# A Software Metrics Based Approach to Enterprise Software Beta Testing Design

by
Carlos Delano Buskey, B.S., M.S.

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies
in Computing

at

School of Computer Science and Information Systems

Pace University

April 2005

We hereby certify that this dissertation, submitted by Carlos Delano Buskey, satisfies the dissertation requirements for the degree of *Doctor of Professional Studies in Computing* and has been approved.

___________________________________________-________________

Dr. Lixin Tao Date

Chairperson of Dissertation Committee

___________________________________________-________________

Dr. Fred Grossman Date

Dissertation Committee Member

___________________________________________-________________

Dr. Charles Tappert Date

Dissertation Committee Member

___________________________________________-________________

Dr. Richard Bassett Date

Dissertation Committee Member

School of Computer Science and Information Systems
Pace University 2005

## Abstract

# A Software Metrics Based Approach to Enterprise Software Beta Testing Design

by
Carlos Delano Buskey

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies
in Computing

April 2005

Today, major manufacturers of enterprise software have shifted their focus to integrating the influence of the user community in the validation of software. This relationship is established through the corporate beta testing programs to confirm to the customer base that the software was properly reviewed prior to releasing to the target market. However, there are no industry standards for beta testing software and no structured software beta testing processes that may be used for the common enterprise application. In addition, beta testing models used today lack clear testing objectives. An additional problem in software beta test processes used by most manufacturers, is that there are no organized procedures for developing an effective beta test. Beta testing models strictly focus on error feedback, time of testing, and the number of beta testing sites participating in the program.

This research addresses the fundamental problems in poor beta testing design by contributing a software metrics based beta testing design model that uses weakness in the current product to plan an effective beta test. A set of formal beta testing objectives is initiated and essential software attributes are suggested to support each objective. Each objective is quantitatively measured with a set of metric functions used to predict risk levels in a software product. The predicted risk levels are used to assist in prioritizing tasks for the pending beta test. A learning process is provided to demonstrate the effects of the metric functions when the process matures.

The metric functions were tested on three real-world enterprise applications to validate the effectiveness of the formulas when employed on a live product. The experiment demonstrated close prediction to the actual risk value for each product and demonstrated how the predictions improved with experience.

## Acknowledgements

The process of writing a thesis was the most humbling experience I have encountered. I could not have successfully completed this without the love and support of so many individuals. I would first like to thank the three wonderful ladies in my life who complete me and consume me with love…Robin, Delani, and Ramsey, this could not have been done without your sacrifice and unconditional love. Robin, thank you for your kindred spirit. I would also like to give a special thank you to Dr. Lixin Tao, you believe in this project and you believed me and I am forever grateful for your support. Another special thank you to one my top supporters, Dr. Rick Bassett, who also influenced me greatly. Thank you to Dr. Fred Grossman for his motivation and support, and Dr. Chuck Tappert for instilling confidence in me to complete this project.

I would like to extend my gratitude to all of my family and close friends. Thank you Sheila, Brittny, Alice, Ella and Joe for supporting this project. To my Mom, who I have always strived so hard to make proud of me. A warm thank you to my special friends for all your support. To my friend Tyson, we have always discussed how important it is to keep "striving for perfection", thank you for fostering this goal with all your support. Alvin Lee Hill III, you told me to never be a quitter, thank you for seeing this before it was complete. Thank you, Corlette, all your positive feedback was motivating. To Jack Mooney and Laurie Sugg, thank you for encouraging me. Thanks to all of my other friends who made this accomplishment possible.

I would like to express a level of appreciation for all of my Pace University cohorts especially Shiryl, Steve, Mike, and Eddie for your support. This program brought us closer, in turn creating a strong bond. Shiryl and Eddie thanks for the long talks. Thank you, Christina Longo, for supporting this mission and your dedication to the entire program. I have matriculated at several universities in my academic career and I would like to state that Dean Susan Merritt is the best in class. Thank you for your dedication to this project, your dedication to the student body, and dedication to this program.

Most importantly, I would like to thank God for sending the people above to assist and support me in this effort.

Table of Contents

# List of Tables

## List of Figures

## List of Use Cases

## List of Use Algorithms

# Chapter 1

# Introduction

## 1.1. What is Beta Testing?

Beta testing is the first “user test” of software that provides enterprise software manufactures with end-user usability and software functionality feedback. Beta testing begins in the last phase of software development cycle prior to final release of the software. Software developers select special users (or user groups) to validate software and provide tools to collect feedback. The purpose of beta testing is to enhance the product prior to general availability (product release). Major software manufacturers are focusing on improving the quality, acceptance, and experience of enterprise software by promoting beta testing programs[12, 22, 30, 32, 40].

Most modern technology dictionaries, glossaries, and journals provide formal definitions of beta testing. The Microsoft Computer and Internet dictionary provides the most comprehensive definition for beta testing. Beta testing is defined as:

*“A test of software that is still under development accomplished by having people actually using the software. In a beta test, a software product is sent to selected potential customers and influential end users (known as beta sites), who test its functionality and report any operational or utilization errors (bugs) found. The beta test is usually one of the last steps a software developer takes before releasing the product to market; however, if the beta sites indicate that the software has operational difficulties or an extraordinary number of bugs, the developer may conduct more beta tests before the software is released to customers.” [34]*

Although, software manufacturers recognize beta testing as a formal and important element of software testing, and most industry dictionaries and glossaries provide formal definitions for beta testing, there are no existing standards or models for this form of software testing.

Manufacturers of software utilize multiple stages of testing to validate sectors of the application. Beta testing is the last stage of the software testing process regarded as a “live” test. During this stage of testing, end-users test the software to provide product usability feedback (negative or positive). Users suggest software features that improve the total end-user experience and provide valuable functionality data. As an example, manufacturers are interested in responses to the following questions: Does the software yield the same results when the end-user provides invalid data? Does the product generate a report when no data is entered? How does the application respond when the end-users extend the length of time between required responses in an application? Beta testing is the most effective process to generate this information.

Software manufacturers seek real-life users to test beta software. These users or testing groups are called beta testers. A pre-released version of the software is provided to the beta testers for a “specific period of time.” Beta testers install and test software in local environments creating software validation domains. Testers receive software documentation and product goals; however, beta testing is executed using black box techniques (with no direction or instructions from software development). Beta testers provide continuous feedback to software manufacturers throughout the beta testing cycle.

Beta testing is also a process that extends the validation and testing phase in the software development life cycle. An iterative software development life cycle includes a domain analysis phase, specification phase, design phase, coding and development phase, integration phase, validation and testing phase, and deployment phase (similar to the common waterfall model)[26]. The domain analysis phase examines business needs, product scope, and suggests a set of solution(s). The specification phase outlines the information and resources required to complete the product. The design phase drafts the software project providing a software blueprint. During the coding and development phase, system developers construct the application based on requirements received from the specification phase. The assembly of code and components emerge during the integration phase of software development. Software validation and testing is an important facet in application development seeking to validate the product by employing a series of tests. Lastly, the deployment phase releases the software to the end-user. Beta testing strengthens the validation and testing phase and fosters the deployment phase by assuring the product is fully tested.

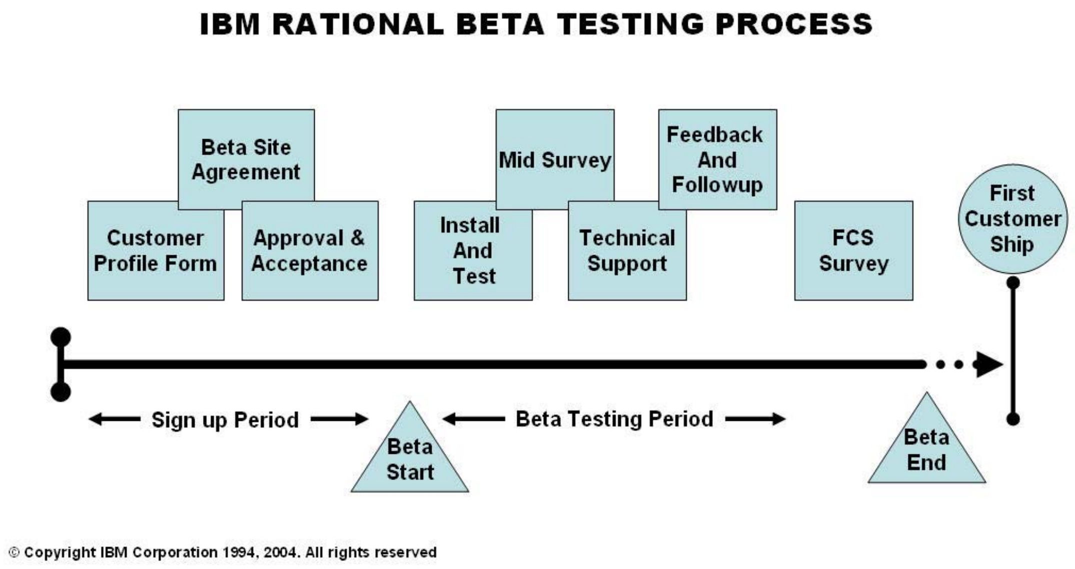

**Figure 1. IBM Rational Beta Testing Process**

Major enterprise software companies focus on processes to improve software through the beta testing process. IBM software advertises beta testing as a platform to ensure the product can withstand the challenges of real-world processing [22]. The company provides a formal beta testing process and the benefits of participating in its beta testing program. IBM beta testing program is divided into two sections: the sign up period and beta testing period. The sign up period, consisting of the customer profile, beta site agreements (non-disclosure agreement), and approval & acceptance phases, focuses on advertising, testing preparation, and testing user group selection. The beta testing period, which includes the install and testing, mid-survey collection, technical support, and feedback and followup stage, executes the process until the software developers end the beta testing stage. After completion of this stage, IBM releases the product for shipment to customer base (See Figure 1. IBM Rational Beta Testing Process[22]).

Computer Associates Corporation (CA) beta testing program recruits suitable users to validate software prior to general availability [12]. CA has two types of beta testing programs, open and selective beta programs. The open beta program is for end-users who do or do not have experience with the product in beta. The benefit is increased participation in new product lines and/or exposure to software with a small user group (i.e. users seeking to test new products or using a new product for proof of concept purposes). Selective beta testing programs restricted to users with former product experience. CA extracts selective beta testers from product advisory councils and “development buddy” programs. Both programs provide a pre-released version of the software, in beta, to end-users with a general testing focus. Beta testers participate in periodic meetings with the manufacturer to provide product feedback, which is addition to the normal support channels (e.g. online support tools, knowledge base, product documentation, etc). The benefits are customers in both programs, impact the quality of the software and CA improves the value and position in the software market.

Formerly, Microsoft relied heavily on its channel of industry certified professionals and user groups to beta test software. This group of professionals comprised of Microsoft Certified Professionals (MCP), Microsoft Certified System Developers

(MCSD), and Microsoft Solution Providers, all part of its TechNet Network[31]. As a part of this network, Microsoft provided a monthly subscription of software, which included the latest beta software programs. Microsoft has now shifted its focus, opening its beta testing program to any user. Microsoft has created TechNet Virtual labs for testers to participate in beta testing. TechNet labs employ remote technology to prepare beta testing environments (coined sandbox development labs) dynamically, which provide additional time for the beta tester to test the software [32]. Microsoft believes this will increase participation in the beta testing program, providing faster end-user response time, and a volume of product feedback.

Another major software distributor, Oracle, solicits developers to provide information for its OTN (Oracle Technologies Network) profile database. Oracle engineers utilize the personal data to match possible test sites for beta testing cycles [40]. This method establishes beta tester relationships and matches test products with the most effective group of end-users.

In addition, there are websites devoted to beta testers and companies that locate testers, manage beta test, and provide a forum to discuss beta testing issues. BetaNews.com provides a one stop location for companies to post a pre-released version of new software and a web-based forum to provide feedback in one location[4]. Large corporations, such as Microsoft, have products posted to the website to recruit beta testers.

### *1.1.1. Beta Testing Objectives*

Software testing occurs in multiple stages utilizing various techniques. The method of software testing employed is directly related to the outcomes desired. Software testing occurs from the introduction of the first set of code until the completion of the product in test. This study focuses on the final stage of testing for enterprise software, which is beta testing. This section identifies and describes the objectives important to manufactures when beta testing enterprise applications. The objectives outlined in this section provide a foundation for the software metrics based model for beta testing design introduced in this study.

The function of beta testing is to measure usability and the functionality. Manufacturers of software develop a set of objectives for an application prior to testing. The objectives are designed by application specifics and outcome expectations. In this thesis, a set of important beta testing objectives is provided for any enterprise software. The objectives are environmental dependency, function coverage completeness, localization, robustness, user interface usability, user interface accessibility, and system vulnerability. This study is limited to the aforementioned set of objectives; however, future studies may expand the number of objectives.

#### 1.1.1.1 Environmental-Dependency

Environmental dependency is an application's reliance on additional components to function correctly in a production environment. Enterprise software manufacturers generate products for large corporations with robust infrastructures. Robust infrastructures house applications, which has adapted to its installed environment (e.g. shared files, access policies, DASD volume control, etc). Historically, enterprise software manufacturers have encountered environmental issues with new applications when deployed on robust infrastructures (especially legacy systems). For example, software manufacturers of operating systems produce a new version or major maintenance release (e.g. service pack, or maintenance build). The memory address or shared library versions change with the upgrade of an application. These changes affect how applications use this information. The result is abnormal system behavior or the existing application ceases to function. During a beta testing cycle, software manufacturers seek to identify and measure environmental dependency issues during deployment of the software.

#### 1.1.1.2 Function Coverage Completeness

During the design phase of application development, software developers decide which functionality is included to meet the scope of the project. Additionally, developers predict need, based on information received during the specification phase. However, the expectation changes when customers use the application. Function coverage completeness is an objective of beta testing, which measures end-user function expectations in enterprise software. The goal is to focus on customer feedback (Does the product meet the end-user expectations?). This feedback promotes minor changes in the software. Additionally, the feedback received from end-users impacts future releases of the software.

### 1.1.1.3 Localization Specification

Environmental-dependency addresses deployment issues in software. However, enterprise software companies with a global presence are focused on localization expectations, which is an additional deployment concern. Localization is the adjustments in enterprise software to prepare the product for foreign infrastructures. Localization specification is a beta testing objective, which measures whether software meets foreign standards. As an example, a database application sorting tables must adjust to the required language. Characters in the English language differ from Japanese language. Beta testing abroad also collects diverse usability feedback.

### 1.1.1.4 Robustness

Software manufacturers utilize multiple forms of testing when designing software. These tests are based on the adoption of testing best practice. However, in-house software testing is conducted by utilizing a “white-box” testing method. White-box tests are based on a finite number of use-cases, empirical evidence, and historical data (e.g. issue logs, event logs, etc.). This form of testing is predicting outcomes based on control issues. However, what happens when the end-user enters erroneous data in a field? How is the software affected if the end-user shuts down the software when data is being processed? Does the software function after unpredictable conditions?

Robustness is an objective that measures how software responds to user habits. This objective measures the complexity of the application and utilizes the data to make changes. Robustness also measures the common mistakes made by end-users, which is used to enhance usability in the software.

#### 1.1.1.5 User Interface Accessibility

Software manufacturers are required to provide modified interface for people with disabilities (e.g. sight limitations, hearing, physical impairments, etc.)
The Rehabilitation Act of 1998, requires electronic technology to be accessible and easily modified for people with disabilities[43]. Accessibility is features in software that provides ease of use for persons with physical disability. The user interface accessibility (UI accessibility) objective measures the efficiency of UI accessibility in enterprise software. Beta testing is the most efficient method of testing to measure this subjective goal and promote change prior to releasing the product to the customer.

#### 1.1.1.6 User Interface Usability

A major goal of beta testing is to measure the usability of software. Usability is the ease in which a user adapts to a program. User interface usability (UI usability) collects end-user feedback on the graphical user interface. The feedback is a requirement often misinterpreted in the specification phase of the software development life cycle. In addition, feedback is based on client expectations or features the end-user believes will ease product usage. This objective is important because feedback from beta testing impacts the current product and adds value to future releases of the product in test.

#### 1.1.1.7. Software Vulnerabilities

Beta testing focuses on measuring usage patterns and functionality of the product. As environmental dependency measures the software ability to adapt to current system configurations, system vulnerability validation is important. Software vulnerability is testing an application for potential security violations. This measures firewall standards and violations of trusted domains. A firewall is a program (or set of programs) used to manage network traffic, monitor, and protect information. Firewall standards are the network policy implemented to protect information. As end users introduce new products to a system, the application behavior changes according to access rights. Beta testing measures the application behavior and validates that firewall standards are not compromised by the new application (or changes).

Trusted domains are network file systems or applications with users' policies implemented. Trusted domains are another layer of security restricting access to system information. Software manufacturers are concerned about how applications perform when information is restricted. Developers of enterprise software simulate application behavior according to standard network configurations, but networks are customized according to corporations' best practices. Beta testing is a proficient form of testing to measure this objective.

### *1.1.2. Beta Testing Design*

The focus of this study is to provide a software metrics-based approach to enterprise software beta testing design. The function of this methodology is to provide a formal model for testing mature software in the final stages of the coding phase, the software development life cycle. The mature code development stage is the period when a version of software is near-finished coding and close to being in final preparation for the release to the end-user (or ready for the deployment phase). A fundamental problem in beta testing is the lack of industry standard or testing design to test enterprise software. The International Organization for Standardization (ISO), which governs non-proprietary technology standards. ISO awards an ISO 9000 certification to companies that properly institute measures to improve the quality of software[9]. ISO expects corporations to institute testing methodologies with limited guidance. Beta testing is a goal for the top software manufacturers; however, testing design models are ad-hoc. Most beta testing design models focus on time parameters and data collection, with limited attention on usability.

This study introduces a beta testing design framework for enterprise software. The beta testing design components for this model are: identifying the proper type of testing group (skill set requirements), size of the testing user group, questionnaire’s design, and predicting beta testing duration (testing period). The components utilized function metrics to predict risk levels for objectives. The outcome provides a focused approach to the beta testing phase and demonstrates the value of this form of software testing.

In the preliminary stage of beta testing, Beta test managers meet with the product developers to discuss the testing (project) scope. The scope provides a summary of the steps required for the desired outcome of the test. The steps are based on a set of testing objectives. Product developers review testing scopes and provide final approval. The beta testing scope is ad-hoc and not an important component of this beta testing design model.

When the scope is complete, beta test managers preselect beta testers or start to solicit for a pending project (i.e. when new software requires testing or the product market is small). Product testers form a testing user group that is created according test objectives, product functionality, and end-user product experience (if applicable). Product experience factors are based on the user group’s level of industry experience. This is not important when new products are in a beta testing phase.

After forming the beta testing user group, beta test managers prepare testing questionnaires. The questionnaire provides an outline to guide beta testers in expected outcomes of the beta test and to solicit feedback. In addition, questionnaires guide testers in prioritizing tasks, but maintaining the black box approach to testing the product. Black box testing is a form of testing where users are not concerned with the mechanics or interested in the code but focus on functionality. The main focus of the testing questionnaire is to guide testers and provide an instrument to collect data.

Following approval of the questionnaire, beta test managers construct a testing time line to predict the amount of time required for a complete beta testing cycle. Timelines are not formal but essential in managing the testing length. Testing time lines are estimated based on historical data or testing methodologies. In addition, beta test managers construct testing timelines based on trends. As the feedback proliferates from testers in the field, testing timelines are adjusted (extending or decreasing testing time).

In Controlling Software Projects: Management, Measurement and Estimation, Tom DeMarco stated,"…you cannot control what you cannot measure in controlling software development." This growing adage outlines the foundation of this study[14].

### *1.1.3. Current Status of Beta Testing*

Beta testing is an essential component of the software validation and testing phase because of its immediate impact on the product in test. It provides a platform to integrate real-world feedback into a product, prior to availability. This collectively improves the software usability and functionality, which lowers cost implication and improves quality[16]. The focus of beta testing is to measure usability; however, deployment of the software is important. Users report issues to developers if software fails to function correctly or documentation does not resolve issues. The impact of beta testing on the software industry is important and global standards are required to manage this process.

During the implementation of a new product, large infrastructures are subject to environmental issues during deployment; this is not common in small environments. For example: A large insurance company with over 9,000 users experienced a memory leak in a new operating system, causing a 30-minute system outage. A senior information technology executive for this company estimated these issues resulted in 1 million dollars in lost time. The manufacturer of the operating system sent a team of five-developers onsite to find the problem and eliminate it. If this client were part of the beta testing program, these issues would have been exposed and eliminated during the testing phase. This created cost issues for the client and both cost and quality issues for the manufacturer. This is just one example of the importance of beta testing.

Beta testing is widely used in the software industry. The top software manufacturers dedicate a section of its development process to beta testing. In addition, software manufacturers provide literature (most in an html format) to educate and solicit testers for current and future releases of software. Major software manufacturers such as CA, Microsoft, IBM, Oracle are actively beta testing enterprise software ranging from operating systems, database applications, and enterprise management tools[18]. In addition to software manufacturers, e-commerce businesses and web resources solicit users to beta test features and applications[1]. Google.com uses beta testers to validate new search engine features to capture usability and functionality feedback[18].

