
MODEL5		1	2	3	4	5
Given error made in…	1	0.17	0.30	0.31	0.18	0.03
	2	0.00	0.24	0.32	0.33	0.12
	3	0.00	0.00	0.21	0.57	0.21
	4	0.00	0.00	0.00	0.71	0.29
	5	0.00	0.00	0.00	0.00	1.00
MODEL6		1	2	3	4	5
Given error made in…	1	0.18	0.34	0.33	0.12	0.03
	2	0.00	0.28	0.42	0.24	0.05
	3	0.00	0.00	0.21	0.38	0.42
	4	0.00	0.00	0.00	0.25	0.75
	5	0.00	0.00	0.00	0.00	1.00

Another significant result is in comparing MODEL5 and MODEL6. Suppose that an error is introduced in Detailed Design. MODEL5 is almost three times as likely to identify that error immediately, while MODEL6 will generally not recognize that error until Implementation.

Balanced Error Rates: In this second case, we altered the error origination probabilities, suggesting that the likelihood of an error occurring at each stage would be:

- Business Requirements: 20%
- Software requirements: 20%
- High-Level Design: 20%
- Detailed Design: 20%
- Implementation: 20%

The summary probabilities are shown in Table 3.

Table 3: Distribution of Recognition Location by Testing Model - Balanced Error Study

		Testing Model					
		MODEL1	MODEL2	MODEL3	MODEL4	MODEL5	MODEL6
Phase of recognition	1	0.048	0.032	0.04	0.028	0.036	0.024
	2	0.16	0.116	0.136	0.228	0.14	0.184
	3	0.248	0.292	0.276	0.236	0.256	0.244
	4	0.296	0.312	0.332	0.32	0.352	0.34
	5	0.248	0.248	0.216	0.188	0.216	0.208

We again highlight the maximum value in each row, indicating the Model most likely to recognize an error at that phase. Two perspectives are noteworthy. We may, at this point, compare the same MODEL over this scenario and the previous one. For instance, note that MODEL1 recognized 14% of the errors in Phase 1, for the first scenario, and roughly 5% in this scenario. This result reflects both the capabilities of the testing Model, as well as the error environment.

The second perspective, as before is in comparing two models. If we have a fairly good understanding of whether the origin of errors follows the first scenario or the second, we would be in a better position to select the appropriate testing model. In this scenario for instance, MODEL4 recognizes 3-6% more errors prior to Implementation than any other model.

We again created the conditional matrices, indicating the likelihood of locating an error, given the origin. Again, it is noteworthy to compare to the previous scenario. For example, suppose that your company consistently employed MODEL6, but did not have a good idea of the distribution of error occurrences. If we compare Table 2-MODEL6 and Table 4-MODEL 6, we will notice a significant change in performance with respect to identifying problems prior to Implementation. For errors originating in Phase 1, the two scenarios are comparable (3% vs. 2%). The same is true for errors originating in Phase 2 (5% vs. 6%). However, for errors originating in Phase 3, the results are significantly different (42% vs. 14%). Likewise for errors in Phase 4 (75% vs. 52%). We may use this information to promote consideration of some alternative to MODEL6 if we believe more errors are likely in Phases 3 or 4 because the delay of recognizing them in Implementation is expensive.

Declining Error Rates: In this third case, we altered the error origination probabilities, suggesting that the likelihood of an error occurring at each stage would be:

- Business Requirements: 40%
- Software requirements: 30%
- High-Level Design: 10%
- Detailed Design: 10%
- Implementation: 10%

In this scenario, we suggest that errors tend to be front-loaded, that is more likely to occur early in development. The summary probabilities are shown in Table 5, and the detailed probabilities are shown in Table 6.