IBM has posted its beta testing model “IBM Rational Beta Testing Process,” which provides a beta testing road-map (see Figure 1). Microsoft Corporation has revamped its beta testing program to allow any qualified user to participate in the beta testing process. In addition, it monitors usability and functionality through its “Customer Experience Program.” Computer Associates has two types of beta testing programs, open and selective, which allows experienced and non-experienced users to test programs. Computer Associates has also formed “development buddy” and “product advisory” councils to build partnerships with experienced end-users. These experienced user groups are often beta testers. These programs are discussed in length in section 1.1.

As corporations continue to model beta testing programs, there lie a series of fundamental problems. Modern beta testing processes focus heavily on error and deployment issues and time constraints. Beta testing processes also lack focus on collecting usability data, properly distilling data collected, dedication to functionality, and proper recruitment of qualified beta testers. Beta testing is an ad-hoc form of testing focused more on time of testing parameters. This study provides a focused approach to software testing using software metrics. The metrics are based on a formal set of beta testing objectives and design components.

## 1.2. Beta Testing versus Other Forms of Mature Code Development Stage Testing

Post-development testing emerges, as the product is near completion and prepared for the deployment phase (preparing the product for release to end-user). In this stage, the code is consider mature and may be tested by the end-user. There are several forms of testing employed during this phase to validate the product is functioning as designed [See Figure.2]. This section provides a contrast between beta testing and other forms of post-development testing. The forms of post-development testing discussed in this section are alpha testing, acceptance testing, integration testing, stress testing, smoke testing, system testing, installation testing, and white box testing.

### *1.2.1. Alpha Testing*

Alpha testing is a controlled software test that is tightly managed by the internal software development team[16]. Alpha testing is performed on internal systems typically in a controlled environment. Other forms of alpha testing provide special testing labs for customers with strict testing requirements. The developer monitors the tester's progress and validates that the product is functioning correctly. This phase of testing eliminates deployment and workload issues.

Alpha testing improves the quality of the product by discovering errors and exposing common usability issues prior to final release of the product.

Alpha testing is unlike beta testing because testing is conducted by end-users in real-life environments. In addition, developers do not control the test; they support end-users for technical or customer support issues. In addition, beta testing does not require special labs and the major focus is not deployment and workload issues. Receiving deployment data is only one objective of beta testing.

### *1.2.2. Acceptance Testing*

Acceptance testing is a level of testing utilized for customized software and applications. Customized applications are software packages typically designed for internal business sectors or applications designed for a special group of end-users. Customized applications characteristically have a definite stakeholder who owns the project and product[41]. During this stage of testing, the end-user teams with the developer to validate software requirements specified in the statement of work. Stakeholders sign off on this process after users agree the software is functioning as designed. Acceptance testing is the final approval process in customized applications.

When testing enterprise software, acceptance testing is not a robust form of post development testing. Enterprise software is designed for a wide range of customers, providing general features, and focused objectives. Additionally, enterprise software does not have an external product stakeholder. The stakeholders are the target market. Beta testing is more effective when testing on a larger scale creating a longitudinal validation of the software.

### *1.2.3. Integration Testing*

Integration testing employs a pattern approach to software validation. Integration testing is akin to extreme programming (XP) testing techniques exploiting agile experimental methods[41]. XP programming uses a simple methodology that implement smaller deliverables when building and testing code[3]. This testing certifies the program structure of the software by testing the application interface. Software developers test sectors of the product to eliminate interface issues. The goal is to accomplish this early and often. This form of testing starts when the software code matures, and continues until the developers release the product to the next phase.

Integration testing implements a diverse method of UI validation and differs from beta testing. In addition, integration testing focuses solely on the software effects when additional codes are introduced, which differs from the goal of beta testing. Integration testing lacks the dynamics required to validate enterprise software packages. Beta testing employs a holistic view of testing with a definite set of objectives, time constraints, and outcomes.

### *1.2.4. Installation Testing*

Installation testing examines software to confirm compatibility with hardware platforms[27]. During this phase of testing, developers install the product on various hardware and operating environments (if applicable) and validate portability (if required). Portability is the ability of software to function on different operation systems. Installation testing is not a complete cycle of testing and is narrow in focus. Installation testing differs from beta testing because it has a single objective and does not require real-world testers for completion.

### *1.2.5. White Box Testing*

White Box testing is a method of testing that uses a strict procedural design to build test cases[41]. This form of testing employs test-user groups to experiment with every module of the application. The procedures are determined by predictions constructed during the coding phase or from historical data. In addition, the logical design is tested including data structures. Developers train the testers, on the application, and monitor the entire testing phase to record results. During white box testing, the goals are not transparent to the user and outcomes are definite. The data and results from white box testing are construed and not effective in a quality post-development testing process. Beta testing is a black box form of testing and performed by end-users, providing unbiased feedback.

### *1.2.6. Black Box Testing*

Black box testing is a form of software validation that reviews circumstances in which the program does not behave according to its specifications[36]. Black box testing reviews the functional characteristics of an application to reveal the presence of issues associated with this goal. This form of testing does not focus on the software code but features of the application.

### *1.2.7. Stress Testing*

Stress testing is performed only by the software developers, which encompasses a set of executables, used to simulate or stimulate abnormal behavior in a program[36]. The purpose of stress testing is to consider situations that normally shutdown, cause an abnormal end (ABEND), or produce irregular conditions in a program. Testing specific constructs in a program uncovers instability in software. Stress testing starts after the software coding is complete and continues until benchmark results are satisfied. Stress testing, alone, validates the stability of the application, but does not provide data for post-development objectives (e.g. usability and functionality data).

Beta testing is not concerned with benchmark data. The data received from stress testing is narrow in focus and does not provide the depth required to validate software.

### *1.2.8. Smoke Testing*

Smoke testing is a testing technique used to validate software after the introduction of a new code or maintenance builds (akin to validation testing)[2].

This form of testing confirms that new updates did not compromise the integrity of the product. Smoke testing is also narrow in focus, since it only provides compatibility and deployment data. Although beta testing reveals potential compatibility issues in software, smoke testing has a narrow focus and a limit amount use cases. Beta testing is more variable incorporating multiple objectives.

### *1.2.9. System Testing*

System testing is a series of independent sub tests that simulate an actual computer system[41]. The subcomponents of system testing are performance testing, recovery testing, security testing, and stress testing. Characteristically, developers are responsible for executing this phase of testing. Unlike beta testing, system testing combines a “pre-packaged” testing model to validate software and is restricted to internal testers (developers or quality assurance groups). System testing is unlike beta testing because tests are performed internally and feedback is assumed.

### *1.2.10. Usability Testing*

Usability testing is a form of software validation tests where the graphical user interfaces and measures ease of use in an application. Usability measures the client-functional characteristic of the application such as user accuracy, user response, user information recall, and end-user input accuracy[50]. Usability testing alone does not complete the goal of software testing and does not cover all use cases associated with complete testing.

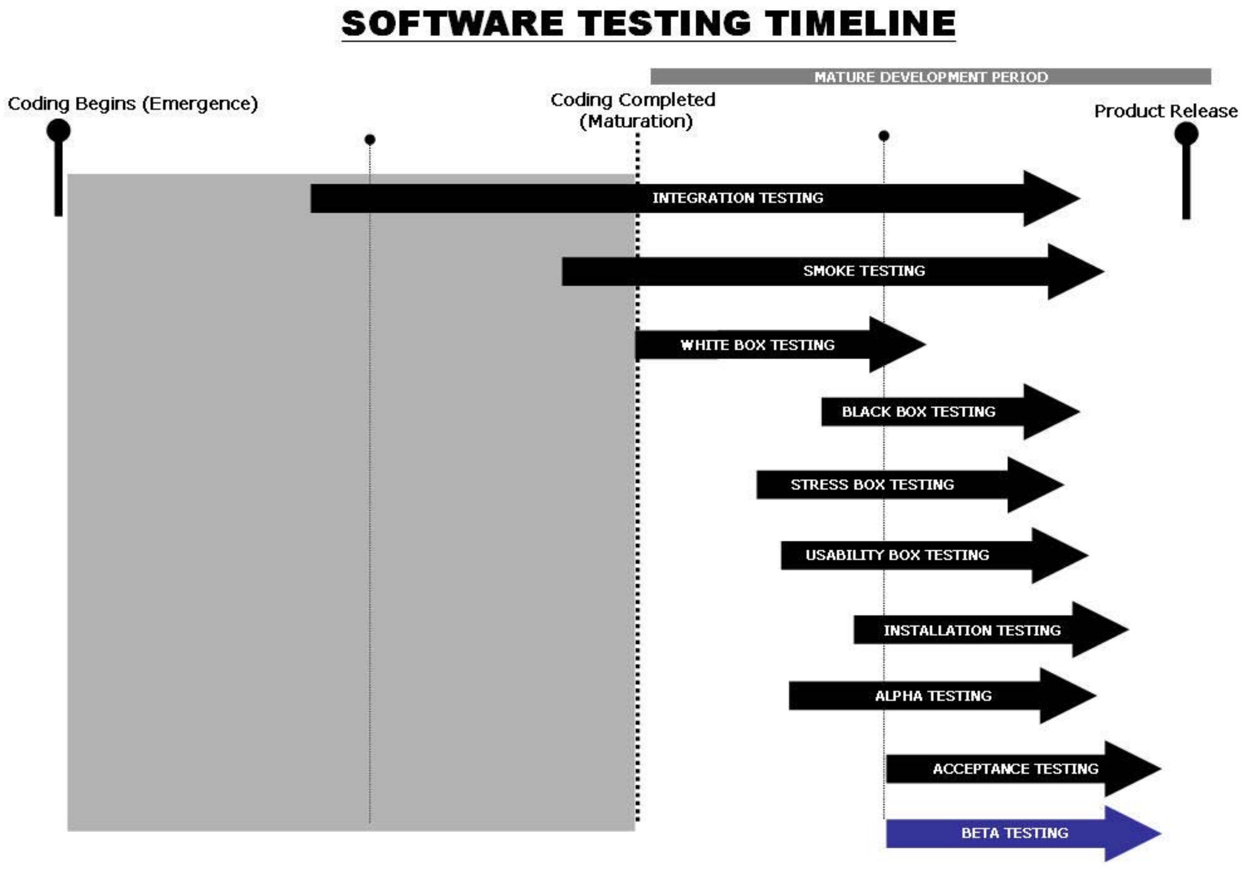

**Figure 2. Software Testing Time Line**

## 1.3. Solution Methodologies

This study provides a framework for beta testing design specifically for enterprise software. The process uses a software metrics based methodology to guide manufacturers in building a schema for beta testing a product. There are three key areas in this study; formally identifying key beta testing objectives and design components for beta testing, present a set of software metrics functions to predict the importance of diverse beta testing objectives, and contribute a beta testing design methodology based on the results of the metric functions. This formal beta testing framework provides a methodology for common enterprise software, which is absent in software engineering.

### 1.3.1. *Identification of Beta Testing Objectives and Attributes*

This study identified a formal set of beta testing objectives and significant attributes of the common enterprise application. The objectives and attributes are key areas of focus when beta testing a product in a real-world environment. There are seven objectives (goals) for beta testing the common enterprise application; each has a unique focus and outcome. For each objective, there is a set of essential attributes of an application that must be validated to support outcome. Each objective has a metric function used to reveal the potential existence of issues. This process is important because it provides clear testing objectives and provides a distinctive base to measure improvements.

### 1.3.2. *Introduction of Software Metrics Functions*

This study also provides a set of software metrics functions used to reveal potential weakness in the software product categorized by objectives. Understanding the potential risk of problems prior to testing a product in production offers the software developers more leverage when preparing the product for beta testing. In addition, the risk levels provided by the software metrics assist software developers in prioritizing testing task. The software metrics provide a predicted value from 1 – 5 with each number representing a risk level. With continuous use, the software metrics improve, providing better risk level predictions.

### *1.3.3. Introduction of Beta Testing Design Methodology*

The software metrics are used to support the beta testing design methodology contribution provided by study. There are 5 unique steps in the design of beta test; identifying beta testing priority clientele groups, identifying priority functions/GUI components for beta testing, designing the questionnaire for controlling beta testing priorities, and deciding the minimum beta testing clientele size prediction. The beta testing design creates a custom beta testing process for enterprise level software. This methodology is important because there is no industry level beta testing design standards that may be adopted by any common enterprise software.

## 1.4. Major Contributions

The major contributions of this study are:

- provide a distinctive set of beta testing objectives that may be used for common enterprise software
- provide an essential set of software attributes that predict risk levels in the testing objectives
- create a set of software-based metric functions when used with the software attributes, predict the potential risk of issues in a production environment
- design a software function learning process that improves the accuracy of the metric predictions through experience

- provide a formal beta testing design methodology using two real enterprise products to demonstrate its influence

## 1.5. Dissertation Roadmap

This dissertation contains six chapters. Chapter 1 introduces the concept of enterprise beta testing and highlights the different areas of testing, introduces the solution methodology, and highlights the major contributions. Chapter 2 provides an overview of the various forms of testing and their impact on software development, discusses the importance of testing, highlights the limitations of testing, and discusses the current beta testing practices and their limitations. Chapter 3, provides a detail description of the software objectives, software attributes, software based metrics and the application of the metrics. Chapter 4 thoroughly defines the beta testing design methodology and provides an example of its usage. Chapter 5 provides an overview of the software metric based functions and an experiment of the functions on two enterprise products. The chapter demonstrates how the metric functions predict risk value and how the functions are trained to produce better predictions with experience. Chapter 6 provides a conclusion of the research and offers suggestions of the future of this study.

# Chapter 2

# Software Testing and Beta Testing

## 2.1 Importance and Limitations of Software Testing

Software testing is an improvement process that goes back more than four decades. Software testing is the process of executing a software system to determine whether it matches its specification and executes in its intended environment[51]. The major focus of software testing is to find errors which support the ideology of E. Dijksra who believes software testing is used to show the presences of defects but never the absence of bugs[39]. In addition, software testing is only as good as its process. Today, manufacturers of enterprise software use testing to influence the quality of an application. There are several forms of testing used to discover errors in the software. However, software testing can only discover bugs in the product it cannot eliminate errors.

Software testing is an exercise in product improvement seeking to refine the way applications are evaluated. It is an important factor in the software development life cycle seeking to assess the operation of the application in turn locating errors in the process and code. Software may also fail by not satisfying environmental constraints that fall outside the specification[51]. In addition, testing provides a perception of quality and demonstrates an activity was initiated to improve the product.

There are many software process improvements such as CMM and IS0 9000; all seeking to manage and improve the way software is developed and tested[9, 42]. Today, software testing remains complex. The focus is not product specific, but geared more towards an unproven process. Manufacturers of application are dedicated to process improvement but not product improvement. In this research, the focus is on the actual products and how their attributes are used to influence the way the applications should be tested.

Testing still remains a conundrum because software developers have difficulty addressing some of the common problems in software testing such as:

- No true methodology to decide when to start and end testing.
- Lack of set of compelling events used to dictate when a product is thoroughly tested and testing should end.
- No guarantee errors will be revealed during initial testing.
- Does not certify an improvement in exceptions handling a product
- Cannot validate every possible use case
- Some testing requires a special skill set to properly test.
- Unreliable outcomes for specific objectives.

## 2.2 Major Types of Mature Code Development Testing and Their Roles in a Software Process

The coding phase of the software development life cycle is the stage in the software building process where software engineers build the software using languages, objects, etc. This section highlights the major form of testing used to validate mature software code. Mature software code is the stage when the software is close to completion. The software testing methods discussed in this section are compared to beta testing later in this chapter to demonstrate their limitations. There are several forms of testing used to validate and support software quality control. The major types of testing employed after the code has matured are integration testing, white box testing, black box testing, usability testing, stress testing, smoke testing, installation testing, alpha testing, acceptance testing, and beta testing.

Integration testing is a step approach to validating software by testing major builds of the application. This form of testing secures modules of an application, tests each unit, and applies it to the program to verify it is functional. This process creates a systematic approach to application design[41]. The process is used to eliminate errors in small units and build an application incrementally.

Integration starts very early in the coding phase of software development, use an agile type methodology. Since integration testing is employed as modules are coded, the process could start early in the coding phase and continue until completion of the project. This process is most effective with usages as capturing fatal errors in improper calling sequences or mismatched parameters early during the coding phase[19]. Integration testing validates the performance and reliability of an application most notably included in the design specification of an application. There are several usage cases solved by this form of testing.

Integration testing uses a skeleton technique to application validation. It is employed to assist with resolving functional issues with its step type methodology. However, there are a few notable limitations outlined in section 2.2.1.

White box testing is a validation process that reviews the structure and code of the program to build test scenarios. The test cases are clear and based on information revealed in program routines. White box testing executes every statement in the program at least once[20]. To guarantee the independent paths in a program have been executed successfully, the software development team performs white box testing[41]. In addition, the data structures are tested to assure validity[41]. The overall goal is to reveal any logical issues in the code. This form of testing is performed later in the software coding process after the product is matured. The software should be near completion so a complete test case may be exercised.

White box testing is an important process because it exercises testing from a holistic point of view. The application code is reviewed to build a test case based on the design of the application. Bill Hetzel, in The Complete Guide to Software

Testing, Second Edition, stated, “we don't even notice such features unless we look in the wall or inside the box to see what has been done”[19]. This form of testing studies the software code to understand how to test it.

Black box testing uncovers potential issues in the functional areas of software. These functional areas include (but not are not limited to) data class boundaries, missing functions, data volume limitations, interface errors, initialization errors, data structures anomalies, and performance errors just to name a few parameters[41]. The unique characteristic of black box testing is that code validation is not the goal but confirmation of the information domain. Black box assures the product features are operating as designed and the software reveals the presence or absence of errors.

Black box testing is a test case procedure exercised in the later stages of the software development process. In several cases, black box testing may proceed (or complement) white box testing to test data rates, the effects of use cases, responsive to input values, and other functional specifications.

Black box testing is used after a series of tests have been performed on the code and to test circumstances in which the program does not behave according to its specifications[36]. For example, black box testing is to test that the application extracted the data from the correct repository and generated results based on a specified set of rules. Black box testing may also reveal common initialization errors. The outcome of this testing resolves a series of problems associated with application design.

Black box testing is most useful at validating test cases created by development. For instance, if the developers of an application wish to understand how the application handles input errors entered by an end-user, black box testing is the method employed to address this concern in an internal setting.

Another form of testing is usability testing. The focus of usability testing is measuring how the end-user interacts with a software product. Usability testing takes into perspective the end-user's views to validate the quality of design. This form of testing uses a small set of the targeted user group to perform a test of features (GUI) to confirm ease of use.

Usability tests are performed in a controlled environment monitored by the program developers. Usability tasks take into account user accuracy, user response, user information recall, and end-user input accuracy[50]. The data is used to refine the design factors of the application. Usability testing is employed on the code close to the end of the coding stage.

Usability testing is performed after the code has matured, later in the software development life cycle. In Glynn Myers, The Art of Software Testing, he believes usability testing is most useful to validate[36]:

1. The program provides ease of use.

2. The software user interface has been tailored to the intelligence, educational background, and environmental pressures of the end user.
3. The program output provides meaningful data.
4. The program produces straightforward error messages in the event of a system exceptions
5. The program provides input redundancy, not containing a large volume of unrestricted text fields
6. The program lacks an excessive number of options, or options that are unlikely to be used.
7. The program user interface is systematic and demonstrate unity
8. The program returns an immediate acknowledgment to all inputs.

This provides a comprehensive list of the test cases for usability testing.

The goals of usability testing mirrors Myers list of test cases by validating that end-users complete specific tasks successfully. Usability testing also reviews the steps the user implements to resolve a task and if the steps are optimal. This form of testing also reviews end-user issues, complications, and operational efficiency[50].

Stress testing is a performance validation process that subjects a program to any excessive conditions[36]. The goal of this form of testing is to introduce a set of executables to demonstrate abnormal behavior in a program. Stress testing validates constructs in a program to uncover the existence of instability in code modules.

Stress testing is used to introduce excessive activity with the specific purpose to “break the program”[41]. The test validates whether the program performs at peak performance over a period of time[27]. For example, in a mathematical program a subset of data may be introduced to a series of algorithms. System engineers will review the system for degradation, accuracy of output, and other performance metrics.

Stress testing is introduced later in the coding phase of the SDLC when benchmarking is required for a specific feature or the program as a whole. The software developers using a routine or separate benchmarking application perform stress testing. The purpose of stress testing is to consider situations that normally shutdown or cause an ABEND in a program. Stress testing produces the existence of errors in sensitivity to large volumes of data, memory leaks, output rates, disk resident issues, and virtual memory problems[41].

Stress testing uses a series of independent sub-tasks to simulate a software system and verify the performance of the application is complete. The overall goal is to exercise the complete computer-based system to validate errors do not exist in the interface of application with hardware (or existence applications)[41].