Table 4: Detailed Phase-based Error Recognition - Balanced Error Study

MODEL		Probability (error is recognized in…)				
MODEL1		1	2	3	4	5
Given error made in…	1	0.21	0.19	0.31	0.21	0.07
	2	0.00	0.28	0.47	0.18	0.08
	3	0.00	0.00	0.27	0.46	0.27
	4	0.00	0.00	0.00	0.56	0.44
	5	0.00	0.00	0.00	0.00	1.00
MODEL2		1	2	3	4	5
Given error made in…	1	0.18	0.30	0.30	0.20	0.03
	2	0.00	0.22	0.30	0.33	0.15
	3	0.00	0.00	0.29	0.51	0.20
	4	0.00	0.00	0.00	0.33	0.67
	5	0.00	0.00	0.00	0.00	1.00
MODEL3		1	2	3	4	5
Given error made in…	1	0.13	0.32	0.32	0.16	0.07
	2	0.00	0.24	0.47	0.18	0.11
	3	0.00	0.00	0.36	0.47	0.17
	4	0.00	0.00	0.00	0.49	0.51
	5	0.00	0.00	0.00	0.00	1.00
MODEL4		1	2	3	4	5
Given error made in…	1	0.15	0.36	0.33	0.09	0.07
	2	0.00	0.22	0.42	0.30	0.06
	3	0.00	0.00	0.28	0.47	0.25
	4	0.00	0.00	0.00	0.48	0.52
	5	0.00	0.00	0.00	0.00	1.00
MODEL5		1	2	3	4	5
Given error made in…	1	0.30	0.21	0.23	0.19	0.06
	2	0.00	0.24	0.34	0.34	0.08
	3	0.00	0.00	0.34	0.52	0.14
	4	0.00	0.00	0.00	0.69	0.31
	5	0.00	0.00	0.00	0.00	1.00
MODEL6		1	2	3	4	5
Given error made in…	1	0.13	0.34	0.30	0.21	0.02
	2	0.00	0.20	0.46	0.27	0.06
	3	0.00	0.00	0.33	0.52	0.14
	4	0.00	0.00	0.00	0.48	0.52
	5	0.00	0.00	0.00	0.00	1.00

Table 6: Detailed Phase-based Error Recognition – Declining Error Study

MODEL		Probability (error is recognized in…)				
MODEL1		1	2	3	4	5
Given	1	0.24	0.32	0.23	0.15	0.06
	2	0.00	0.22	0.32	0.29	0.18
	3	0.00	0.00	0.32	0.38	0.30
	4	0.00	0.00	0.00	0.86	0.14
	5	0.00	0.00	0.00	0.00	1.00
MODEL2		1	2	3	4	5
Given error made in…	1	0.21	0.32	0.29	0.13	0.06
	2	0.00	0.25	0.28	0.30	0.16
	3	0.00	0.00	0.16	0.48	0.35
	4	0.00	0.00	0.00	0.64	0.36
	5	0.00	0.00	0.00	0.00	1.00
MODEL3		1	2	3	4	5
Given error made in…	1	0.15	0.29	0.33	0.19	0.04
	2	0.00	0.20	0.46	0.23	0.11
	3	0.00	0.00	0.53	0.33	0.15
	4	0.00	0.00	0.00	0.25	0.75
	5	0.00	0.00	0.00	0.00	1.00
MODEL4		1	2	3	4	5
Given error made in…	1	0.19	0.37	0.33	0.10	0.01
	2	0.00	0.24	0.37	0.30	0.10
	3	0.00	0.00	0.34	0.48	0.17
	4	0.00	0.00	0.00	0.50	0.50
	5	0.00	0.00	0.00	0.00	1.00
MODEL5		1	2	3	4	5
Given error made in…	1	0.22	0.26	0.30	0.18	0.04
	2	0.00	0.18	0.47	0.28	0.07
	3	0.00	0.00	0.20	0.57	0.23
	4	0.00	0.00	0.00	0.64	0.36
	5	0.00	0.00	0.00	0.00	1.00
MODEL6		1	2	3	4	5
Given error made in…	1	0.14	0.43	0.17	0.22	0.04
	2	0.00	0.20	0.37	0.29	0.14
	3	0.00	0.00	0.25	0.53	0.23
	4	0.00	0.00	0.00	0.67	0.33
	5	0.00	0.00	0.00	0.00	1.00

Table 5: Distribution of Recognition Location by Testing Model – Declining Error Study

		MODEL1	MODEL2	MODEL3	MODEL4	MODEL5	MODEL6
Stage of recognition	1	0.08	0.092	0.068	0.08	0.088	0.06
	2	0.256	0.24	0.204	0.26	0.268	0.28
	3	0.272	0.348	0.352	0.388	0.276	0.348
	4	0.256	0.228	0.28	0.204	0.276	0.236
	5	0.136	0.092	0.096	0.068	0.092	0.076

Expected Cost Analysis: We might use the probability information in a much more direct fashion, and determine the costs inherent within a particular environment. Suppose for instance, that we postulate the cost of creating each type of testing model as follows:

- MODEL1 - $2000
- MODEL2 - $2500
- MODEL3 - $2650
- MODEL4 - $3000
- MODEL5 - $6000
- MODEL6 - $12000

Further suppose that the costs of recognizing and rectifying an error increased significantly as development advanced, as suggested in [5]:

- An error recognized in the Business Requirements (BR) stage costs $100 to rectify
- An error recognized in the Systems Requirements (SR) stage costs $1000 to rectify
- An error recognized in the High-Level Design (HLD) stage costs $10000 to rectify
- An error recognized in the Detailed Design (DD) stage costs $100000 to rectify
- An error recognized in the Implementation (Imp) stage costs $1,000,000 to rectify

In such an environment, the long-term cost to rectify errors, when employing MODEL1, is:

= (the proportion of errors recognized in BR) *(cost to rectify in BR) +
(the proportion of errors recognized in SR) *(cost to rectify in SR) +
(the proportion of errors recognized in HLD) *(cost to rectify in HLD) +
(the proportion of errors recognized in DD) *(cost to rectify in DD) +
(the proportion of errors recognized in Imp) *(cost to rectify in Imp)
= (.144)*100 + (.268)*1000 + (.296)*10000 + (.212)*100000 + (.08)*1000000
= $104,442.40

Similarly, the error rectifying costs for the other five models are:

MODEL2:	$113,770.00
MODEL3:	$115,926.40
MODEL4:	$115,357.60
MODEL5:	$85,801.60
MODEL6:	$82,374.40

Finally, total costs (including tester costs) for each MODEL would be:

MODEL1: $104,442.40 + $2000 = $106,442.40
MODEL2: $113,770.00 + $2500 = $116,270.00
MODEL3: $115,926.40 + $2650 = $118,576.40
MODEL4: $115,357.60 + $3000 = $118,357.60
MODEL5: $85,801.60 + $6000 = $91,801.60
MODEL6: $82,374.40 + $12000 = $94,374.40

This suggests that, in the long-term, if we are operating under the framework of the first scenario, that the best testing structure is MODEL5. Although the team cost is higher, that is countered by the significantly lower expected error recognition costs.

It might be the case, however, that we do not have a very good understanding of the probabilities as presented in Table 1. In such a case, we might consider a graphical, sensitivity analysis approach. Consider Figure 7 below.

Noting that the expected costs for MODEL5 and MODEL 6 are comparable, we might wish to consider any uncertainty we have in the probability of finding an error at Implementation in MODEL5. That is, how would this uncertainty influence our decision-making process? Recall that the total expected cost under MODEL6 was $94374.40. As long as the probability of finding an error at Implementation is within 4% of the base value, our expected costs are less than MODEL6, and we would prefer to institute MODEL5. However, if our true value is potentially more than 4% of the base, then we would prefer to institute MODEL6.

We may perform a similar sort of sensitivity analysis, when there are uncertainties with respect to cost values. Suppose that there were uncertainty with respect to recognizing an error in the High-Level Design stage. How then does MODEL5 compare to MODEL6?

If we change the cost to recognize and correct an error in the High-Level Design stage, we see the results in Figure 8. Note that, as long as the decrease is no more than about 480% - a huge amount – MODEL5 remains preferable. Thus, there is little sensitivity with respect to this cost parameter.

Note that a similar sort of analysis could be executed, if cost information were available with respect to recognizing or remediating a problem in one phase, given it was created in some specific phase. This would explicitly make use of the probabilities in Tables 2, 4, and 6.