Stress testing takes into account the target computer system and the entire domain[51].

Stress testing is a massive validation exercise used to eliminate the existence of errors in the total computer system. Stress testing is best used to confirm system requirements' specifications, all functions are exercised, procedures and interfacing systems are executed, and invalid system input is accepted or rejected[19]. Stress testing requires a large amount of preparation prior to execution. The coordination efforts are time consuming and must be based on good design process.

Stress testing is performed in the late stages of the coding phase proceeding the integration testing[23]. When executed correctly, stress testing demonstrates the performance and influence the quality of the application by[19]:

- Testing the performance of the application
- Benchmark performance by introducing use cases that push the system to its limits
- Analyze specification carefully (especially in scenarios that reveal errors)
- Test source data and/or use simulators to generator application use cases
- Successfully evaluate the effectiveness of performance parameters such as response time, run requirements, and file sizes.

These outcomes as a whole provide a summary of cases solved by this type of testing.

When updates and maintenance are produced for an existing application, smoke testing is issued to produce the existence of bugs.  Smoke testing adds value to software when changes or errors are identified in a particular module.  Also, smoke testing is employed to execute a series of mini-tasks to validate the software is test-ready for more extensive testing.

Smoke testing is a smaller version of integration testing used to assure features in an application still exist after introducing the new code[2].  For example, if an end-user recommends a functional change to an application that is required for its environment (customization change).  Smoke testing is used to verify the application does not affect the remainder of the application.  In this scenario, only a few small changes were made to a module and a mini test is run to assure a new code does not impede production.

Smoke testing complements integration often creating a little overlap.  The testing technique is often employed early in the coding phase in the SDLC after a new module is added.  It is also exercised in the later stages after new changes are recommended during the alpha testing, acceptance testing, usability testing, and beta testing.  Smoke testing solves the fundamental problem in software testing that ensures that the same base level of functionality is still present after the latest check-ins[2].

Installation testing is a validation method used to reveal the existence of errors in the install process. Installation testing validates compatibility with other platforms. Installation has several test cases, which are[27]:

- Ensures all programs are interconnected correctly
- The system options selected by the end-users are compatible as a unit
- The hardware is configured correctly
- All installation files have created the correct content
- And all parts of the system exist.

Combined, the process assures the application functions on target platforms.

Installation testing is employed to validate the options, user data files, and libraries are loaded correctly[36]. The software must also properly interconnect with the current system and not create any integration issues. Installation testing is employed later in the SDLC after the code has matured. Since enterprise software is designed to operate on a variety of hardware/operating environments, this form of testing is most effective in validating these test cases. Installation testing is regarded as a hardware and operating system acceptance test[27]. The installation testing reveals compatibility issues introduced by the software in test. The test case also locates errors in the installation process[36].

Alpha testing is a controlled test where test-subjects team with the developers to test the product in a homogenous environment. The developer views the testers of the software to make certain the product is functioning correctly. Alpha testing adopts the same techniques used in black box testing because test subjects are not concerned with the code but the functionality of the application.

Alpha testing is employed during the later stages of the coding phase after the software is matured and near completion.  Alpha testing is conducted before beta testing with partner developers, dedicated client sites, and internal business units.  For example, a software developer of a service desk application (help desk software) would install the latest version internally and allow service desk engineers to use the product to eliminate functionality issues in a test environment.

The outcome of alpha testing produces potential code or design changes [2]. Although, alpha testing does not capture the essence of end-user inference, it eliminates design and functional errors prior to a longitudinal test (beta test).

Acceptance testing is a level of testing that is used when customized software is being developed for group or unit and validation is required by the end-users[41]. The software is final test before releasing the product to the business unit. During this stage of testing the product may be tested for a short period and final acceptance is conducted where the stakeholders signoff on completion of the product.

Acceptance testing is most effective when business units required a final approval of the development process. In this scenario, end-users verify the scope of the application matches their expectations. The acceptance test is the final stage of testing for small unit based applications. The main focus of acceptance is to demonstrate that the software is ready for operational use[19]. The test use case validates user requirements, ensures the user is involved, validates user inference, and completes quality testing.

### *2.2.1 Limitations of Major Types of Mature Code Development Testing*

This study focuses on testing that validates the software in an actual production setting. Beta testing is the most robust form of software testing providing the explicit feedback from the target user group. Most of the applications mentioned in Section 2.2 serve a specific purpose in the testing process; however, their limitations fall short in total quality management.

Although black box testing is used to validate function issues in software, it does not take into account actual use cases or all possible use cases. This is impossible without the influence of external systems. In addition, black box testing may not reveal any errors. Another limitation of black box testing is that it is closely managed by product developer unlike beta testing which is executed by the beta testers. Software developers on support the testing process (e.g. technical support, customer support, etc.)

The limitation with employing integration testing is the level of skilled engineers required to implement this process appropriately. The engineers must have experience with the software discipline and this type of testing use case. An additional limitation of integration testing is the total time it takes to implement this approach, since each module requires testing prior to introducing it to the software project. Additionally, integration testing is meticulous in nature and must be tested thoroughly. The testing technique does not take into account real-world use cases at this point.

The importance of usability testing is important to measuring the users view of quality in the application[5]. However, usability testing has it limitations because it is a very task driven process focusing more on specific tasks or instructions than full use cases. Usability testing is more scenario-based than reviewing multiple use cases. In addition, usability testing is performed in a controlled environment.

Stress testing is a performance validation procedure with a narrow set of test use cases. This form of testing is limited to a finite set of software programs not useful for batch processing applications or compiler applications[36]. Also, stress testing uses a simulated stress load but does not provide the large number of use cases, which is virtually impossible. Beta testing is a more robust testing practice because it provides the real world use cases not demonstrated in an internal testing environment.

Another limitation revealed in most forms of code development testing are the limitations outlined in system testing. System testing is an engineering system that requires a number of testing activities. Its limitation is based on effective time utilization because it introduces overlap in some test cases, which creates redundancy. This form of testing is conducted on internal systems and requires knowledge of the product and produces the ideal target environment.

Also having a unique set of traits is the beta test. Beta testing uses a base of end-users (and projected end-users) to validate code prior to final release. It is also robust in scope. Unlike smoke testing, it is very limited in scope and only validates a specific test case. Smoke testing is performed in an isolated environment and by skilled software engineers.

Also limited in scope is installation testing. The largest problem with installation testing is performed by software developers on internal systems. This form of testing is not as robust as beta testing because it does not take into account other environmental issues such as existing applications, limitations of the hardware, security, etc. The test case is performed by the software development group, which is most familiar with the expected operations and characteristics of the application[27].

The limitation associated with white box testing is most notably that it is restricted to internal environments and must be performed by the software developers. Since white box testing typically analyze the data flows and controls in the software code to build a test scenario, software engineers with experience are required[37]. This is an activity with clear objectives and must be performed internally. White box testing would not be performed in the real production environment because of the risk associated with the unproven code. This makes white box testing less robust than beta testing.

Acceptance testing is not robust enough to provide efficiency for enterprise level software. In addition, acceptance only provides feedback for a small base of users and not capable of providing the robust level of feedback required for applications designed to support a larger user base. The major limitation of alpha testing is it is conducted in a controlled environment. Alpha test is influenced by the presence of software developers [41].

The limitations in the major types of testing outlined in this section (with the exception of beta testing) are:

- each provide a narrow scope in software improvement
- some lack a clear set of objectives creating overlap and redundancy
- none take into account actual production data (mimicking production settings)

- a large amount are performed on internal systems which provides a false perception of completeness
- several lack a true set of design steps
- performed on internal systems only
- require skilled software developers or systems engineers
- and practically all create some level of overlap which impedes progress and wastes valuable time.

Beta testing is a more comprehensive form of testing because the product is tested on the target platform. This form of testing increases the confidence level of the end-user and the direct feedback validates user inference.

## 2.3 Current Beta Testing Practices and Their Limitations

Today, there are no beta testing standards or industry best practices. However, most major software manufacturers have beta testing programs devoted to raising the awareness to its user community that products are adequately beta tested. The beta testing practices used by most companies are unique to the specific company only focusing on test duration, the ratio of testers, existence of errors, and refining of a process.

Companies today advertise openly beta test durations to alert the user base that the product has been tested over the course of a certain period[22]. The beta test ends after the testing period if there are not outstanding issues and the number of participants' requirements was met. The limitation in this methodology is it does not validate the product has been tested effectively. In addition, this method does not validate the weakness in the application was tested thoroughly.

Manufacturers also focus on recruiting a set amount of beta testers, based on the ratio of current end-users. There is typically a set amount of users required to test products based on some predefined principle. There has also been a shift in using testing recruiting services to obtain the desired amount of testers for a product in test. This method is weak because the number of users does not automatically indicate the product will be tested effectively. In addition, this method does not account for skill and quality of testers.

Often the key focus for beta testing is to eliminate bugs in the application. Software testers' work closely with beta testers to validate the code does not create errors. However, there is little guidance in the areas that potentially generate problems when implemented in production.

Beta test plans today are more focused on providing a uniform template for beta testing products. This activity is driven by process improvement activities. Beta testing should be unique by product or product domain and influenced by past data.

The beta testing design provided in this study strengthens limitations in modern beta testing plans by proposing a software-based method testing design. This process demonstrates the importance metrics is in the improvement of software testing.

# Chapter 3

# Software Metrics for Enterprise Software Beta Testing

## 3.1 Major Features of Common Enterprise Software

This section outlines seven core features of an enterprise application. The core features are the focus of the beta testing metrics provided in this study. However, it is important to understand the structure of an enterprise application. The term “enterprise,” in computing, is a large scalable application designed to support robust and large transactions in a business environment. Enterprise level applications are deployed on large networks successfully integrating with other applications, operating platforms, and networking components. Software manufacturers develop enterprise software to support a vast user base yet scalable enough to adapt to diverse environments. Today, general-purpose enterprise solutions provide out-of-box functionality with software customization options for specific organizations. However, it is common for software manufacturers to design proprietary software for a specific company or business vertical. The enterprise-level software solutions are developed for corporations, government agencies, hospitals, universities, legal firms, non-profit organizations, and other large institutions.

Designed to function on large user base infrastructures, enterprise applications support a single business focus, providing solutions to a number of business problems (multiple use cases) in a specific focus area. An example is enterprise management applications (EM software), which are  designed to support infrastructure software management having the capacity to perform automatic software delivery, manage maintenance updates, provide enterprise hardware and software inventory and reporting (to name just a few features).  In this scenario, software management is the single business focus area, and to solve multiple business requirements, software management applications  manage support issues for a large user base, update systems simultaneously, controls inventory, and provide reporting (if required).

The common enterprise software product has various features based on its explicit design.  To meet the demand and expectations of the targeted production environment, enterprise software (at a minimum) is scalable, network enabled, customizable, provides multithreading, out-of-box functionality, fault tolerant, and integrated security.  In addition, enterprise software is capable of handling external and large transaction requests and integrating successfully with existing production software.  Combined, these attributes permit the product to function effectively in an enterprise environment.

Chapter 1 provided information on the major enterprise software corporations (Microsoft, IBM, Computer Associates, Oracle etc.) and the individual corporate direction for beta testing software.  These companies develop and manufacture enterprise applications ranging from network management solutions to security applications.  The most widely accepted enterprise application is software management solutions, which provide help desk functionality, software delivery options, enterprise license agreement management, asset management, knowledge database, and reporting features. The help desk (or service desk) feature of a software management application provides features (components) that allow end-users (administrators or non-administrators) the ability to manage problem logs. The software delivery option provides a resource to manage large system implementations and a utility to push applications and/or maintenance updates to end-users on a network. Large corporations, with robust networks, encounter issues managing application licenses or software authorization codes. Software management applications provide license management features offering organized license schemas. The asset management component collect information from enterprise hardware/software, stores the information in a centralized database, and in turn offers administrators a utility to manage valuable company resources. Another feature of software management applications is the knowledge base solution, which provides users with a database to research solutions to issues. Reporting utilizes information in the database for trending, updates, and ad-hoc reports. This example of a software management application provides a basis for understanding of the core features in a typical enterprise application.

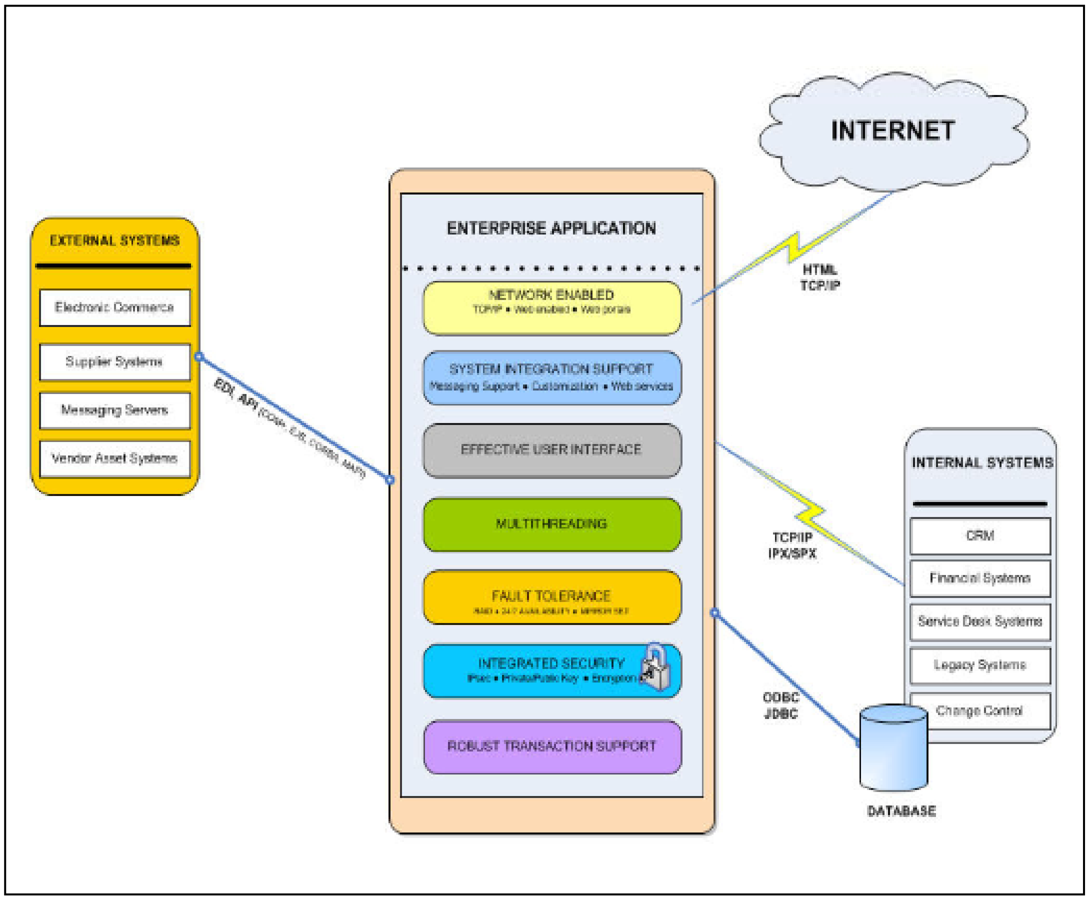

Figure 3. Core Features of an Enterprise Application

An enterprise application's core features are simplified administration, web enabled components, support multithreading, contain an effective graphical user interface (GUI), support robust transactions, include fault tolerance components, and obtain integrated security. Each of these common features operates as a unit to support the capacity of large and demanding infrastructures (See Figure 3. Core Features of an Enterprise Application). The core features are valuable in studying beta testing by revealing which components or modules' attributes must be properly tested.

### *3.1.1 System Integration Support*

Enterprise application integration is one of the major challenges businesses face when introducing new applications, software components, into production (i.e. an enterprise infrastructure). Integration is not just a clean installation of the application; it includes the deployment of the application into production without impeding the current infrastructure. Many modern enterprise applications are following a new demand from corporations to design applications with out-of-the-box functionality for a specific organization (proprietary). In either case, software manufacturers design applications to be customizable and communicate with other internal and external applications.

During the beta testing phase, system integration is tested to insure the product is functioning in the environment. Software testers validate the application is operating and has not compromised other applications. Testers also validate communication is functioning between internal and external systems.

#### 3.1.1.1 Internal System Integration

Inter-company systems are internal applications or a collection of software that support a company's daily operation. Inter-company systems integration is the software's ability to function with internal systems and not impede current operations. Inter-company systems includes (but are not limited to) CRM (customer relationship management) systems, internal financial systems, service desk applications, legacy systems, change control systems and databases consisting of all the major units supporting the line of business.

### 3.1.1.2 External System Integration

Extra-company systems are shared systems outside of a corporation but critical to success of the business. Newly implemented applications are required to provide support for external systems and not corrupt existing platforms. External systems are key to maintaining business-to business (B2B) and business-to-consumer (B2C) relationships, external client support, and other business functions. Examples of external systems include supplier systems, messaging servers, vendor assets systems, etc. It is common for enterprise applications to support EDI (electronic data interchange), which is a standard in e-document exchange.

Although not mandatory, some manufacturers design enterprise applications to utilize web services for external system integration. Web services are utilized in business applications to support point-to-point communication via the Internet. In addition, software manufacturers design enterprise software with web-enabled components to help the end-users and expand business relationships. Specifically, corporations have business relationships established with vendors, suppliers, partner firms, and clients utilizing special infrastructures to communicate with external systems.

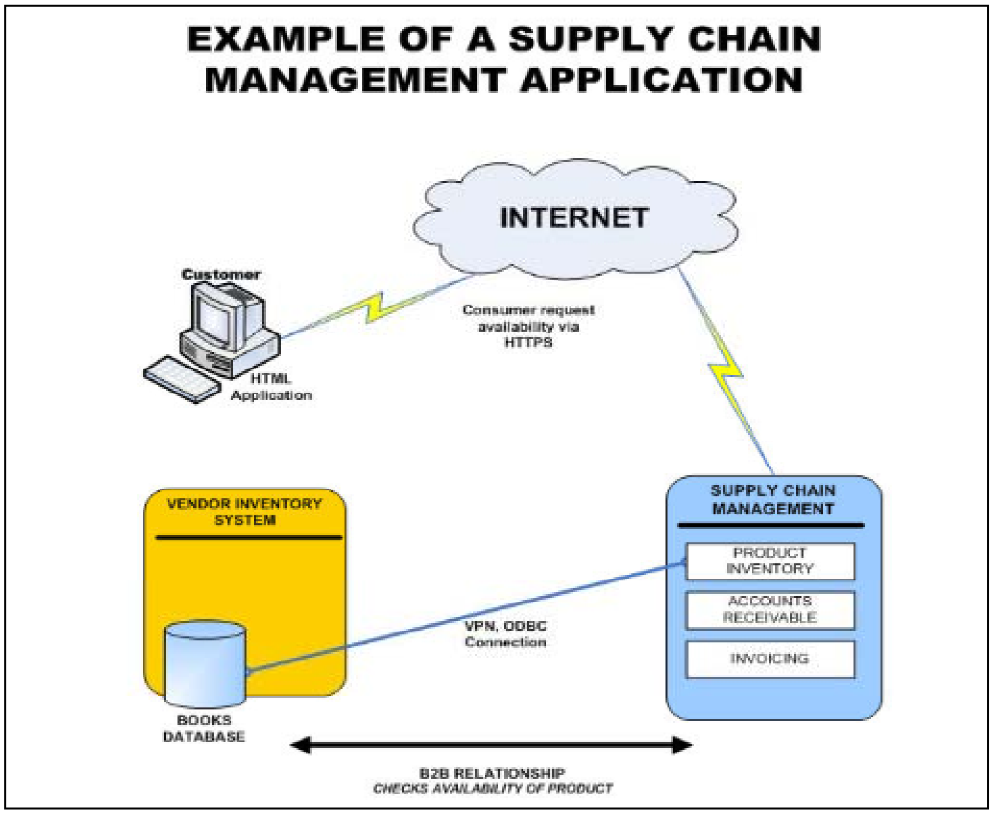

**Figure 4. Core B2B Application**

Corporations use secured web connections to support B2B and B2C systems. B2B are trusted relationships established with two or more companies to conduct business using a specific system or group of systems (See Figure 4. Core B2B Application). B2C is the same relationship; the difference is the relationship is established between business and customer (e.g. ecommerce, technical support, etc).

Corporations utilize secure socket layers (SSL) and/or virtual private networks (VPN), to establish secured connections with a business to carry out a transaction. A classic example of this practice is supply chain management applications (SCM) like those used for online book ordering. Supply chain applications check the availability of books directly with the manufacturer. This practice streamlines the inventory process and fosters customer service.

#### 3.1.1.3 Message-based System Integration

Enterprise software must conform to its targeted infrastructure by including the proper communication features required to support other production applications. Message-based integration assures the product communicates, when necessary, to other software components and platforms. This is done by requesting a service or using a standard application-programming interface (API). An API is employed to make a call to an application to facilitate communication. A common messaging API in WINTEL and IBM environments is NetBIOS. Open Database Connectivity ODBC is a standard open API, which facilitates communication with databases. APIs are characterized as asynchronous or synchronous, which rely on the source and expectation of the call. The difference between synchronous and asynchronous APIs is how they communicate with the host application. Synchronous APIs provide simultaneous communication with the host application. Asynchronous APIs' communication is performed with a specified start and stop point. Often, enterprise software manufacturers create unique messaging systems or rely on middle-ware applications to support communication (relatively).

#### 3.1.1.4 Customizable Features

When implementing applications in a production environment, regularly enterprise-level corporations are interested in applications that provide functionality out-of-the-box. Out-of-box functionality is required to assist in streamlining the integration of the product reducing the need for expert integrators or special implementation services. However, there are instances that require the application to be tailored to perform a specific task. Customization is a feature of enterprise application that provides the ability to perform unique adjustments to foster flexibility in software. This is exercised by creating or updating scripts or rules generated to meet a requirement not offered in the products in its native format. Customization may be required if an application does not inherently support an external function or system condition.

An example of customization is adjusting web-enabled applications to support anti-ad pop up features in web browsers. System support applications (e.g. helpdesk software or service desk applications) require web deployment to support the user base of large institutions. Common web interfaces incorporate a pop-up login screen for authentication purposes. In the case of financial institutions, which use integrated software to suppress web pop-ups, the application requires customization to recreate a log-in screen that does not pop-up for logging into the application.

Enterprise applications are capable of conforming to any environment including minor customization features that do not ruin the integrity of the application.

Customized components are not tested during the beta testing phase. However, customizable features included in the product specification are tested during the beta testing phase to assure the product is not compromised after changes are implemented (i.e. web portal applications, exceptions configured to limit when a back-up will be executed, etc).

### *3.1.2 Network Enabled*

A vital part of application communication is its ability to support networking environments. Large corporations have established network infrastructures linking nodes to servers, mainframes, mid-range, and storage devices. Enterprise infrastructures have multilayer hardware firewalls, routers, and hubs used to establish cross-platform communication while protecting the company's information. Enterprise software provides support for network communication and adheres to security policies.

Enterprise applications require robust networking modules, which provide the ability to communicate via standard communication interfaces. Communication is facilitated utilizing network protocols such as TCP/IP, IPX/SPX, LDAP, X.500 etc. Enterprise applications must also communicate with an entity's firewall (whether it is hardware/software based, or both). Firewalls protect data externally and internally by managing access. Enterprise network administrators grant access by specific ports. If an application does not have rights to the correct ports, errors occur or the application may not function correctly. This is a difficult feature to design and normally requires customization to adhere to firewall policies. However, some enterprise applications may provide software wizards to ease firewall administration.

The same wizard features are utilized to assist administrators with configuring user rights and group policies. Commonly, enterprise applications provide a base set of user rights to restrict application access. In general, three-levels are included:

- Administrators - full access to the application (useful when trouble-shooting and customizing the software).
- Power-User - advance application ability but normally lower than administrators (provides rights to advance application components).
- and Users - limited in scope providing basic application usage with some self-service administration function (i.e. changing account password).

Applications use the same footprint to assist administrators in setting up group policies. All are important in easing the administration of enterprise applications.

#### 3.1.2.1 Web-enabled Components

It is common to find some enterprise applications designed to utilize web protocols to communicate using the Internet as a global communication backbone. Other web-enabled applications support HTML, WML, SGML, SOAP, and a number of web services. Supporting these services, features, and protocols impact the deployment, support, and availability of enterprise applications.

Applications available via the web are a common expectation in enterprise applications reducing hardware and administration overhead. Hardware cost is reduced because web-enabled applications create thin clients utilizing HTTP to access applications via the Internet (e.g. Internet Explorer, Netscape etc.). Thin clients are computer systems with no physical implementation (e.g. dumb terminals, Citrix clients, etc.). Employing thin clients also reduces administration cost by creating centralized applications environments lowering the cost of additional hardware for each end-user, reducing licensing cost, and creating a single point of failure for troubleshooting purposes (among many other administration requirements).

### *3.1.3 Effective Graphical User Interfaces (GUI)*

Enterprise applications are designed to ease the end-users learning curve by incorporating a comprehensive graphical user interface (GUI). The GUI is often well-organized, designed clearly, and properly instruct the end-user upon execution. GUI's utilize a well-defined command button, text boxes, drop down menus, directional arrows, and other options. Since, the GUI function as the "face" of an application, consumers' expectations are high and software developers spend quality time developing this feature.

The GUI should include similar appearance whether it is the native, java-based, or web-based interface. In addition, enterprise software developers are expected to provide some level of customization, which enhances the user experience by providing familiarity.

### *3.1.4 Multithreading*

As an application advances, it must be capable of starting other tasks while the operating system is managing other requests. Multithreading is a feature in applications which spawns independent execution paths giving applications the ability to perform tasks simultaneously. This feature increases performance and accelerates processing.

Multithreading is essential in supporting the graphical user interface (GUI), integral in servicing multiple clients simultaneously, performing calculations, and other tasks initiated or requested by the application. The GUI is used to initiate a thread by the end-user (i.e. to print a document). If multithreading were allowed, the user would have to wait until the thread is finished before initiating a new task. This feature also allows multiple clients to function (based on the resources available).

Today, most operating systems support multitasking enhancing the applications ability to utilize the processor more efficiently. All operating platforms support 32bit processing from Microsoft Server to IBM z/OS.

### *3.1.5 Robust Transaction Support*

Enterprise applications are mission critical software in large corporations and must operate successfully. When a transaction is required, it must operate accurately and efficiently each time. Robust transaction support is the ability of the application to function well when demand is high, and still provide the correct warning when erroneous data is entered or correct the error. Enterprise applications must be capable of executing transactions in large volumes seamlessly. In addition, be capable of differentiating between user inputs and provide warning if an error occurs (instead of providing a valid incorrect answer). Examples of robust transactions are those facilitated by financial applications. Software manufacturers design financial transactions to facilitate check handling and the distribution of funds to other systems. If a financial application requests electronic submission of funds to another system, it is critical that the transaction is performed successfully. This system requires this action to occur at a high rate (approximately hundreds of transactions per minute). During the transfer of information to specific accounts, if there are discrepancies in a field (where information is identical) the system should provide a warning not to continue with the process.

### *3.1.6 Fault Tolerance capability*

Failover and recovery is an important feature of enterprise software. In meeting the market demand, enterprise corporations expect software to provide continuous availability (especially for those entities that have a global presence). Enterprise software includes features that provide seamless recovery in the event of disaster. If the application fails abruptly, it must have the ability to recover quickly, limiting the outage time and loss of data. This concept is referred to as high availability. The application has 24/7 availability features that constantly store transactions such as failover, which is a second version of the application. When the first fails this second version is started and information is switched.

In the case of corporations that utilize enterprise-level service desk applications, an outage would be detrimental to corporations. For example, most energy companies employ service desk applications to log power issues from clients (i.e. reporting a power outage, or natural disaster). In the event of a storm or other natural disaster, the volume of calls from customers increases. If the service desk fails, it must be able to recover quickly to support the volume of calls.

### *3.1.7 Integrated Security Policies*

Enterprise software must incorporate a standard layer of security. Security layers users and groups access levels, data encryption, and data integrity. User access level includes the application ability to manage specific end-user (and groups) rights to various components and access to data. Data encryption protects software information from unauthorized use. Data integrity assures unauthorized sources do not modify or change data. The integrity of the application demands that all of these security components work collectively to protect the application.

User and group policies (highlighted in section 3.1.2 Network Enabled), assure the proper access is provided to the end user base and manages user accessibility. In a networking environment, administrators establish groups of users based on some corporate specific access policy or methodology. To foster current network policies, enterprise applications provide a base set of similar integrated features to assist in managing function access and data availability. This section is an important security feature and creates an additional security layer.

Enterprise software includes an integrated security feature designed to prevent unauthorized access data generated and utilized. To support this requirement, enterprise software employs an encryption methodology such as private and public key encryption to secure data. In the public key scenario, there are two keys, a public and private key. The public key encrypts the data and the private key decrypts the data.

Data integrity is also a type of integrated security feature used to assure that unauthorized users or resources do not modify application data (e.g. external hackers, Trojan horse programs, virus, or other forms of corruption). This includes data scanning mechanism, encryption, and authentication to protect the integrity of the data. Security logs are created to police unauthorized access to information. Enterprise applications rely on APIs to establish relationships with software firewall applications and virus scanning engines to protect data entering and exiting an application.

## 3.2 Beta Testing Objectives

Chapter 1 introduced a core set of beta testing objectives for enterprise software. The core objectives measure the major attributes of the common enterprise application (outlined in Section 3.1). In addition, beta testing objectives form the foundation for the testing design model introduced in this research. The focus in this section is to provide a detailed description of the beta testing objectives using real use cases to demonstrate the importance. Future studies will increase the number of objectives to enhance the beta testing design.

### *3.2.1 Measuring Environmental-Dependency Issues in Enterprise Software*

Enterprise software manufacturers design applications to utilize current resources to maximize efficiency in its targeted environment. Each requirement may create a level of dependency, which is related to resources required for operation such as network services, physical processors, access to data etc. If a dependency issue occurs, it will generally be discovered during deployment of the application into a production environment. It is important to test the product for deployment issues prior to introducing it to the target environment.

For example, legacy systems customarily employ mission critical applications, which support a large and diverse group of users. Legacy systems' dependency on the enclosing environment and other attributes create a level of precaution during implementation [35]. Understanding this level of precaution, most enterprise corporations have special implementation models used to test applications for dependency issues prior to releasing to the production environment.

In this study, environmental dependency is defined as the application’s reliance on hardware, components, and other applications to function correctly. The objective is to identify issues that may impede operation (production). Environmental dependency is a “use" association, which forms within software systems [8]. In addition, software may create dependencies based on special customizations. System or application reengineering impacts software dependencies, which is formed by poor planning or maintenance. All of these activities influence environmental dependencies in software, creating a direct effect on deployability.

Legacy systems house applications with limited changes over an extended length of time. During the application’s operating tenure, the software has developed hardware and software dependencies. To illustrate this point, job management applications rely on resources to manage the volume of programs competing for resources. Programs competing for system resources are subject to a limited window to utilize the mainframe to execute a job, or program. Most programs request datasets from external locations such as volume libraries, DASD, or some database resources. In this case, programs are expecting the operating application to manage resource utilization and access to information.

Changes to the operating application or location of the dataset will create dependency during operation.

Case 1. Mainframe dependency of older OLE to function with host applications

*As Wintel applications are developed to utilize legacy software dependencies are formed to facilitate communication with heritage applications. Distributed job management applications are a growing trend bridging the utilization of desktop operating systems to manage production control in mainframe environments. In this case, a job management application developed prior to the year 2002 utilized OLE as a standard for communication. Dependencies were formed because applications were configured to utilize the OLE framework. However, since 2002 most developers utilize newer communication protocols (such as COM or DCOM) to support job management applications moving away from the OLE framework. Enterprise corporations deploying newer job management applications would encounter dependency issues with newer features because OS kernel would request services from older APIs. This issue can be reduced with proper beta testing.*

Customization in applications and networks creates environmental dependencies. Often, enterprise applications require system customizations to adjust to environment (infrastructure) changes or system dependencies. Customizations are the direct result of reengineering projects, which promote system evolution in corporations. Legacy systems often spawn applets that are developed by engineers to adapt to rapid system growth. In Case 1, distributed job management applications are important to manage mainframe workflows, assist in building and managing critical path job monitoring, creating reports, and utilizing the standard task features offered by desktop application (such as drag-and-drop functionality, multi-tasking, desktop portals, etc.). In this case, vendors have created new client/server job management applications to manage the flow of programs executing on a mainframe. However, the new application was developed with codes that utilize new system communication components (e.g. Microsoft COM). In this scenario, the legacy application requires common communication upgrade to maintain dialogue with the distributed application to function correctly.

Enterprise corporations are most concerned with the effects that new software and maintenance will have on its current production environment. Software developers are interested in whether software can adapt to an infrastructure without creating irreparable effects. Beta testing is an effective testing product to eliminate the occurrence of errors that may be associated with environmental dependency issues. Proper beta testing assists in encapsulating and isolating various forms of application dependencies such as:

- Operating system dependencies (Platform dependencies)
- Resource dependencies (Shared library, databases, network application)
- Hardware dependencies

- Protocol dependencies
- Security/Access dependencies

During a beta test, implementation is expected to occur in a “real" production environment or model environment. If implementation is successful, beta test managers provide a series of actions to measure environmental dependencies. If the dependencies have caused an impediment in system production, the feedback will influence the current application. Developers will add the updates either via a maintenance release or include it in a future release of the application in test.

Operating system dependencies are applications needed for OS components, files, or services to function. The kernel is the core module of operating systems, which create dependencies in applications and hardware. The operating system kernel functions as a translator between software and hardware components. Applications depend on the kernel and are designed specifically to maintain dialogue with this core feature in the operating system. Because applications are designed around the OS kernel, changes to operating system impact installed applications. For instance, most windows applications utilize special executable routines for specific tasks. If an application request for a specific task and kernel has changed from 16 to 32 bits, the call will fail. Beta testing provides software manufacturers with a platform to measure application behavior if the adjustments are made to the operating system.

Resource dependencies require specific components, information, or access to information to produce a desired result. Applications make calls to a resource to perform or complete a specific task. By the way of comparison, there are several application typologies, which require resources. Client/server applications request access resources on a network. Stand alone applications operate on systems that request resources from the local machine. Mainframes manage a large volume of application requests from 3270 terminals.

Client/server applications send requests to special resources to complete a task (i.e. network resources, datasets, etc). For example, a user of Microsoft Word attempting to use a component, that was not installed on the local machine during the initial installation, may require access to the original source if the user requests usage of the feature at a later time. In this case, the application was initially installed using a network resource and the application refers back to this location to install additional components. This type of dependency relies on network access, availability, and authentication. Any changes to the three will return an error or failure.

Implemented on the local machine, Stand-alone applications require more system resources than client/server applications (network applications). Dependencies are formed on the local machine, which include memory, disk capacity, additional software resource, and more executable routines. Stand-alone applications use a special memory address for routines, and changes in the memory address will produce an error in the execution path.

Legacy applications require mainframe resources to function or provide a specific task. These dependencies are access to DB2 information, mainframe processors, tape libraries, and other mainframe elements. Legacy applications are accustomed to limited system changes and develop a number of environmental dependencies such as sharing like communication interfaces (highlighted in Case 1).

Case 2.Minimum Hardware Requirements.
*Manufacturers of a software product provide software specifications that list the minimum hardware requirements. These requirements are: minimum storage required, physical memory, video standards, physical network requirements, peripheral requirements, prerequisite software, etc. However, manufactures do not take into account, software already installed on a particular system. In most cases, existing applications are currently utilizing a portion of the physical and virtual resource and a new application increases utilization.*

Hardware dependencies are related to physical CPU components or peripherals to deliver a required effect. By way of comparison, consider Case 2. Today, most desktop publishing applications request a large percentage of available CPU processing. If an end-user is installing a messaging client that requires equal system CPU processing, this effect will create a conflict. This type of hardware dependency is exposed during the initial installation of the software creating a major effect on environmental dependency.

Protocol dependencies are an application's utilization of a specific messaging protocol to initiate communication with a desired program or component. Software in similar domains or manufactured by the same company share a common communication infrastructure (proprietary protocols). Computer Associates Common Communication Service (CA-CCS) is utilized to streamline compatibility and dialogue between Computer Associates mainframe application (z/OS) and applications on other platforms (e.g. Wintel, Linux, VM etc).[11] Working closely with TCP/IP, CA-CCS functions as the requestor and establishes contact and dialogue with Computer Associates products applications. Consider the job management example; CA-CCS facilitates communication between the job management application and mainframe automation. The mainframe automation software, OPS/MVS, manages alerts and sends the request back to the job management application to start the program. The overall goal of this tool is to integrate applications and assist in the management of IT resources.[11]

Security dependencies are related to user-authentication requirements for accessibility purposes. A particular application depends on proper access to network or components to function effectively. A classic example is a user requiring a special account to install an application. The software records special characters into the system registry. If the installed user executes the program, it will perform as designed. However, if another user attempts to execute the same program the dependencies formed by the application will return errors or not operate. During beta testing, this security dependency will be exploited if the user lacks the access right to a required resource. For example, if a user is installing an enterprise management application that requires system administrator (sa) access to a local SQL database. In addition, if the installer's lack the proper privileges, installation will fail immediately. Using the same example, enterprise applications require an initial system poll to create business process views (for system management purposes). If the installer and/or physical server lack access to a particular subnet (for firewall protection), nodes, or servers in the targeted subnet will return no data. This is the valuable feedback beta testing provides to enhance the product. In this case, the feedback will enhance the installation wizard or provide the user with the correct information to correct environmental issues.

### *3.2.2 Function Coverage Completeness*

Function coverage completeness is a beta testing objective that validates the software meets end-user functional expectations (verifying user interface). End-user functional expectations are verified during beta testing from correlated user feedback. The information is weighted against specifications from the produce life cycle's analysis and design phase to determine the features that require immediate addition and/or modifications to the product in test. This information, outlined in a software requirements report, outlines all the features of a specific application. Software developers reevaluate other requirements or product supplemental requests for future additions or product enhancements. The outcome of function coverage completeness refines the product in test and fosters future releases of the product.

Case 3. Beta testing a document management application

*Today, document-managing applications are increasing in popularity seeking to reduce the volume of physical documents, in turn creating a virtual paperless society and improving document standardization. Document standardization offers enterprise users' consistency, advance editing, and security for electronic documents. In this example, beta testers are testing a popular industry leading document management application. Beta testers of this product would expect the product to provide single function document conversion. The beta testers would also expect the product to offer document compression, minor security, and backwards compatibility. However, after completely testing the product, beta testers provided suggestions to have security wizards included with the application and a request to manage the level of document compression (e.g. low, medium, and high). Beta testers would also provide negative feedback on the quality of the document after conversion. The purpose of this test was to measure the acceptance rate based on expected and desired result.*

Before developers proceed to the coding phase, software specifications require finalization. Product owners utilize a software specification document (SSD) to correlate and present requirements to a team of developers. The product standards outlined in this document evolve during the design phase of the SDLC, from the targeted user community, which developers integrate into the application. However, the user requirements change over the course of the development process based on need, advancement in technology, or industry standards. During a software beta test, information from end-user feedback provides developers with the proper information to qualify whether the software meets reasonable expectations.

Reasonable expectations are those misinterpreted or under represented features in software, overlooked during the early stage of development or ignored by product development. Using Case 3 to illustrate this point, the product lacked “simple" functionality highlighted by the group of beta testers prior to sending the document managing application to beta. The single-function document conversion option, akin to converting a Microsoft Word document to Adobe PDF format, was a reasonable expectation to the product at the enterprise tier. However, developers easily overlooked the feature to allow the user to manage the level of compression to a document. In this example, the function coverage completeness objective was instrumental in capturing data used to refine the product in test. In addition, it was effective in presenting the customers expectation of the product.

Case 4. Expected functionality in Linux Word-processing Application.

*An enterprise company is scheduled to release a new Linux word processing application designed to compete with the major Windows-based word processing* *applications. The product is currently in the beta testing phase and a few beta testers of this application are actual database administrators (DBA). The beta testers have found a missing functionality when performing the "save" option on the local machine. The testers believe the product should provide the option to allow end users to save information to a comma-delimited flat file. When the beta testers select the print option, they are provided a base set of printers. However, there are no options to print to a flat file. Since DBAs often utilize flat files to create comma-delimited documents to easily upload information to a database management system (DBMS). The beta testers of the product consider this a major requirement.*

Integrating information into the current product, based on the level of feedback received from beta testers. Case 4 provides a classic example of the developers overlooking common functionality. Here the beta testers were able to provide information that required additions to the product to satisfy function completeness.

### *3.2.3 Measuring Localization Specification in Enterprise Software*

Enterprise corporations generate a large volume of its revenue in the global market. The LISA organization, a localization standards organization supporting major IT manufacturers and professionals in streaming business on a global platform, estimates that 20 of the largest IT companies are generating between 3 billion to 15 billion annually in revenue (estimates are stated in USD)[28]. Generating this amount of revenue in the foreign market creates challenges to focus on the proper testing and localizing of enterprise applications. Beta testing is critical in providing a proper evaluation process to cultivate enterprise software and assure the product will function as designed in the foreign market.

Beta testing is the final evaluation phase for localized software. In this phase, software is tested by foreign (or target) environments to validate the product is functioning correctly. More importantly is if the translation is functioning properly, software components meet foreign requirements (does not violate foreign policies) and content is demonstrated appropriately.

Localization specifications are components and functions added to software, which permit the application to function in foreign countries. More importantly, localization specification is a process by which software is analyzed and adapted to the requirements of other countries, making the software more usable for customers in targeted countries[10]. Localized software provides multi-lingual functionality, data exchange, prepares applications for foreign standards, assures the product operates with special peripherals and assists in meeting foreign country infrastructures. Beta testing supports the evaluation of localized functionality in software by recruiting clients with production environments in targeted countries to test global software features. The outcome of testing localization specification validates the product will work effectively in foreign information environments.