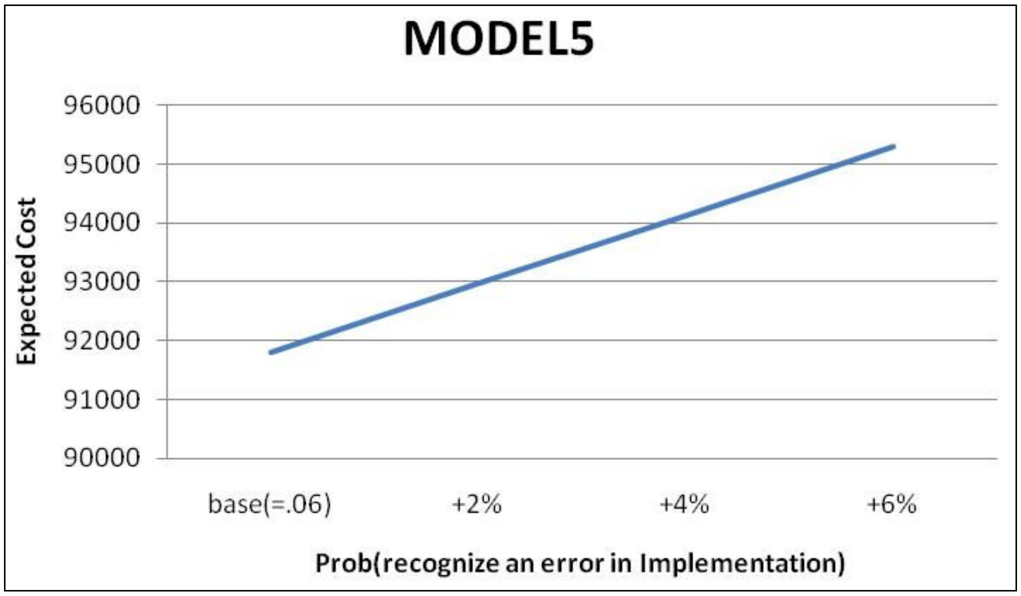

Figure 7: Cost Sensitivity

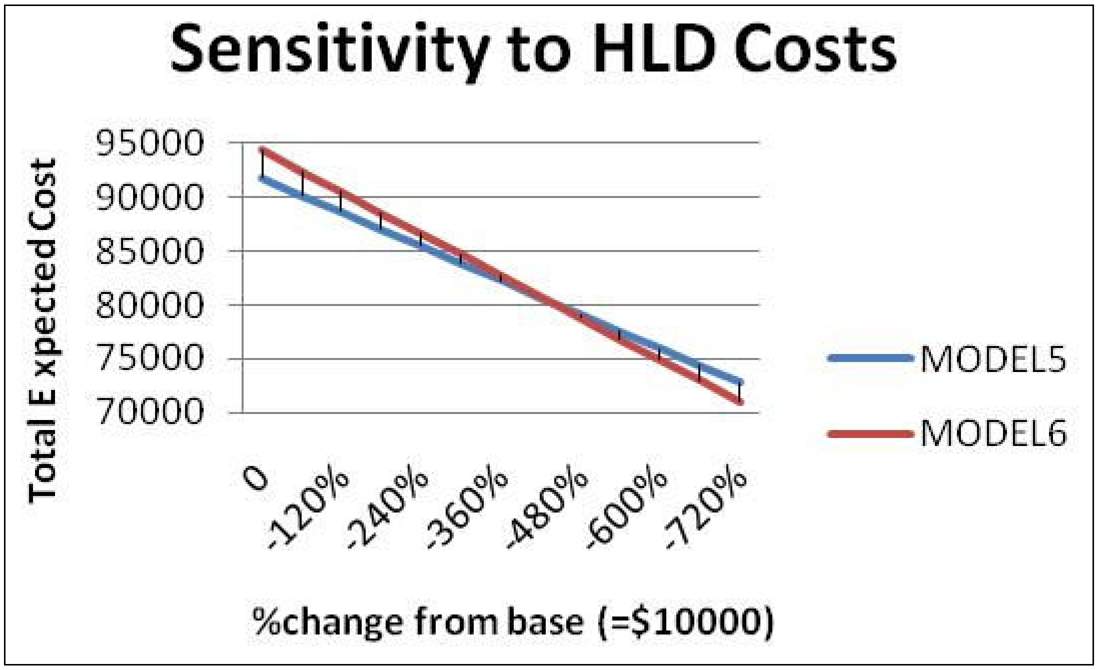

Figure 8: Sensitivity to HLD Costs

CONCLUSION

This paper proposed six models for engaging testers early and throughout the software development process. It also described the development and the analysis of simulation models of these six tester embedding models. Overall, the goal is higher quality application code with fewer errors produced, leading to higher quality applications being introduced into production. Furthermore, by engaging early and throughout the software development process, testers will see themselves as stakeholders in the quality of the finished applications by virtue of their work throughout the software development process. This will lead to the further development of software testing as a recognized and respected specialty within software development organizations.

REFERENCES

[1] Banks, J. "Introduction to Simulation," *Proceedings of the 1999 Winter Simulation Conference*, Phoenix, Arizona, December 5-8, 1999, pp. 7-13.

[2] Benediktsson, O., Dalcher, D., and Thorbergsson, H. "Comparison of Software Development Life Cycles: A Multiproject Experiment," *IEE Proceedings Software*, Volume 153, Number 3, 2006, pp. 87-101.