Localization is a common process in enterprise application because of the diverse level of end-users. Diverse users from foreign countries present unique challenges and various standards, different infrastructures, and multiple languages barriers. Localization is taking a product and changing it to assure that it is linguistically and culturally appropriate to the target locale (e.g. country, foreign region and language) where it will be sold and implemented[28]. Language barriers present major challenges for software developers. These barriers are formed when developing multi-lingual functionality, especially in the GUI design, and matching usage issues[17].

Case 5. Function mapping in localized software.

*Enterprise software providers are often faced with language barriers, which create terminology issues. These same issues regularly create screen layout issues when the character type and font change to adjust to the local language of country that utilizes a particular application. Another large issue is function mapping where implication changes in other countries. To illustrate this point, end-users selecting the "Save" option in an application must yield the same results regardless of language. If the word "save" has dissimilar meanings, application mapping is employed to resolve this conflict.*

An additional measurable localization feature is data exchange. LISA assists IT professionals in globalization, internalization, translation, and localization. This organization has over 400 corporate technology manufacturers and service providers creating a globalization community. An important goal of localization is managing data exchange to assure the quality of data, terminology, and information exchange is accurate and consistent. To assist corporations, The LISA organization has two standards benefiting data exchange translation memory.

TBX and TMX are vendor neutral open extensible markup language (XML) standards used to streamline the flow of data between foreign systems. XML is a method used to create common information formats to share data on the World Wide Web, intranets, and other web-enabled applications[6]. TBX is used to manage terminological data. This standard benefits terminology consistency in software packages and service related research. In addition, TBX promotes a centralized approach to terminology information, creating one database for terminology data in software. This standard provides an open methodology to allow terminological information to be more standardized and accessible to the public, producing a positive impact on industry localization.

As TBX provides an open XML standard structure, TMX mirrors this approach providing a standard for translation memory. Translation memory matches application sources and targets language segments. The translated data is stored in a database for future reuse [53]. TMX is a non-vendor XML standard streaming exchange of data between vendors without losing or corrupting data. For example, a translation memory database pairs equivalent sources to target language segments (e.g. sentences), together with the software, when provided with a source language input to translate, will search in the database and retrieve samples that closely match the input. The concept is target-language components can then serve as models for the translator[52].

Localization also assures the product conforms to foreign country specific peripherals or special input devices. A classic example is the use of specially mapped foreign language keyboards (non-English) end-users. This device must be properly tested to eliminate any character mapping problems.

### *3.2.4 Testing Enterprise Software Robustness*

Manufacturers employ various forms of software testing techniques utilizing standards, best practices, and procedures. However, it is difficult to measure or predict the correct outcome when an application encounters a unique production exception. For example, what happens if a user has an unusual delay between data entry? How does the application react to a warm reboot or if the operating environment hibernates? What effect does the application provide if a user enters erroneous data? If an exception occurs does the application respond favorably to these issues? Does the application predict this situation and make adjustments to continue processing? This objective measures how enterprise software responds to user habits capturing usage patterns used to improve the software. Testing software robustness validates dependability and anecdotally assures stability in enterprise software. Beta testing provides a platform to measure software robustness, which is the degree a software elements functions correctly when presented with a stressful environment or exceptional input[23].

Case 6. Multithreading Exception in a server management application.

*An administrator of an enterprise management application launched multiple instances of an alert management option to monitor a set of forty application-tier servers. The alerts were configured to monitor high CPU usage for the targeted servers. In the event of high utilization, it was configured to trigger email messages to on-call administrators. However, the software's multithreading feature was not designed to manage a large amount of instances of a single component, causing the application to crash the operating system.*

Developers of software have limited resources when benchmarking the performance of an application. If performance issues are discovered during the initial internal test, the product is tweaked based on the specific internal settings. Beta testing is the most effective method to test the product because of the "real" infrastructure issues it provides and user inference can be verified. Case 6 provides a typical example of stressful conditions exposing the robustness of an application. Users often stretch the limitation of application based on available resources. If an application that monitors servers in an enterprise, supports multithreading, and an administrator opens 40 instances of a single application to monitor 40 different servers causing the software to end abnormally. Here the end-user created a unique stressful situation with the application in test. The developer of this application will utilize the system dump to address this situation.

Case 7. Windows XP Service Pack 2 created exception errors for third party back-up applications.

*Microsoft Corporation released a major maintenance update for its popular windows operating system Windows XP Service pack 2. However, it created several exception errors with major software manufacturers. In particular, end-users who upgraded to Windows XP SP2 home edition, were unable to utilize a major enterprise third party application to perform full backups after installing the new service pack. The issue was created by Windows XP's new firewall restrictions. Microsoft released a fix requiring users of the product to create an exception in the firewall option in the Windows XP Suite of desktop operating systems.[29].*

In Case 7, the manufacturer was unable to predict exception errors with third party applications. Enterprise software developers do not focus on exceptional operation conditions when developing software[44]. The objective of software developers is to design enterprise software to adjust to exceptional conditions. There are several testing tools, benchmarking applications, and testing methodologies used to accomplish this goal. The Ballista testing methodology is used to simulate faults in an application and filter situations, which provide abnormal behavior[25]. Special APIs are used to introduce faults to the application under test and another applet is used to monitor and record results, creating a close comparison to beta testing because this test uses API instead of a source code.

Outside of internal rigorous benchmarking and testing, enterprise applications employ components to foster dependability to include fault tolerance techniques, testing and debugging, and quality management[48]. Dependable software is required on the enterprise level and encompasses the applications ability to never fail in production. The benefit of using beta testing to collect “actual” production data (correlated from user usage patterns) is to improve the applications reaction to exceptions. Beta testing is more practical because it provides data used to improve dependability, which directly impacts stability.

### *3.2.5 Measuring Software Vulnerabilities*

Today, security violations are common and the culprit of most system exceptions. Software vulnerabilities are creating issues with system outages, denial of services (DoS), an influx in spyware, and Trojan horse viruses. Security violations have affected how corporations secure environments to limit the amount and type of access to technological assets (e.g. data, shared resources etc.). Vulnerabilities are managed from two perspectives, which include limiting external access to secured entities and managing internal access to resources. Managing technological resources is an evolving task that requires the employment of several tools to assist in managing this need, which include:

- Information Firewalls (software/hardware)
- Administrator monitoring and information logging

- Access management software
- And security alerts (automation) software to name a few.

Corporations require an extensive level of security to be included with the operating system (e.g. Linux, Windows XP/2000/2003, IBM z/OS etc.). Equally, the security expectation is also implied for enterprise software providing an additional layer of security and the elimination of vulnerabilities.

Modern enterprise applications include security levels, designed to restrict access level to the applications and/or specific components of the application, which adhere to other security policies. It is important that the software does not compromise established enterprise security and user policies (i.e. end-user access to trusted domains), current firewall standards, access to information or data resources and access/content filtering. Enterprise applications inherit established infrastructure security databases such as Microsoft active directory, which uses directories to locate the proper user rights for a specific application. In addition, user access has special security to coordinate admission to required data resources in an enterprise. Firewall access is important when access to resources is required in another subnet of a network or external access. Filtering is a process of using special strings to limit admittance to internal resources. Filtering is a process used by a large majority of security tools.

Case 8. Software vulnerability in the implementation of SNMP opens the network to DoS vulnerabilities.

*Enterprise software corporations with software that requires Simple Networking Management Protocol (SNMP) version. 1 are vulnerable to the potential of a denial of service (DoS) attack. The DoS is exposed because an attacker utilizing a special script to intercept network information used by SNMP to manage alerts and other dialogue. IBM posted a vulnerability disclosure to its website informing end-users of vulnerabilities in a number of applications, which employs this version of SNMP[21]. The disclosures suggest the utilization of ingress filtering to eliminate the vulnerability. Ingress filtering manages the flow of network traffic as information enters a network based on rules established by the administrator[7]. Ingress filtering prohibits external traffic to internal services. This is normally implemented using a firewall.*

The SNMP vulnerability (highlighted in Case 8) influenced how enterprise software utilizes this protocol for communication. Enterprise software manufacturers are faced with the challenge of how to use SNMP in the future to eliminate this vulnerability. In this case, a beta tester would evaluate whether the firewall security is in place to properly secure SNMP v.1 and eliminate any conditions that create a security risk to the corporation. In addition, the test validates firewall standards are not compromised during implementation. Beta testing also confirms other security issues, such as data can be accessed without violating security, user access is configured properly and other security related issues.

### *3.2.6 Feedback from User Interface Accessibility*

User interface accessibility (UIA) is comprised of features in software designed to assist end-users with physical disabilities, dexterity, or special limitations. UIA is designed to enhance and ease the software transition (operation). Governed by the Rehabilitation Act of 1998, UI accessibility requires software manufacturers to modify technology for individuals with physical disabilities. From a development perspective, it is easier to manage accessibility requirements during the design and specification phase. In addition, testing is simple because most accessibility options are direct. However, there are cases of accessibility exceptions, which were not part of the product scope during the early development phase.

Case 9. Enterprise Management Profiles

*An end-user of an enterprise management application has carpal tunnel syndrome in both wrists. He is in the process of building a network profile that will monitor a subnet of applications for server outages. The EM applications require the end user to drag network components to specific repository. Because of his limitations in using the mouse, the drag-and-drop feature is limiting this ability to build the required profile. During beta testing this issue was reported and the software developers added a screen to the application enhancing the drag-and-drop feature by allowing the end user to delay when dragging items to a repository or using a component select feature. This feature allows the end-*

*user to select the desired network components and add them to the desired repository.*

Software developers simulate standard accessibility components in enterprise software. Standard components are keyboard shortcuts, screen and color augmentations, command line options, accessibility scripts, and special sound notifications. Keyboard shortcuts used to correlate a set of actions into one key. Screen and color augmentations are used to enhance the view of applications for individuals with visual limitations (i.e. the screen can be customized to increase the monitor view above 200%). Special command line options and accessibility scripts are utilized for end-users who may have dexterity issues using a mouse. With this option, users enter or execute a script (which in this example, is a single term resulting in the execution of a batch file) or use a special command to carry out a task or series of tasks. End-users with auditory challenges utilize sound accessibility, which encompasses the employment of more screen colors and pop messages for assistance with application alerts. These options are easier to simulate internally.

### *3.2.7 Feedback from User Interface Usability*

Enterprise software companies are faced with the challenge of designing software to meet a robust user base. This encompasses designing a software interface that is functional but practical to the common user. In addition, the software should reduce the learning curve and provide interface consistency.

Corporations have focused a large amount of time on research and development with the purpose of enhancing the end-user experience.

The pressure from major buyers of enterprise software has forced software manufacturers to focus on usability. The Boeing corporation, the worlds leading manufacturer of commercial and government aircrafts, believes usability is the most significant factor of total cost of ownership (based on an internal study)[49]. The aerospace giant seeks to eliminate usability issues prior to considering the software for its production environment. To assist software manufacturers with standardizing usability concerns, the National Institute of Standards and Technology has created a Common Industry Format for Usability Test reports (CIFs) now adopted by major enterprise software producers.

CIFs are reports, generated from usability tests, used by major software suppliers as templates to validate the product demonstrates ease of use. The purpose of CIFs are to challenge corporations to understand the needs of end-users; establish a common usability reporting format for sharing usability test data; determine the value of usability reports; and determine the value of usability information for software procurement[5]. Major software manufacturers, such as Microsoft, has adopted CIF to enhance UI usability in its suite of windows based operation systems[49].

As user interface usability (UI usability) is the ease to which a user adapts to software, usability testing measures user adaptation at the graphical user interface layer. This is important because software developers often misinterpret user requirement in the specification phase. Commonly, end-users discover UI anomalies after the product is implemented or the application is used for the first time.

Case 10. Word-processing Document Wrapping Issue

*Users of a popular word-processing (WP) application noticed significant changes when upgrading from WP version 2.0 to the new version 3.0. The changes to the application included new toolbars, input options, and organization of special components. The most significant change is the actual document area is now cluttered with new toolbars that were not in version 2.0. In addition, when users open documents created in WP 2.0 the data does not wrap causing the user to reduce the size (and font in some cases) to fit the new screen layout. This is a serious concern with experienced users of the product.*

One of the most common UI usability issues are user interfaces that reduces the practicality and impacts the functionality of the application. In Case 10, the manufacturer of the WP application introduced layout changes to the application, which impacted the usable area designated for managing, creating, and updating documents. The addition of new toolbars and attributes (new application features) also reduced the usable area and created view issues when the user opened older file versions. In this example, developers of the software clearly overlooked the impact of the end-users functional area and changes to older documents during internal testing creating functional issues. Beta testing is designed to assist the software developers with relevant user feedback on functionality issues.

Case 11. DB2 Tuning Difficulties

*A large tax consulting organization purchased a third-party DB2 resource performance and optimization application to address resource intensive DB2 application SQL statements. However, end-users have not accepted the application because of its lack of interface consistency with other DB2 applications, which include difficulty to navigate using TSO terminals, and difficulty in understanding how to generate reports (to include the quality of statistical output). The organization has decided to shelve the product costing the company thousands of dollars in licensing cost.*

An increased focus on UI usability, by software manufacturers, influences the end-users learning curve because of application consistency. Case 11 provides an example of end-users unfamiliarity with a product leading to non-acceptance. In this case, the product developer must focus on revamping the product to be more consistent with the software it is developed to support. Actual field testing provides the most prevalent feedback for UI usability. Beta testers of software provide the first level of usability issues in software providing quality feedback used to impact final release of the product.

## 3.3 Major Categorized Software Attributes for Beta Testing

The core objectives of beta testing software is to assist beta test managers and software developers in structuring the focus of a product ready for beta testing. However, this segment outlines software attributes for each product to enrich the outcome of each objective. The software attributes are essential parts of the common enterprise application, which requires validation to achieve a specific testing objective. Attributes are characteristics of an enterprise application that are can be represented by a numerical value. The Attributes are segregated into sections, and weights are assigned, by the potential level of problems they create in production. Since, some attributes provide multiple objectives, overlapping is common. This section outlines and describes the significance of each attribute. An overall mapping of attributes to objectives is provided in Table 1.

**Table 1. Beta Testing Objectives Attributes Mapping Table**

| **Attributes** | Variable | Environmental Dependency | Function Coverage Completeness | Localization | Robustness | Software Vulnerability | UI Accessibility | UI Usability |
|---|---|---|---|---|---|---|---|---|
| class inheritances | **CI** | | | | ● | | | |
| code locale | **CL** | | | ● | | | | |
| incoming client service request | **CS** | ● | | | | | | |
| CPU utilization | **CU** | ● | | | | | | |
| program switches lacking default clause | **DC** | | | | ● | | | |
| changes to dynamic data | **DD** | | | | ● | | | |
| dynamic link library requirements | **DL** | ● | | | | | | |
| fonts effect on UI | **FE** | | | ● | | | | |
| firewall port required. | **FP** | ● | | | | ● | | |
| hardware and software requirements | **HR** | | ● | | | | | |

| Attributes | Variable | Environmental Dependency | Function Coverage Completeness | Localization | Robustness | Software Vulnerability | UI Accessibility | UI Usability |
|---|---|---|---|---|---|---|---|---|
| user interface controls | IC | | | | | | | ● |
| special input devices | ID | | | ● | | | | |
| languages supported | LS | | | ● | | | | |
| message boxes | MB | | | | | | | ● |
| multitask command buttons | MC | | | | ● | | | ● |
| multiple system requirements | MS | ● | | | | | | |
| network connections required | NC | ● | | | | ● | | |
| open API | OA | ● | | | | ● | | |
| physical processors required | PP | ● | | | | | | |
| amount of source code | SC | | | | ● | | | |
| special fonts | SF | | | | | | ● | |
| special hardware requirements | SH | | | | | | ● | |
| screens affected by font adjustments | SA | | | ● | | | | |
| screen traversal | ST | | | | | | | ● |
| special user accounts required. | SU | ● | | | | ● | | |
| software wizards | SW | | ● | | | | | ● |
| amount of threads generated | TG | ● | | | ● | | | |
| use case totals | UC | | ● | | ● | | | |
| user interface complexity | UI | | | | ● | | | |
| user-levels | UL | | | | ● | | | |
| UI responses required | UR | | | | | | | ● |
| unrestricted text fields | UT | | ● | | ● | | | ● |
| web portals required | WP | ● | | | | ● | | |
| web service request requirements | WS | ● | | | | | | |

### *3.3.1 Environmental Dependencies Attributes for Beta Testing*

The focus of the environmental dependencies objective is to test an applications reliance on hardware, shared software attributes, and other application to validate the required dependency does not impede operation. The essential attributes highlighted in this objective are:

1) physical processors – the minimum number of physical processors required for operation
2) network connection – the total number of network connections (node-to-node) requirements
3) multiple system requirements – the amount of sub-systems required for operation (e.g. cluster requirements)
4) CPU utilization – CPU utilization required to manage tasks generated by the software
5) dynamic link library (DLL) – total number of shared files or access to shared files required to function effectively
6) security access rights:
    a. firewall port access - the total number of physical ports required (both suggested and required)
    b. user account stack - requirements for special user accounts to operate the software
7) web portals – the maximum number of web portals supported by the software

8) incoming client request - the expected amount of clients requesting services
9) open API – the number of open API required to support the application
10) web services requirement - The required amount of web services needed
11) threads generated – the total number of threads generated by the application

Each attribute is unique in measuring the environmental impact on its targeted environment.

Applications requiring a high-demand utilization are commonly employed on computers with multiple processors. Multi-processor systems are computers with two or more physical processors. Multiple processors are used for applications that support load balancing (clustering), robust transactions, and servicing requests from a large user base. However, each processor creates a new dependency because of the amount of CPUs required to manage enterprise-level activity.

Network connections are the number of communication requirements to support a specific platform or user base. Applications supporting multiple network connections create unique dependencies for each. For example, email applications require a number of messaging protocols to function effectively such as SMTP, POP3, and x.400 (for European users). All of the applications rely on

TCP/IP in a network environment. If a connection is eliminated the application may return errors because of the dependency to that communication conduit. During the beta testing phase, each network connection requires testing to validate each network connection.

Multiple system requirements are attributes of an enterprise application that require access to different platforms to operate successfully (e.g. sub-systems). This scenario is referred to as clustering environments, which creates the highest form of dependencies because each entity relies on another. Clustering is a concept used for high availability in computing environments, which is the connection of multiple systems to form one unit. This interconnection is formed by multiple computers (i.e. application server, storage server, DASD), databases, and other systems. In job management environments a sysplex is created to support the executions of multiple programs in an environment. However, if the shared DASD is unavailable the entire system could fail. This attribute is essential to account for when planning to beta test a product.

Developers of software provide a system resource baseline (recommendation) for applications. The baseline is based on minimum and recommended ratio, which is derived from the applications sensitivity to CPU utilization. If the sensitivity to CPU utilization is low the opportunity to develop a dependency is minimized. In this scenario, beta testers will install the product on systems at the lowest baseline (provided by the software manufacturer) and the best recommended. To illustrate this point, Microsoft Exchange 2003 is a messaging server application designed to support enterprise email. This application recommends the minimum processor configuration to be an Intel Pentium or compatible 133-megahertz (MHz) and recommended Intel Pentium or compatible 733-MHz processor[33]. The developers of the application have set a baseline for the product, however, sensitivity is measured after implementing the application to a production setting. The CPU utilization attribute will expose dependencies during the installation of the product by introducing system degradation.

Environmental dependencies are also created by data or resources stored in dynamic link libraries (DLL). Applications that require access to shared DLL (DLL provided by an operating environment or application) initiate a level of dependency. In this case, the more DLL required the greater the dependency. A classic example is spreadsheet applications. For most of its processing, access to DLL are required to facilitate special printing functions, complex calculations, access to embedded objects, database access, and special character utilization (to name a few). Each DLL requirement creates a special test case and requires validation in the beta testing phase.

Security dependencies are introduced when applications require access to special ports on an enterprise firewall. Additional security dependencies are created when the application requires special user accounts to function effectively.  An example of these attributes is an application that requires access to multiple firewalls and bridges such as agent management software.  Agent management applications require an environment discovery prior to building special alerts (e.g. utilization high alerts, storage capacity alerts, etc.).  This discovery requires special access to firewall ports to build the database.  In addition to port access special accounts are required to initiate discovery (e.g. administrator accounts, power user etc.).  If accesses to these ports are not available, the application will return unfavorable results thus forming a dependency.

Web portal requirements are measurable attributes valuable to the enterprise dependencies objective.  Portals are individual web accessible content collected on a single screen creating a customized Internet gateway.  Each portal requests a separate service for a specific task.  Many portals request increased environmental dependencies because of the amount of common gateway interface (CGI) and client service requests required to facilitate the demand.  A prime example is Yahoo! Incorporated provides one of the most widely used portals to web users[54].  This website contains individual portals servicing access to music, eMail, instant messenger, stock quotes, news bloggers, and other valuable web resources.  In this example, each resource requires a large amount of scripts and services to function effectively, thus increasing the amount of client services and CGI creating more dependencies.  These are valuable performance attributes increasing the amount of dependencies for a particular application.