[3] Boehm, B. "Software and Its Impact: A Quantitative Assessment," *Datamation*, Volume 19, Number 5, 1973, pp. 48-59.

[4] Boehm, B. "A View of 20th and 21st Century Software Engineering," *Proceedings of the 28th International Conference on Software Engineering*, May 20-28, 2006, Shanghai, China, pp. 11-29.

[5] Boehm, B. and Basili, V.R. "Software Defect Reduction Top 10 List," *Computer*, Volume 34, Number 1, 2001, pp. 135-137.

[6] Carstensen, P.H., Sorensen, C., and Tuikka, T. "Let's Talk about Bugs!" *Scandinavian Journal of Information Systems*, Volume 7, Number 1, 1995, pp. 33-54.

[7] Cohen, C.F., Birkin, S.J., Garfield, M.J., and Webb, H.W. "Management conflict in software testing," *Communications of the ACM*, Volume 47, Number 1, 2004, pp. 76-81.

[8] Dahlbom, B. and Mathiassen, L., *Computers in Context – The Philosophy and Practice of Systems Design*, Blackwell Publishers, Cambridge, Massachusetts, 1993.

[9] Dalal, S.R., Horgan, J.R., and Kettering, J.R. "Reliable Software and Communications: Software Quality, Reliability and Safety," *Proceedings of the 15th International Conferences on Software Engineering*, Baltimore, Maryland, 1993, pp. 425-435.

[10] Faraj, S. and Sproull, L. "Coordinating Expertise in Software Development Teams," *Management Science*, Volume 46, Number 12, 2000, pp. 1554-1568.

[11] Fried, L. "Team Size and Productivity in Systems Development," *Journal of Information Systems Management*, Volume 8, Number 3, 1991, pp. 27-41.

[12] Fuerst, W.L., and Martin, M.P. "Effective Design and Use of Computer Decision Models," *MIS Quarterly*, Volume 8, Number 1, 1984, pp. 17-26.

[13] Gelperin, D. and Hetzel, B. "The Growth of Software Testing," *Communication of the ACM*, Volume 31, Number 6, 1988, pp. 687-695.

[14] Gorla, N. and Lam, Y.W. "Who Should Work with Whom? Building Effective Software Project Teams," *Communication of the ACM*, Volume 47, Number 6, 2004, pp. 79-82.

[15] Hamlet, R. "Special Section on Software Testing," *Communications of the ACM*, Volume 31, Number 6, 1988, pp. 662-667.

[16] Hetzel, B., *The Complete Guide to Software Testing*, QED Information Sciences, Wellesley, Massachusetts, 1983.

[17] Hotle, M. "Agile Development: What's Still Facts and What's Still Fiction?" *Gartner Research*, (ID: G00175807), June 14, 2010.

[18] Huang, A.H., "Model for Environmentally Sustainable Information Systems Development," *Journal of Computer Information Systems*, Volume 49, Number 4, 2009, pp. 114-121.

[19] Johnson, P.M. and Tjahjono, D. "Improving Software Quality Through Computer Supported Collaborative Review," *Proceedings of the Third European Conference on Computer-Supported Cooperative Work*, Milan, Italy, 1993, pp. 61-76.

[20] Kraut, R.E. and Streeter, L.A. "Coordination in Software Development," *Communications of the ACM*, Volume 38, Number 3, 1995, pp. 69-81.

[21] Laplante, P.A. and Neill, C.J. "The Demise of the Waterfall Model Is Imminent and Other Urban Myths," *ACM Queue*, Volume 1, Number 10, 2004, pp. 10-15.

[22] Light, M. "How the Waterfall Methodology Adapted and Whistled Past the Graveyard,"

Gartner Research, (ID: G00173423), December 18, 2009.

[23] McKeen, J.D. "Successful Development Strategies for Business Application Systems," *MIS Quarterly*, Volume 7, Number 3, 1983, pp. 47-56.

[24] Moore, W., Nolan, E., and Gillard, S. "Towards a Higher-Level Systems Development Life Cycle, with Universal Applications," *International Journal of Management*, Volume 23, Number 3, 2006, pp. 646-652.

[25] Myers, G.J., *The Art of Software Testing,* John Wiley and Sons, New York, New York, 1979.

[26] Pendharkar, P.C. and Rodger, J.A. "The Relationship between Software Development Team Size and Software Development Cost," *Communications of the ACM*, Volume 52, Number 1, 2009, pp. 140-145.

[27] Pettichord, B. "Design for Testability," *Presented at the Pacific Northwest Software Quality Conference,* Portland, Oregon, October 2002.

[28] Pressman, R.S. "*Making Software Engineering Happen – A Guide for Instituting the Technology*," Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[29] Pyhäjärvi, M. and Rautiainen, K. "Integrating testing and implementation into development," *Engineering Management Journal*, Volume 16, Number 1, 2004, pp. 33-39.

[30] Ragunath, P.K., Velmourougan, S., Davachelvan, P., Kayalvizhi, S., and Ravimohan, R. "Evolving a New Model (SDLC Model-2010) for Software Development Life Cycle (SDLC)," *International Journal of Computer Science Network Security,* Volume 10, Number 1, 2010, pp. 112-120.

[31] Royce, W.W. "Managing the Development of Large Software Systems," *Proceedings of the IEEE WESCON,* Los Angeles, California, August 1970, pp. 1-9.

[32] Schach, S.R. "Testing: Principles and practice," *ACM Computing Survey*, Volume 28, Number 1, 1996, pp. 277-279.

[33] Shannon, R.E. "Introduction to the art and science of simulation," *Proceedings of the 1998 Winter Simulation Conference*, Washington DC, December 13-16, 1998, pp. 7-14.

[34] Vijay, N. "Little Joe Model of Software Testing," *Proceedings of the 3rd Annual International Conference on Software Testing,* Bangalore, India, 2001, pp. 1-12.

[35] Waligora, S. and Coon, R. "Improving the Software Testing Process in NASA's Software Engineering Laboratory," NASA's Software Engineering Laboratory, 1996.

[36] Zhang, X., Hu, T., Dai, H., and Li, X. "Software Development Methodologies, Trends and Implications: A Testing Centric View," Information Technology Journal, Volume 9, Number 8, 2010, pp. 1747-1753.

[37] Zhu, H., Hall, P.A.V, and May, J.H.R. "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys,* Volume 29, Number 4, 1997, pp. 366-427.

AUTHOR BIOGRAPHIES

Mark L. Gillenson is Professor of Management Information Systems in the Fogelman College of Business and Economics of the University of Memphis. He received his B.S. degree from Rensselaer Polytechnic Institute and his M.S. and Ph.D. degrees in Computer and Information Science from the Ohio State University. Dr. Gillenson worked for the IBM Corp. for 15 years and has consulted for major corporations and government organizations. Dr. Gillenson's research has appeared in *MIS Quarterly*, *Communications of the ACM*, *Information & Management*, and other leading journals. His latest book is *Fundamentals of Database Management Systems*, 2005, John Wiley & Sons.

Michael J. Racer is Associate Professor of Marketing and Supply Chain Management in the Fogelman College of Business and Economics at the University of Memphis. He is the Associate Director of eSOL, the Enterprise Simulation and Optimization Laboratory, and the Associate Director for Supply Chain in the Center for Biofuels Energy and Sustainable Technologies. Dr. Racer has been a member of INFORMS since 1990, and is currently the VP-Special Projects for INFORM-ed. Dr. Racer's current research and outreach topics include the following: (1) modeling, simulation and optimization tools for decision-making in a wide range of applications, and (2) design of metaheuristics for combinatorial optimization problems.

Sandra M. Richardson is an Assistant Professor in the Department of Management Information Systems at the University of Memphis. Her research focuses on strategic leveraging of information technology to enable economic and social value creation and the role of individual stakeholders in this process. She conducts her research in two primary contexts: healthcare and the social sector (social entrepreneurism). Current projects focus on the impact of information technology on enabling emergent leadership, global collaboration, and autonomy for individual stakeholders. She has published in *Communications of the AIS*, *Decision Support Systems*, *Journal of Information Systems Education*, and *Information Systems Frontiers*.

Xihui Zhang is an Assistant Professor of Computer Information Systems in the College of Business at the University of North Alabama. He earned a Ph.D. in Business Administration with a concentration in Management Information Systems from the University of Memphis. His teaching and research interests include technical, behavioral, and managerial aspects of Information Systems. His research has appeared or will appear in *Journal of Strategic Information Systems*, *Journal of Database Management*, *e-Service Journal*, *Journal of Information Technology Management, Journal of Information Technology Education, Information Technology Journal*, *International Journal of Operations Research and Information Systems*, *Communications of the International Chinese Information Systems Association*, and *Pacific Asia Journal of the Association for Information Systems*.