Like CGI and other services, open application programming interfaces (open API) create dependency requirements because each API demand initiates a separate request for service.  If a windows-based application initiates peer access with another application on the same network, it will employ NetBIOS to request the services required to exchange messages.  Other open API includes MAPI and ODBC commonly used with enterprise applications.

Web services are an additional attribute of enterprise dependency objective facilitating the exchange of data via the internet.  Examples of web services are applications that use XML to create standardization for document data exchange, an application that uses Google to search for resources on the web, or electronic data interchange (EDI).  Each separate web service creates a unique dependency.

Client service requests are based on the number of services required for operation.  Client services initiate or facilitate some action requested (or required by the application).  However, services may also act as a facilitator to accept request from an external entity (i.e. messenger service handles alert request). It is important to measure the expected amount of requests to test the scalability of an application. If requests are high the number of use cases increases creating the potential for a problem.

Adjacent to accounting for the expected amount of requests for client services are the amount of threads required by the program. Threads are individual program instructions initiated by a program to execute a specific tasks. Each thread requires hardware or system resources. If the amount of threads required is high it will create a large demand for resources increasing the level of environmental dependency.

### *3.3.2 Function Coverage Completeness Attributes for Beta Testing*

The focus of the function coverage completeness objective is to validate that the application in beta meets the customer expectations. This objective matches the goal of the application with user inference, which is the essence of beta testing. The attributes outlined for function coverage completeness are:

1. hardware and software requirements – the system requirements provided by the software manufacturer
2. use cases – variations of actual usages of the software (derived from the product specifications)
3. software wizards – provides assistance with robust tasks
4. Unrestricted text fields – the input area on a UI that require static data from the end-user

Customer expectations are refined with ease of use. However, ease of use is perception managed by the application design. Hardware and software requirements are those physical resources necessary to operate the application. Complexities increase when the amount of requirements is more than the average application in its domain, which consist of hardware and operating system requirements. If the software is complex, it lowers the functional coverage of the application and may require a special skill to implement the product.

Enterprise applications include a set of product specifications, which describes how the product is designed to function. Each specification represents a classic specific use case. More specifications increases the difficulty in creating use for test purposes because end-user environments are unique and may create dissimilar challenges.

Akin to switches, enterprise applications provide software wizards’ design to ease application usage by assisting users through a series of instructions to fulfill an operation or task. Wizards are helpful to train the user through use cases. However, they are often complex and require dedicated threads. If an application offers a large volume of wizards, the complexity of the application increases which negatively influences the software coverage.

Unrestricted text fields are an attribute of function coverage completeness that if provided in abundance lowers expectations of the user. Unrestricted text fields require static data from the end-users. End-users prefer user interfaces that require a limited amount of data or most of the data is provided using more drop-down boxes or options.

### *3.3.3 Localization Attributes for Beta Testing*

Enterprise software adjusted to function in foreign countries are properly localized. However, the extent of localization is managing language and conversation changes in the application. The attributes of the localization objectives are:

1. special input devices – peripherals used to enter data or information into the software
2. languages supported – the total foreign languages supported
3. font effect on UI – adjustments foreign font changes make to the user interface
4. screens affected by font adjustments – the number of screens affected by foreign font changes
5. code locale – the number of code locale used to make geographical adjustments in the program.

Each attribute is unique in its ability to provide functional software to the targeted foreign market.

When software is localized for other countries, there are occasions when special foreign input peripherals are required to optimize utilization of the application (e.g. Hebrew Keyboard, Portuguese layout, and other foreign proprietary keyboards) . For example, software localized for the Japanese market may require a kanji keyboard to optimize utilization of the software. Each special input device requires special experts for beta testing to receive user approval.

Another attribute important to software localization is the number of foreign languages supported by the application. More languages extend the testing time required to validate no language or translation errors exist in the application. If the number of countries' languages supported are low, it will reduce localization testing time.

As languages change, special characters are employed to provide UI, which is readable by the targeted market. The fonts required for foreign markets may have some effect on the UI layout. In this case, each screen requires validation to assure font changes do not negatively affect UI. For example, if an application is targeted for the Japanese market, changes in the Japanese character expand and reduce the size of the UI. As changes are statically implemented, several screens will be affected. If font changes are high, the beta testing time is negatively impacted.

Code locale (or locale objects) is important to localized software. Code locales manipulate programming terms to adjust to foreign languages and regions. It affects time, dates, metric systems, etc. For example, displaying a number for a specific country is a locale operation because the number must be formatted according to the conventions of the end-user's country, region, or culture[47]. However, many code locales create more problems and increase the level of maintenance and manipulation. In addition, each locale requires validation in the beta testing phase, which impact the test time.

### *3.3.4 Robustness Attributes for Beta Testing*

The focus of the robustness objective is to identify wrong data, and validate how user errors and usage patterns impact the software (error handling). The attributes outlined in this objective seek to measure those features that will strengthen the robustness of enterprise applications. There are nine attributes:

1. software use cases – variations of actual usages of the software (derived from the product specifications
2. user interface complexity - the total number of input fields and command buttons on a user interface
3. unrestricted text fields – the input area on a UI that requires static data from the end-user

4. amount of source code – the total of lines-of-code in the entire application
5. amount threads generated – total threads generated by software dedicated to request made by the application
6. class inheritances – total amount of properties derived from the main class (general class)
7. multitask command buttons – amount of single UI input options that initiate a series of functions
8. user-levels - the end-user accounts and/or user stack levels required to successfully execute an application
9. changes to dynamic data – the amount of updates made to data that is managed or shared with the operating environment.
10. program switch statements lacking a default clause – switches not created dynamically or provided by the software documentation.

Software use cases are the amount of outcomes or scenarios an application provides based on request from the end-user(s). The amount of use cases are derived from the list of specifications provided by the software manufacturer. If the amount of use cases is large, the complexity of the application is increased. If use cases are low, it decreases the likelihood of usage errors and streamlines validating the robustness of an application. For example, a virus protection application is more robust than a photo editing application because the virus software has less use cases. The average virus protection application is limited to scanning the files of a system for potential viruses and protects the system from future threats. However, photo-editing application typically provides a large list of photo editing specifications, which may complicate the outcome of a specific task.

Complexities of the user interface are the amount of data input fields, restricted text boxes, combo boxes, radio buttons, and drop down boxes in the user interface. If there are a large number of input locations, attributes requiring some action, or an abundance of buttons, the probability of an error increases and data sequence complications increase. To illustrate this point, users of most eCommerce sites are required to submit a large amount of data when ordering a product or service. If the user enters the wrong data, or fails to enter the required information the application may provide some adverse reaction (or do nothing). In addition, this increases the complexity of the application.

Lowering the amount of unrestricted text boxes decreases the amount of end-user errors. Unrestricted text boxes on a user interface include blank text boxes, empty input cells, and label boxes. All of the input fields are unique in the information required to initiate a specific task or yield a result. An increased amount of unrestricted input fields negatively affect the type of data, the desired result, and the flow of data increasing the complexity of the software. For example, users of a helpdesk application use online forms to open a problem request. One of the input devices requires the user to enter the date of the problem in a specific format (i.e. *yyyyddmm* – Year, two-digit date, two-digit month). However, the user enters the date incorrectly. The application may immediately alert the users of the error, accept the data and store the results creating invalid record locators, or do nothing.

Although seamless to the end-user, the amount of source code in an application affects the robustness of an application. If the amount of source lines of code is large, software testing is extended and often require more physical resources. In addition, in the event of an error where a software fix is required, implementing the fix into the current code is complicated requiring greater compilation time[24].

Another coding attribute are software threads. Software produces separate threads to facilitate request initiated by the program user. However, if an application generates too many threads, control issues surface and resource over load increases. For example an application has initiated a series of threads to validate the user access level, request access to a data resource located in a storage area networks (SAN), and resources from the operating system kernel. If a thread initiated for resources on an unavailable SAN, the software loses control of thread and could potential create an error. The same scenario requires a large volume of data to be written to facilitate the request, which in excess increases the probability of a system overload.

Sub-classes inherit properties from general classes. In programs designed to use many levels of classes increases the complexity of the application and introduces the potential of future errors.  If a program inherits a class, currently utilized by another object (or program) the potential for errors are prevalent.  If the number of class inheritance is low, the properties are more precise.

Software project complexities occur when multi-command buttons are present. Multi-tasked command buttons are those functional attributes of UI that initiate a specific set or series of tasks when executed by the end-user.  Project complexities are measured by how many separate functions or scenarios are created when a button is selected on any functional option on a form.  A single functional attribute responsible for multiple tasks introduces robustness issues because many threads are required.  This scenario may confuse the end user and increase the probability of errors.

User level requirements are the amount and types of specific end-user accounts required for an application.  If multiple levels of user accounts are mandatory, the risk of erroneous data increases (or introduction of user errors).  To illustrate this point, if a knowledge base application that allows multiple accesses to the library stores data, any user could add or delete important information.  In this scenario, beta testers must validate each user-level use case to eliminate the potential for errors.  If user level access is low, testing is limited and robustness is stable.

Often software extracts data from a database (or data storage source) and is replaced by an application.  However, there are instances when the operating environment dynamically changes or reallocates the information. Applications using the Microsoft platform have access to the Dynamic Data Exchange Management Library (DDEML) to share dynamic data. If multiple applications in a windows environment are using the DDEML to share common data, conflicts occur (i.e. resources cause an application to stall) when the data is changed prior to the application returning the data. This conflict causes data corruption. In instances where large volumes of changes are required to dynamic data, an extensive test is required to validate most uses cases. The lower the number of required updates reduces the probability of future problems.

Developers of software include a number of code switches used to change the type of execution based on a set value. A default switch in the application provides the specific clause to the end-user without end-user intervention. This attribute focuses on the non-default switches in a program. For example, in legacy application, batch programs are often initiated from a command line on the console. When an operator executes a batch program, the program name is followed by a switch that controls how the program executes. If the application contains a high amount of non-default switches, software coverage is low.

### *3.3.5 Software Vulnerability Attributes for Beta Testing*

The software vulnerability objective measures the application to exploit potential security violations. The objective's attributes measure the features of software, which requires communication for operation. The attributes identified in this objective are:

1. web portals – the maximum amount of web portals supported by the software
2. open API – the amount of open APIs required to support the application
3. network connection – the total number of network connections (node-to-node) requirements
4. firewall port access - the amount and type of physical ports required
5. user accounts – total number of user accounts required

Web portals (described in section 3.3.2) utilize various services to provide a single collection of content. Since this technology initiates communication dialogue with the service provider there is potential for security vulnerabilities. This is extreme in cases that allow instant message portals, push updates, web mail, and content bloggers (e.g. stock tickers, news updates, technology alerts, etc.). As web portals are created the level of vulnerabilities increase with each additional web service request.

Additional security vulnerability is the software utilization of open application programming interfaces (open API). This attribute measures the risk associated with the employment of standardized API to communicate with other software or systems. Today, worm and Trojan horse viruses are created to utilize MAPI to transmit the virus to other systems. Again, if the requirement of many open API is mandatory, this increases the vulnerability level in the software.

Network connectivity is a standard in common enterprise applications. Network connections are the conduit used to maintain communication. However, each connection, if not properly secured, presents a vulnerability. If the number of connections required is large (system to system), the risk of exposure increases.

Firewalls are designed to manage and monitor the traffic in a networking environment. However, many enterprise applications require access to firewall ports for operation. Applications requiring access to common firewall ports increase the risk of system attacks thus, lowering the security of the application. Enterprise monitoring applications often rely heavily on access to firewall ports to monitor system states to generate system alerts. Each special port request increases the risk of vulnerability in the software in the environment.

Enterprise applications are designed to support a robust user base. However, access to the system heightens the level of security complexity. Access to the system requires monitoring to eliminate the existence of user-based threats. Using messaging software to illustrate this point, eMail servers support a large volume of users. Most of the system viruses enter and exit enterprises using messaging. Software developers of applications that allow a large base of users must thoroughly test attributes, which reduce vulnerability in the software.

### *3.3.6 UI Accessibility Attributes for Beta Testing*

User interface accessibility (UIA) objective of beta testing validates the features of an application designed to assist end-users with special physical needs. The attributes for UIA are formulated differently from other objectives. Since most accessibility attributes are seamless and can be thoroughly tested by the internal group of developers, the amounts of UIA attributes are limited to:

1) the amount of special fonts
2) and special hardware dependencies

Both attributes are unique in collecting information used to improve the experience of users with limited capabilities.

Special fonts are a style of character used to provide special UI presentation for end-users with visual challenges. Operating platforms provide a special repository used to store shared fonts. Users with special visual challenges require fonts which are legible and contain basic dark colors. Applications designed with visual accessibility features, such as the ability to enlarge the size of text, must be thoroughly tested. To illustrate this point, most word processors applications have visual accessibility features which offer the end-user the ability to increase the size of tool bars eliminating the words associated with a specific command button (i.e. the print button would replace the word “Print” with a special printer character). If the amount of special fonts provided by the application is low, it limits the accessibility of the application.

Applications should be designed with accessibility features not relying on special hardware to function appropriately. If an application requires such additional hardware as the use of a special magnification screen for the monitor or special function character type keyboards it creates hardware dependencies.

### *3.3.7 UI Usability Attributes for Beta Testing*

User interface usability is the key goal of beta testing. This objective focuses on validating that the graphical user interface is simple and promotes ease of use. There are several attributes that form the nucleus of this objective, which are:

1. user interface controls – the UI graphical fields and option
    a. calendar control
    b. radio buttons (option)
    c. check boxes
    d. command button
    e. combo box
2. multitask command buttons – amount of single UI input options that initiate a series of functions
3. unrestricted text fields – the input area on a UI that requires static data from the end-user
4. screen traversal in an application – the debt of each individual screen
5. UI responses required – total message box that prompt some response from the end-user

6. messages boxes – the text boxes which prompts warning, error, or provide instructions to the user. This attribute is more concerned with the code provided and the amount of shared meanings per code.
7. software wizards – provide assistance with robust task

The attributes are designed to measure the graphical user interface providing an overview, which is used to eliminate UI complexities.

User interface controls are application objects that allow an end-user to initiate a task, control the screen, or spawn an action. The most common types of UI controls are command buttons, radio buttons, check boxes, combo boxes, link bards, calendar controls, and scroll bars. Software that contains a large volume of controls increases usage complexity, broadens the amount of use cases, and increases the users' learning curve.

Multi-tasked command buttons are UI functional attributes that initiate a specific set or series of tasks when executed by the end-user. When the muti-tasked command buttons are large in volume, it reduces the software simplicity because a user is not able to control the flow of the application, reducing the learning curve. If the user of a database application selects the "eliminate redundancy" command button, it will run a query that eliminates data based on a set of predefined key fields and generate a report. Most end users would view this command as complex.

A large volume of unrestricted input devices lowers the end-users learning curve. Unrestricted input devices are text boxes or input cells that require static data from the end-user. If the UI has too many unrestricted text boxes it increases the complexity of the application and users may lose control. The most common example are eForms, which collect user data and store it in a database.

The amount of traversal a screen provides is an essential attribute of UI usability. This attribute measures the debts of a form to determine the level of complexity in application (e.g. sub menus). If a screen is nested with many successors, the flow may become confusing to the end-user creating usage problems.

Another feature of UI that increases the software complexity is the amount of responses required for message boxes (e.g. non-warning messages). UI responses are the request for a specific action to be delivered by the application. For instance the logon/logoff boxes, save message boxes, print boxes, etc. There are also responses, which verify an action prior to initiation. Although helpful to end-users, a large volume creates complexities and reduces usability of the application.

Applications often provide message boxes instructing the user of some action or prompting the user prior to executing a command. If an error or warning occurs, the message box displays a code and commonly an explanation of the code, which may include several meaning per code. This is not ideal for an application because it confuses the end-user in-turn negatively affecting usability.

Wizards are designed to assist a user in streamlining use cases. They are also used to train users on a process; however, a large volume of wizards included with an application increases the complexity of the application by introducing steps in a process that are not needed or desired by the end-user. For example, if a system administrator uses the “user creation wizard” to create a new user account on a network, the wizard will create an account based on responses by the administrator. However, if the administrator needs to restrict certain access this feature does not allow fine-tuning. In this case, it would be more efficient to create the account natively.

## 3.4 New Software Metrics for Predicting the Priority of Beta Testing Objectives

The next step in identifying which objectives are more vulnerable for a product or class of products is to use attribute weights and apply the correct formula definition. Each objective contains a metric and set weighted attributes, which apply to the general enterprise application. Prior to each beta test, enterprise software developers utilize metrics to better manage the pending project. The outcomes will provide a prediction for the area requiring the most attention, assist software manufacturers in determining how to allocate time during the beta testing process, assist in questionnaire design, and assist in the prediction of total beta testing durations.

In this section, attributes from two small samples will be used to demonstrate how the vulnerable points are properly calculated. The first application is a government imaging application that is designed to store ship requisitions for a commissioned war ship. The images are stored on a secured database and extracted using secured front-end proprietary application. This application is web-based and designed for users with special physical limitations.

The second application is an enterprise check processing application designed for a major bank with a global presence. The check processing application is designed to handle a high volume of personal checks assisting in the posting of funds in accounts in a shorter period.

### *3.4.1 Environmental Dependency Metrics*

Outlined in Section 3.3.1 environmental dependency is important to the beta testing process to measure the impact of application dependencies on operation of the software product. Definition 1 is used to calculate the vulnerable point for environmental dependency.

Using data collected from defense imaging application, first Algorithm 1 is used to calculate the proper attribute weights for the current project (See Table 2). After attributes are weighted, Definition 1 provides a predicted risk value of 2.714, which is an average problem for this application.

The calculations are demonstrated as:

e = 1(.082) + 4(.102) + 3(.082) + 3(.102) + 2(.082) + 3(.061) + 3(.061) + 3(.102) + 3(.082) + 3(.102) + 2(.061) +2(.082)
e = 0.082 + 0.408 + 0.245 + 0.306 + 0.163 + 0.184 + 0.184 + 0.306 + 0.245 + 0.306 + 0.122 + 0.163
***e = 2.714***

**Definition 1**: Metric for predicting the vulnerable points of software environmental dependency is a function of:

$e= w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7 + w_8x_8 + w_9x_9 + w_{10}x_{10} + {}_{11}x_{11} w_{12}x_{12}$

where $x_1$ is the number of threads generated;
$x_2$ is the number of CPU utilization;
$x_3$ is the number of DLL required;
$x_4$ is the number of firewall ports required;
$x_5$ is the expected amount of incoming client service request;
$x_6$ is the number of systems required;
$x_7$ is the number of network connections required;
$x_8$ is the number of open API;
$x_9$ is the number of physical processors required;
$x_{10}$ is the number of special user accounts required;
$x_{11}$ is the number of web portals required;
$x_{12}$ is the web service request requirements
and $w_1$, $w_2$, $w_3$,…. and $w_{12}$ are their weights for relative importance.

Table 2. Environmental Dependency Metrics Data

| Environmental Dependency | | | |
|---|---|---|---|
| i | Attributes | Normalized Attribute Value | Weight |
| | | x | w |
| 1 | amount of threads generated | 1 | 0.082 |
| 2 | CPU utilization | 4 | 0.102 |
| 3 | dynamic link library requirements | 3 | 0.082 |
| 4 | firewall port required. | 3 | 0.102 |
| 5 | incoming client service request | 2 | 0.082 |
| 6 | multiple system requirements | 3 | 0.061 |
| 7 | network connections required | 3 | 0.061 |
| 8 | open API | 3 | 0.102 |
| 9 | physical processors required | 3 | 0.082 |
| 10 | special user accounts required. | 3 | 0.102 |
| 11 | web portals required | 2 | 0.061 |
| 12 | web service request requirements | 2 | 0.082 |

### *3.4.2 Function Coverage Completeness Metrics*

As outlined in Section 3.3.2 function coverage completeness validates customer expectations. In this example, data collected from the check processing application is utilized. Algorithm 1 is used to calculate weights for each attribute (Table 3). After securing the correct weights, Definition 2 is employed to predict the risk level for this objective, which is later compared with other objectives. The calculated risk value for function coverage completeness is 2.70 predicting an average level problem for this project. The calculations for this objective are:

$f = 3(.167) + 3(.333) + 3(.250) + 2(.250)$
$f = .501 + .999 + .750 + .500$
$\boldsymbol{f = 2.750}$

---

**Definition 2**: Metric for predicting the vulnerable points of function coverage completeness is a function of:

$f = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$

where $x_1$ is the number of hardware and software requirements
$x_2$ is the number software wizards;
$x_3$ is the number of unrestricted text fields;
$x_4$ is the number of use case totals;
and $w_1$, $w_2$, $w_3$, and $w_4$ are their weights for relative importance.

---

Table 3. Function Coverage Completeness Metrics Data

| **Function Coverage Completeness** | | | |
|---|---|---|---|
| **i** | **Attributes** | **Normalized Attribute Value** | **Weight** |
| | | **x** | **w** |
| 1 | hardware and software requirements | 3 | 0.167 |
| 2 | software wizards | 3 | 0.333 |
| 3 | unrestricted text fields | 3 | 0.250 |
| 4 | use case totals | 2 | 0.250 |

### *3.4.3 Localization Metrics*

Repeating the process in Section 3.5.2, the data collected from the check processing application is used to demonstrate how to obtain the vulnerable point for localization, which is focused on measuring proper preparation for the global market. While the localization attributes are limited, the weights assigned to each are larger to compensate for a partial set of variables. Using the information in Table 4. Definition 3 is used to calculate the predicted risk value for localization is 3.421. The calculations for this objective are:

$l = 3(.263) + 3(.211) + 4(.158) + 3(.105) + 4(.263)$
$l = .789 + .632 + .632 + .316 + 1.053$
***$l = 3.422$***

---

**Definition 3**: Metric for predicting the vulnerable points of localization is a function of:

$l = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5$

where $x_1$ is the number of code locale (locale objects);
$x_2$ is the number of fonts effect on UI;
$x_3$ is the number of languages supported;
$x_5$ i is the number of special input devices;
$x_5$ i is the number of screens affected by font adjustments;
and $w_1$, $w_2$, $w_{3,}w_4$ and $w_5$ are their weights for relative importance.

---

Table 4. Localization Metrics Data

| Localization | | | |
|---|---|---|---|
| i | Attributes | Normalized Attribute Value | Weight |
| | | x | w |
| 1 | code locale | 3 | 0.263 |
| 2 | fonts effect on UI | 3 | 0.211 |
| 3 | languages supported | 4 | 0.158 |
| 4 | special input devices | 3 | 0.105 |
| 5 | screens affected by font adjustments | 4 | 0.263 |

## *3.4.4 Robustness Metrics*

In this section, metrics collected from the imaging application are used to demonstrate how to obtain the vulnerable points for the robustness objective. Table 5 was constructed by implementing Algorithm 1 to secure the proper weights for each attribute. After applying Definition 4, the predicted risk value for the robustness objective 2.514. The calculations are as follows:

$r = 3(.135) + 2(.152) + 4(.152) + 2(.091) + 2(.091) + 1(.121) + 2(.061) + 1(.121) + 3(.091) + 3(.061)$
$r = .405 + .162 + .541 + .054 + .162 + .027 + .216 + .135 + .405 + .405$
***$r = 2.514$***

---

**Definition 4**: Metric for predicting the vulnerable points of robustness is a function of:
$r = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7 + w_8x_8 + w_9x_9 + w_{10}x_{10}$

where $x_1$ is the number of lines of source code;
$x_2$ is the number of threads generated;
$x_3$ is the number of changes to dynamic data;
$x_4$ is the number of class inheritances;
$x_5$ is the number multitask command buttons;
$x_6$ is the number of program switches lacking a default clause;
$x_7$ is the number of use case totals;
$x_8$ is the number of unrestricted text fields;
$x_9$ is the number of user interface complexity;
$x_{10}$ is the number of user levels;
and $w_1$, $w_2$, $w_3$,…. and $w_{10}$ are their weights for relative importance.

---

Table 5. Robustness Metrics Data Table

| Robustness | | | |
|---|---|---|---|
| i | Attributes | Normalized Attribute Value | Weight |
| | | x | w |
| 1 | number of lines of source code | 1 | 0.082 |
| 2 | amount of threads generated | 4 | 0.102 |
| 3 | changes to dynamic data | 3 | 0.082 |
| 4 | class inheritances | 3 | 0.102 |

| Robustness | | | |
|---|---|---|---|
| i | Attributes | Normalized Attribute Value | Weight |
| | | x | w |
| 5 | multitask command buttons | 2 | 0.082 |
| 6 | program switches lacking default clause | 3 | 0.061 |
| 7 | use case totals | 3 | 0.061 |
| 8 | unrestricted text fields | 3 | 0.102 |
| 9 | user interface complexity | 3 | 0.082 |
| 10 | user-levels | 3 | 0.102 |

### *3.4.5 Software Vulnerability Metrics*

The check processing system is used as an example to demonstrate the metric used to predicted risk value for software vulnerabilities. In this example, this measure would be extremely important to this type of application that handles financial data. The attribute values outlined in Table 6 are employed by the metric described in Definition 5 to calculate which is 2.941

$s = 3(.294) + 3(.235) + 3(.118) + 3(.294) + 2(.059)$
$s = .882 + .706 + .353 + .882 + .118$
$s = \boldsymbol{2.941}$

**Definition 5**: Metric for predicting the vulnerable points of software vulnerability is a function of:

$s(x_1, x_2, x_3, x_4, x_5) = w_1x_1 + w_2x_2 + w_3x_3 + w_4 x_4 + w_5x_5$
where $x_1$ is the number of firewall ports required;
$x_2$ is the number of network connections required;
$x_3$ is the number of open API;
$x_4$ is the number of special user accounts required;
$x_5$ is the number of web portals required;
and $w_1$, $w_2$, $w_3$,$w_4$, and $w_5$ are their weights for relative importance.

Table 6. Software Vulnerability Metrics Data Table

| Software Vulnerability | | | |
|---|---|---|---|
| i | Attributes | Normalized Attribute Value | Weight |
| | | x | w |
| 1 | firewall port required | 5 | 0.294 |
| 2 | network connections required | 4 | 0.235 |
| 3 | open API | 2 | 0.118 |
| 4 | special user accounts required | 5 | 0.294 |
| 5 | web portals required | 1 | 0.059 |

### *3.4.6 UI Accessibility Metrics*

UI Accessibility is another beta testing objective that is limited in the number of attributes (limited to just two). Its limitation is because the operating environment manages most accessibility features. However, to provide an example of how applications designed for end-user with physical challenges variables are utilized from the imaging application since the imaging application was designed to support a large base of end-users with disabilities. Using the attributes in Table 7, definition provides a vulnerable point for UI accessibility as 2.1 indicating the potential of a moderate problem.

*$a = 1(.4) + 3(.6)$*
*$a = 4.2$*
*$a =$ **2.200***

---

**Definition 6**: Metric for predicting the vulnerable points of UI Accessibility is a function of:
$a = w_1x_1 + w_2x_2$
where $x_1$ is the number of fonts effect on UI;
$x_2$ is the number of languages supported;
and $w_1$ and $w_2$ are their weights for relative importance.

---

Table 7. UI Accessibility Metrics Data Table

| UI Accessibility | | | |
|---|---|---|---|
| i | Attributes | Normalized Attribute Value | Weight |
| | | x | w |
| 1 | special fonts | 1 | 0.400 |
| 2 | special hardware requirements | 3 | 0.600 |

### *3.4.7 UI Usability Metrics*

In this section, Definition 7 is employed to calculate the vulnerable points for the check processing application. The object is to predict the vulnerable point to assist software developers in predicting the impact of usability prior to beta testing. Here values from Table 8 are utilized to obtain the predicted risk value of UI usability as 3.08.

$u = 4(.208) + 2(.125) + 5(.208) + 3(.208) + 1(.125) + 2(.083) + 1(.042)$
$u = .833 + .250 + 1.042 + .625 + .125 + .167 + .042$
***u = 3.084***

**Definition 7**: Metric for predicting the vulnerable points of user interface usability is a function of:
$u = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7$

where $x_1$ is the number of message boxes;
$x_2$ is the number of multitask command buttons;
$x_3$ is the number of screen traversal;
$x_4$ is the number of software wizards;
$x_5$ is the number of UI responses required;
$x_6$ is the number of user interface controls;
$x_7$ is the number of unrestricted text fields;
and $w_1$, $w_2$, $w_3$, $w_4$, $w_5$, $w_6$, and $w_7$ are their weights for relative importance.

Table 8. UI Usability Metrics Data Table

| UI Usability | | | |
|---|---|---|---|
| i | Attributes | Normalized Attribute Value | Weight |
| | | x | w |
| 1 | message boxes | 4 | 0.208 |
| 2 | multitask command buttons | 2 | 0.125 |
| 3 | screen traversal | 5 | 0.208 |
| 4 | software wizards | 3 | 0.208 |
| 5 | UI responses required | 1 | 0.125 |
| 6 | user interface controls[1] | 2 | 0.083 |
| 7 | unrestricted text fields | 1 | 0.042 |

## 3.5 Metrics Function Training for Software Attributes

The essence of beta testing design is the utilization of software-metrics to measure the vulnerability of objectives. The objectives are impacted by attributes, which are those essential parts of an application that influence a specific beta testing objective. The attributes have quantitative values (weights) used to predict the level of complexity for a particular product. The outcome provides software manufacturers with a baseline of specific areas requiring the most focus during a beta test.

This section outlines and provides the appropriate steps required to assign suitable weights to attributes (metrics). The overall objective is for the weights to mature through a learning process, which forms by the accumulation of beta testing feedback for products in a similar domain. However, the methods provided in this study will be suitable for common enterprise software.

Software metrics for beta testing design are the measurements of software properties to formulate a test plan before the product enters the beta testing phase. The metrics results predict areas of vulnerabilities (risk) used to prioritize testing criteria such as required skill and size of beta testers, assistance in developing testing questionnaire, time required to test product; and how to manage testing feedback. This section demonstrates how to obtain good software metrics and the application of the metric functions.

The model in this study was influenced by trainable pattern classifiers originally introduced by Nils J. Nilsson in Learning Machines: Foundations of trainable pattern-classifying systems published in 1965. Nilsson believed machines learn by past experience. His trainable pattern classifiers were instrumental in predicting chance of rain based on the categorizing of a set outcome [from 1 to 3]. His model used a set of attributes in a pattern, and patterns to be categorized into pattern classifiers to produce an outcome[38]. The objective value functions in this research are a component of Nilsson's *linear discriminant function* that classifies values of parameters into families (a set).

The essence of beta testing design is the understanding of testing objective risk level prior to preparing a product for beta testing. Risk value is best assessed through application attributes, which determines where potential issues lie within an enterprise product. In this study, risk is determined by a value from 1 to 5 where 1 represents a low level of potential problems during production and 5 denotes the potential for strong problems. The process assists in software process improvement, which influences software quality.

A detailed set of steps creates a structure process for obtaining good software metrics. The steps are collecting attribute values, normalizing attribute values, weights initialization (required for initial usage of software based metrics), and using objective functions to calculate predicted risk levels. After the project has completed the software metrics process and beta testing is complete, the actual values are collected and the metric training process is employed on future projects to receive better predictions (See Figure 5).

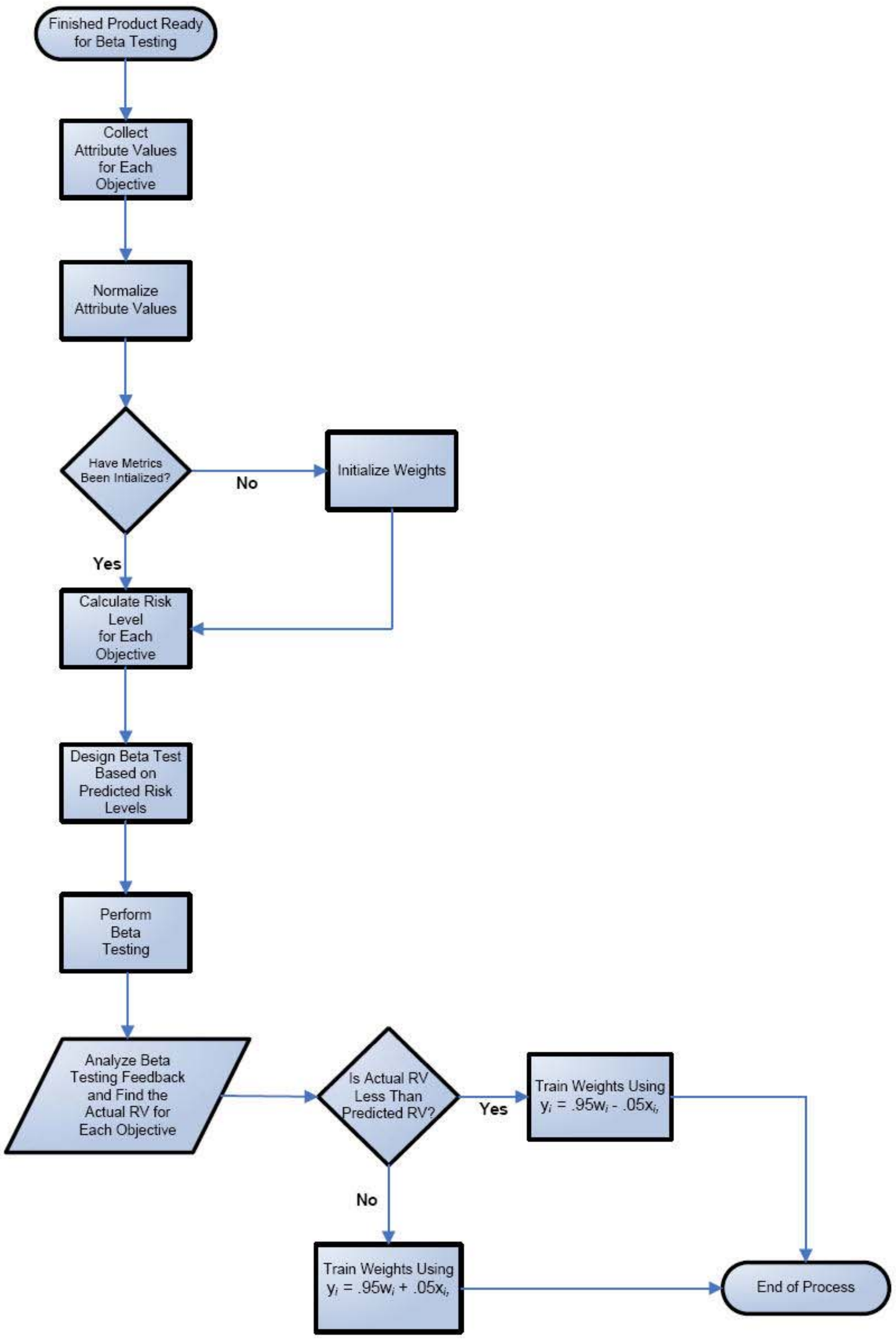
Finished Product Ready for Beta Testing
Collect Attribute Values for Each Objective
Normalize Attribute Values
Have Metrics Been Intialized?
No
Initialize Weights
Yes
Calculate Risk Level for Each Objective
Design Beta Test Based on Predicted Risk Levels
Perform Beta Testing
Analyze Beta Testing Feedback and Find the Actual RV for Each Objective
Is Actual RV Less Than Predicted RV?
Yes
Train Weights Using $y_i$ = .95$w_i$ - .05$x_i$,
No
Train Weights Using $y_i$ = .95$w_i$ + .05$x_i$,
End of Process

**Figure 5. Software Metrics Process Flow**

### *3.5.1 Collecting Actual Attribute Values*

Beta testing design starts with a software project that has finished the development process and is then  ready for beta testing.  Prior to starting the beta test, *attribute values* are collected in each objective.  An example is the attribute value for number of source lines of code = 4 million or the number of firewall ports = 7.  The complete list of attributes is outlined in Section 3.3 Major Categorized Software Attributes for Beta Testing.  After each attribute value is recorded, the values will require normalization.

### *3.5.2 Normalizing Attribute Values*

Normalization is a process used to bring an attribute value within the range of 1 to 5.  This process improves the accuracy of the predicted function value revealing the level of risk associated with an objective.  Normalization is achieved in the understanding of minimum and maximum values for a specific attribute, making normalization unique for each attribute.

Minimum and maximum values for attributes are based on past software projects and may vary by software and software domain.  In this study, the attribute minimum and maximum values were determined from past projects using the statistical mode.  For example, a sample of minimum and maximum attributes was collected for an enterprise application for the group of attributes in the Software Vulnerability objectives.  After reviewing the attribute values from past software projects it was determined that the application (and like applications) require access to between 0 and 20 firewall ports (using the statistical mode).  In the project, the actual attribute value for number of firewall ports is 7. The attribute value must be normalized to fit within the range of 1 to 5. In this case, the minimum and maximum function is used to scale the value between the target range. The minimum and maximum function is:

$$normalized\_attribute\_value = \left( \frac{a_i - a_{min}}{a_{max} - a_{min}} \right) \times \left( r_{max} - r_{min} \right) + r_{min}$$

Where $a_i$ is the non-normalized attribute, $a_{min}$ is minimum value of the attributes range, $a_{max}$ is the maximum value of the attribute range, $r_{min}$ is the minimum value of the new ranges (desired) range, and $r_{max}$ is the maximum value of the new range. The normalized attribute value is rounded up to the nearest whole number. Using this formula, the attribute value of 7 is normalized to *2*. The normalization process is not project, but attribute specific. Minimum and maximum values improve over time and are adjusted from a case-by-case objective.

After all attributes are standardized to obtain normalized attributes values, the weights are required to measure the level of relative importance to the objective. If the software metrics function is utilized for the first time, weights initialization is required. Software that has utilized the software metrics process for past projects, applies the learning process to improve weights.

### *3.5.3 Initializing Attribute Weights*

The initialization of attribute weights is determined by data from past beta tests. Since the applications are experienced products, past data is used to determine attributes weights using empirical knowledge. Weights must be expressed by real numbers with all attribute weights totaling *1* for each objective. A more structural approach to obtain real number values for each objective is to assign estimated weights, based on numeric value from 1 to 5. The value range is based on the level of problem relevant to the attribute (i.e. The number 1 represents low problem and number 5 is a more problematic attribute). After the estimated weights are assigned (within range), the values are converted to weights (real numbers) using a weight initialization function. The total of all weights must equal 1. The weight initialization function is:

$$w_i = \frac{e_i}{\left(\sum_{i=1}^{n} e_i\right)}$$

where $w_i$ is the weight, $e_i$ is the estimated weight. This process is not required for future projects, because the training process will be employed to manipulate weights.

### *3.5.4 Training Future Functions*

Software metrics based approach to beta testing design is a process that learns with experience and metric functions improve in accuracy with frequent utilization. The learning process is employed on future beta testing products by reviewing results and feedback from past projects to determine an actual risk value. The actual risk value is compared to the predicted risk value from previous tests to incorporate adjustments to the training process. The purpose of this method is to improve the risk predictions.

The training process introduces a learning sub-model that takes into account the attribute weights from previous projects and the normalized attribute value of the current project to generate a new weight. The new weight is then applied to the objective metric function to calculate the predicted risk value. There are two different formulas in the learning process. To determine the correct formula, a comparison of the actual and predicted risk value is required.

$$y_i = r \bullet w_i \pm (1 - r)(\frac{x_i}{5})$$

The two formulas are:

Formula A) $y_i = r \bullet w_i + (1 - r)(\frac{x_i}{5})$ where $y_i$ is the new attribute weight, r is a real number between 0 and 1, $w_i$ is the weight from previous beta test, and $x_i$ is the normalized attribute value for the current project.

Formula B) $y_i = r \bullet w_i - (1 - r)(\frac{x_i}{5})$ where $y_i$ is the new attribute weight, r is a real number between 0 and 1, $w_i$ is the weight from previous beta test, and $x_i$ is the normalized attribute value for the current project.

Formula A. is applied when the predicted risk value is lower than the actual risk value. For example, when a product is ready for beta testing, objective metrics are used to obtain the predicted risk values for each objective. The values are used to build the current beta test design. After completion of the project, the results and feedback is used to determine the actual risk value. In this scenario if the predicted risk value for Robustness was 2 and the actual risk value was 3, formula a. $y_i = r \bullet w_i + (1 - r)(\frac{x_i}{5})$ , is used to train the weights for the next beta test of this product. If the predicted metric value for enterprise dependency was 4 and the actual metric value is 2, then Formula B. $y_i = r \bullet w_i - (1 - r)(\frac{x_i}{5})$ , is used to train weights for the next beta test.

In this study, r = .95 because the value represents the standard confidence coefficient. This means the current sample is expected to contain the true mean and computed confidence interval is 95%[46]. However, the actual percentage values in each formula may be adjusted to match the proportion of past data. After training the weights, the values are applied to the objective metric functions to predict risk levels.

If the predicted risk value and actual risk value are equal, the learning process is not required. This will hold true case until a change to the actual risk value is prompted by a change in the product.

### *3.5.5 Objective Metric Functions*

After weights are initialized or trained, the objective value function is employed to calculate the predicted risk value. The predicted risk value is a number between 1 and 5 that exposes areas of weaknesses in the product. The prediction provides guidance in how to prepare the current beta testing project. The predicted risk value numbers are assigned a level, which are:

- 1 = Low Problem
- 2 = Moderate Problem
- 3 = Average Problem
- 4 = Significant Problem
- 5 = Strong Problem

To calculate the objective metric risk value for a specific objective the following formula is used:

$$objective\_metric\_value = \sum_{i=1}^{n} w_i x_i$$

where *i* is an instance of an attribute, $w_i$ is the weight of an attribute, $x_i$ is the normalized attribute value. The objective metric value is calculated for all 7 objectives of an enterprise product.

## 3.6 Example Practice of Objective Metric Function

In this section, an example from a medium-sized enterprise product is used to demonstrate how the objective metric function is used to predict the risk value for the software vulnerability objective. Three algorithms are used to calculate the predicted risk value for each objective. The major steps in calculating the predicted risk value for an objective are as follows:

1. a completed enterprise that has been beta tested in the past;
2. the attributes' values are determined;

3. the minimum and maximum values are determined based on historical data (this step is only performed once);
4. actual attribute values are converted to normalized attribute values;
5. weights are initialized for product that have not used the software metrics process in the past (only required one time);
6. the objective metric function is used to calculate the predicted risk value;
7. and the training formula is used after actual risk value is determined (actual risk value is determined after the current beta test and impacted from past projects).

Three algorithms are provided in this study to calculate the predicted risk values for each objective.

The first algorithm (Algorithm 1. Weights Initialization) is used for projects that have completed a non-software metrics-based beta test. These products have completed other types of software beta test but are using the software metric approach to beta testing design for the first time. Since products in this category have no prior experience with this process, weights will require initialization.

**Algorithm 1. Weights Initialization**

1) Completed enterprise software that has accomplished a beta test in the past
2) Provide attribute values for each attribute for all 7 objectives ($a_i$)
3) Review the Min/Max values for each attribute
4) Normalize all attribute values to a range between 1 and 5

$$x_i = \left( \frac{a_i - a_{min}}{a_{max} - a_{min}} \right) \times \left( r_{max} - r_{min} \right) + r_{min}$$

where $x_i$ is the normalized value, $a_i$ is the actual attribute value, $a_{min}$ is minimum value of the attributes range, $a_{max}$ is the maximum value of the attribute range, $r_{min}$ is the minimum value of the new ranges (desired) range, and $r_{max}$ is the maximum value of the new range.

5) Initialize weights for current project

$$w_i = \frac{e_i}{\left( \sum_{i=1}^{n} e_i \right)}$$

where $w_i$ is the weight of relative importance, $e_i$ is the estimated weight where $i$ is an instance of an attribute, $w_i$ is the weight of an attribute, $x_i$ is the normalized attribute value

6) Calculate the predicted objective risk value for each objective

$$risk\_value = \sum_{i=1}^{n} w_i x_i$$

where $i$ is an instance of an attribute, $w_i$ is the weight of an attribute, $x_i$ is the normalized attribute value

The second algorithm (Algorithm 2. Function Learning Process) is used for software that has completed the software metrics-based process. The function learning process is used to train the metric functions to improve risk predictions, after the actual risk values are determined at the end of a complete beta test project.

**Algorithm 2. Function Learning Process**

1) Compare the actual metric risk value from each objective to the predict risk value from previous beta test. If predicted risk value > actual risk value goto (a). If predicted risk value < actual risk value goto (b). if the predicted risk value = risk value then skip to 3)

$y_i = r \bullet w_i - (1-r)(\frac{x_i}{5})$ where $y_i$ is the new attribute weight, r is a real number between 0 and 1, $w_i$ is the weight from previous beta test, and $x_i$ is the normalized attribute value for current project.

(a) $y_i = r \bullet w_i + (1-r)(\frac{x_i}{5})$ where $y_i$ is the new attribute weight, r is a real number between 0 and 1, $w_i$ is the weight from previous beta test, and $x_i$ is the normalized attribute value for current project.

2) If total weights ≠ 1 then scale weights using formula

$$w_i = \frac{y_i}{\left(\frac{\sum_{i=1}^{n} y_i}{n}\right)}$$

where $y_i$ is the trained weight for the current project, n =1

3) goto Algorithm 3

The third algorithm (Algorithm 3. Standard Objective Risk Level Function) is used when product has experience using the metric functions and weights initialization is not required.

**Algorithm 3. Standard Objective Risk Level Function**

1) Completed enterprise software that is ready for beta testing.
2) Provide attribute values for each attribute in all 7 objectives ($a_i$).
3) review the Min/Max values for each attribute
4) Normalize all attribute values between 1 and 5

$$x_i = \left(\frac{a_i - a_{min}}{a_{max} - a_{min}}\right) \times \left(r_{max} - r_{min}\right) + r_{min}$$

where $x_i$ is the normalized value, $a_i$ is the actual attribute value, $a_{min}$ is minimum value of the attributes range, $a_{max}$ is the maximum value of the attribute range, $r_{min}$ is the minimum value of the new ranges (desired) range, and $r_{max}$ is the maximum value of the new range.

5) Calculate the predicted objective risk value for each objective

$$risk\_value = \sum_{i=1}^{n} w_i x_i$$

where *i* is an instance of an attribute, $w_i$ is the weight of an attribute, $x_i$ is the normalized attribute value

### *3.6.1 Weights Initialization Process*

In this section, Algorithm 1 is used to calculate the predicted risk value for the software vulnerability objective.

**Step 1.** The completed enterprise product is a medium-sized enterprise product that has completed a beta test in the past 18 months. The data from the past test is used to determine the weights and instrumental in normalizing the values.

**Step 2.** The attribute values for this objective are provided by development for the current application and are later normalized to a range of 1 to 5. Outlined in Table 1., the non-normalized attribute values (*a. attribute*) are:

number of firewall ports = 5
number of network connections = 4
number of open API = 3
number of special user account = 5
number of web portals = 6

Although the values (as is) appear to be within the correct range, the normalization process will change the value based on the minimum and maximum attribute range for the project.

**Step 3.** Normalization is the process of converting the actual attribute value into a range between 1 and 5, used to better assess risk. However, it is important to understand the minimum and maximum values for attributes, which differs by application domain. In this scenario, the min/max values are determined by taking the statistical mode of past projects and using the min/max function to construct a normalization matrix for determining the normalized value of an attribute.

| | Min | Max |
|---|---|---|
| firewall port | 0 | 20 |
| network connections | 1 | 7 |
| open API | 0 | 9 |
| special user accounts | 0 | 7 |
| web portals | 0 | 12 |

**Step 4.** In Step 4, the values are applied to the min/max function to convert actual attribute value to normalized attribute value. The result is rounded up to the nearest whole number between 1 and 5. Using the min/max function, the normalization calculations are:

*firewall port*

$$x_1=\left(\frac{5-0}{20-0}\right)\times(5-1)+1$$
$$x_1=\left(\frac{5}{20}\right)\times(4)+1$$
$$x_1=(.25)\times(5)$$
$$x_1=1.25$$
$$x_1 = \boldsymbol{2}$$

*network connections*

$$x_2=\left(\frac{4-1}{7-1}\right)\times(5-1)+1$$
$$x_2=\left(\frac{3}{6}\right)\times(4)+1$$
$$x_2=(.5)\times(5)$$
$$x_2=2.5$$
$$x_2 = \boldsymbol{3}$$

*open API*

$$x_3=\left(\frac{3-0}{9-0}\right)\times(5-1)+1$$
$$x_3=\left(\frac{3}{9}\right)\times(4)+1$$
$$x_3=(.333)\times(5)$$
$$x_3=1.667$$
$$x_3 = \boldsymbol{2}$$

*special user accounts*

$$x_4=\left(\frac{5-0}{7-0}\right)\times(5-1)+1$$
$$x_4=\left(\frac{5}{7}\right)\times(4)+1$$
$$x_4=(.714)\times(5)$$
$$x_4=3.57$$
$$x_4 = \boldsymbol{4}$$

*web portals*

$$x_5=\left(\frac{6-0}{12-0}\right)\times(5-1)+1$$
$$x_5=\left(\frac{6}{12}\right)\times(4)+1$$
$$x_5=(.5)\times(5)$$
$$x_5=2.5$$
$$x_5 = \boldsymbol{3}$$

The normalized values will be multiplied by the attribute weight to determine the predicted risk value.

**Step 5.** The attribute weights are determined using a sub-model that is based on product experience. In this scenario, the target product has completed a beta test and weights require initialization because the product is using the software metrics-based process for the first time. Based on experience weights are determined and must be real numbers with all values equaling 1 for the objective. However, a more structured method is employed to initialize weights for the first time. The weight initialization formula is calculated using the formula:

$$w_i = \frac{e_i}{\left(\sum_{i=1}^{n} e_i\right)}$$

where $e_i$ is a number from 1 to 5, and $e_i$ is the estimated weight based on previous projects. The number 1 is used for attributes with least problems and 5 are more problematic attributes. The values are converted to real numbers for usage with the objective function. For example, based on experience the attributes for the software vulnerability are:

| | Estimate Weights |
|---|---|
| number of Firewall ports | 5 |
| number of network connections | 4 |
| number of open API | 3 |
| number of special user account | 4 |
| number of special users accounts | 5 |

The estimated weights are converted to real numbers totaling 1. The calculations used to convert estimated weights to proper weights (real numbers) are as follows:

$$w_i = \frac{e_i}{\left(\sum_{i=1}^{n} e_i\right)}$$

$$w_i = \frac{e_i}{\left(e_1 + e_2 + e_3 + e_4 + e_5\right)}$$

*firewall port*

$$w_1 = \frac{e_1}{(e_1 + e_2 + e_3 + e_4 + e_5)}$$
$$w_1 = \frac{5}{(5+4+3+4+5)}$$
$$w_1 = \frac{5}{(21)}$$
$$w_1 = .238$$

*network connections*

$$w_2 = \frac{e_2}{(e_1 + e_2 + e_3 + e_4 + e_5)}$$
$$w_2 = \frac{4}{(5+4+3+4+5)}$$
$$w_2 = \frac{4}{(21)}$$
$$w_2 = .190$$

*open API*

$$w_3 = \frac{e_3}{(e_1 + e_2 + e_3 + e_4 + e_5)}$$
$$w_3 = \frac{3}{(5+4+3+4+5)}$$
$$w_3 = \frac{3}{(21)}$$
$$w_3 = .143$$

*special user accounts*

$$w_4 = \frac{e_4}{(e_1 + e_2 + e_3 + e_4 + e_5)}$$
$$w_4 = \frac{4}{(5+4+3+4+5)}$$
$$w_4 = \frac{4}{(21)}$$
$$w_4 = .190$$

*web portals*

$$w_5 = \frac{e_5}{(e_1 + e_2 + e_3 + e_4 + e_5)}$$
$$w_5 = \frac{5}{(5+4+3+4+5)}$$
$$w_5 = \frac{5}{(21)}$$
$$w_5 = .238$$

**Step 6.** After the attributes are normalized and weights are determined, the objective metric function is employed to calculate risk. Objective function for the software vulnerability objective is:

$$objective\_metric\_value = \sum_{i=1}^{n} w_i x_i$$

For the software vulnerability the objective metric function is:

In this case, $\mathbf{s = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5}$
*where s is the software vulnerability objective, $x_1$ is the normalized number of threads generated; $x_2$ is the normalized number of network connections required; $x_3$ is the normalized number of open API; $x_4$ is the normalized number of special user accounts required; $x_5$ is the normalized number of web portals required; and $w_1$, $w_2$, $w_3$, $w_4$, and $w_5$ are their weights for relative importance.*

The values are calculated as:
$s$ = .238(2) + .190(3) +.143(2) +.190(4) +.238(3)
$s$ = .476 + .571 +.286 +.762 +.714
$s$ = **2.810**

which provide a predicted risk value for software vulnerability as 2.81. The predicted value is measured against other values to determine priority of events with designing the beta test. After the completion of the beta test, the predicted value is compared to the actual risk value for training future tests.

Table 9. Software Vulnerability Weights Initialization Data

| Software Vulnerability | | | | | |
|---|---|---|---|---|---|
| | Attribute Name | I. Weight | A. Value | N. Value | Total |
| i | | $w_i$ | $a_i$ | $x_i$ | |
| 1 | firewall port | 0.238 | 5 | 2 | 0.476 |
| 2 | network connections | 0.190 | 4 | 3 | 0.571 |
| 3 | open API | 0.143 | 3 | 2 | 0.286 |
| 4 | special user accounts | 0.190 | 5 | 4 | 0.762 |
| 5 | web portals | 0.238 | 6 | 3 | 0.714 |
| | Totals | **1.000** | | | **2.810** |

### *3.6.2 Weights Stabilization Process*

As this software metrics based process matures, the weights are stabilized and risk value prediction becomes more streamlined. This is done by using the function learning process, which adjusts the weights for the new project by incorporating older weights with new values to determine the importance of attributes. This is achieved after products have completed a few beta tests using the software metrics-based approach.

Using the metrics from the previous case and applying Algorithm, the above test has completed a previous beta test and the actual risk value for the software vulnerability objective is 3. The predicted risk value from the previous test was 2.810. The weights for the current project will be trained to receive a better predicted risk value for use in the beta testing design. The training process begins after completing steps 1 – 4 (which mirrors activities from Step 1, 2, 3, and 4 from the previous section).

**New Step 5**. The new Step 5. weights stabilization (for Project 2)uses a different set of functions to train weights for use with the current project. Since the actual predicted value from the previous project ( 2.810) is lower than the actual risk value (3), which was determined at the end of the previous project. In this scenario the correct formula is:

Based on the learning function formula $y_i = r \bullet w_i \pm (1 - r)(\frac{x_i}{5})$ the actual risk value is greater than the predicted risk value. So the correct formula used to train the weights is $y_i = r.w_i + (1 – r) (x_i / 5)$, where $y_i$ is the new attribute weight, r is .95, $w_i$ is the weight from past beta test, and $x_i$ is the normalized attribute value for current project.

*firewall port*

$$y_1 = .95(.238) + (.05)(\frac{2}{5})$$
$$y_1 = .95(.238) + .05(.4)$$
$$y_1 = .226 + .02$$
$$y_1 = .246$$

*network connections*

$$y_2 = .95(.238) + (.05)(\frac{3}{5})$$
$$y_2 = .95(.190) + .05(.6)$$
$$y_2 = .181 + .03$$
$$y_2 = .211$$

*open API*

$$y_3 = .95(.238) + (.05)(\frac{2}{5})$$
$$y_3 = .95(.143) + .05(.4)$$
$$y_3 = .136 + .02$$
$$y_3 = .156$$

*special user accounts*

$$y_4 = .95(.238) + (.05)(\frac{4}{5})$$
$$y_4 = .95(.190) + .05(.8)$$
$$y_4 = .181 + .04$$
$$y_4 = .221$$

*web portals*

$$y_5 = .95(.238) + (.05)(\frac{3}{5})$$
$$y_5 = .95(.238) + .05(.6)$$
$$y_5 = .226 + .03$$
$$y_5 = .256$$

After calculating the trained weights, the sum must equal 1. If value is higher or lower than 1, $y_i$ must be scaled to equal 1 (see A. Value in Table 3). To scale the weights to 1, the weights adjustment formula (similar to initial weights function) is used to scale up or down the weights.

$$w_i = \frac{y_i}{\left(\frac{\sum_{i=1}^{n} y_i}{n}\right)}$$

$$w_i = \frac{y_i}{(y_1 + y_2 + y_3 + y_4 + y_5)}$$

*firewall port*

$$w_1 = \frac{y_1}{(y_1 + y_2 + y_3 + y_4 + y_5)}$$
$$w_1 = \frac{.246}{(.246 + .211 + .156 + .221 + .256)}$$
$$w_1 = \frac{.246}{(1.090)}$$
$$w_1 = .226$$

*network connections*

$$w_2 = \frac{y_2}{(y_1 + y_2 + y_3 + y_4 + y_5)}$$
$$w_2 = \frac{.211}{(.246 + .211 + .156 + .221 + .256)}$$
$$w_2 = \frac{.211}{(1.090)}$$
$$w_1 = .194$$

*open API*

$$w_3 = \frac{y_3}{(y_1 + y_2 + y_3 + y_4 + y_5)}$$
$$w_3 = \frac{.156}{(.246 + .211 + .156 + .221 + .256)}$$
$$w_3 = \frac{.156}{(1.090)}$$
$$w_3 = .143$$

*special user accounts*

$$w_4 = \frac{y_4}{(y_1 + y_2 + y_3 + y_4 + y_5)}$$
$$w_4 = \frac{.221}{(.246 + .211 + .156 + .221 + .256)}$$
$$w_4 = \frac{.221}{(1.090)}$$
$$w_4 = .203$$

*web portals*

$$w_5 = \frac{y_5}{(y_1 + y_2 + y_3 + y_4 + y_5)}$$
$$w_5 = \frac{.256}{(.246 + .211 + .156 + .221 + .256)}$$
$$w_5 = \frac{.256}{(1.090)}$$
$$w_5 = .235$$

After the weights are scaled, they are applied to the objective value function to obtain the predicted risk value for the current project.

**New Step 6.** The objective risk value function for the software vulnerability objective uses the function:

$$\mathbf{s = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5}$$
$$s = .226(2) + .194(3) + .143(2) + .203(4) + .235(3)$$
$$s = .452 + .581 + .286 + .811 + .705$$
$$s = 2.834$$

The predicted risk value is *2.834* after applying the objective risk value function for software vulnerability, which is close to the actual metric value (*3*) (See Result in Table 10). In Table 11 the learning process was executed 6 times (six experiments) to demonstrate how the weights are stabilized and the prediction is improved. After five applications the predicted risk value improved to 2.905, which is .095 from the actual risk value of 3.

Table 10. Software Vulnerability Function Training Data

| **Software Vulnerability** | | | | | |
|---|---|---|---|---|---|
| | **Attribute Name** | **N. Weight** | **A. Value** | **N. Value** | **Total** |
| i | | $y_i$ | $w_i$ | $x_i$ | |
| 1 | firewall port | 0.246 | 0.226 | 2 | 0.452 |
| 2 | network connections | 0.211 | 0.194 | 3 | 0.581 |
| 3 | open API | 0.156 | 0.143 | 2 | 0.286 |
| 4 | special user accounts | 0.221 | 0.203 | 4 | 0.811 |
| 5 | web portals | 0.256 | 0.235 | 3 | 0.705 |
| | Totals | *1.090* | *1.000* | | **2.834** |

## Table 11. Functions Stabilization Data Set

| **Software Vulnerability** | | | | | | **Experiment 1** | | | | **Experiment 2** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Attribute Name** | **Weight** | **I. Weight** | **A.Value** | **N. Value** | **Total** | **L Weight** | **A Weight** | **Value** | **Total** | **L Weight** | **A Weight** | **Value** | **Total** |
| firewall port required | 5 | 0.238 | 5 | 2 | 0.476 | 0.246 | 0.226 | 2 | 0.452 | 0.254 | 0.216 | 2 | 0.432 |
| network connections required | 4 | 0.190 | 4 | 3 | 0.571 | 0.211 | 0.194 | 3 | 0.581 | 0.230 | 0.196 | 3 | 0.588 |
| open API | 3 | 0.143 | 3 | 2 | 0.286 | 0.156 | 0.143 | 2 | 0.286 | 0.168 | 0.143 | 2 | 0.286 |
| special user accounts required | 4 | 0.190 | 5 | 4 | 0.762 | 0.221 | 0.203 | 4 | 0.811 | 0.250 | 0.213 | 4 | 0.850 |
| web portals required | 5 | 0.238 | 6 | 3 | 0.714 | 0.256 | 0.235 | 3 | 0.705 | 0.273 | 0.233 | 3 | 0.698 |
| | *21* | *1.000* | | | **2.810** | *1.090* | *1.000* | | **2.834** | *1.176* | *1.000* | | **2.854** |

| **Experiment 3** | | | | | **Experiment 4** | | | | **Experiment 5** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| firewall port required | **L Weight** | **A Weight** | **Value** | **Total** | **L Weight** | **A Weight** | **Value** | **Total** | **L Weight** | **A Weight** | **Value** | **Total** |
| network connections required | 0.261 | 0.208 | 2 | 0.416 | 0.268 | 0.201 | 2 | 0.402 | 0.275 | 0.195 | 2 | 0.390 |
| open API | 0.249 | 0.198 | 3 | 0.594 | 0.266 | 0.200 | 3 | 0.599 | 0.283 | 0.201 | 3 | 0.604 |
| special user accounts required | 0.180 | 0.143 | 2 | 0.286 | 0.191 | 0.143 | 2 | 0.286 | 0.201 | 0.143 | 2 | 0.286 |
| web portals required | 0.277 | 0.221 | 4 | 0.883 | 0.304 | 0.228 | 4 | 0.910 | 0.328 | 0.233 | 4 | 0.933 |
| | 0.290 | 0.231 | 3 | 0.692 | 0.305 | 0.229 | 3 | 0.686 | 0.320 | 0.227 | 3 | 0.682 |
| | *1.257* | *1.000* | | **2.870** | *1.334* | *1.000* | | **2.884** | *1.407* | *1.000* | | **2.895** |

| **Experiment 6** | | | | |
|---|---|---|---|---|
| firewall port required | **L Weight** | **A Weight** | **Value** | **Total** |
| network connections required | 0.281 | 0.190 | 2 | 0.381 |
| open API | 0.299 | 0.202 | 3 | 0.607 |
| special user accounts required | 0.211 | 0.143 | 2 | 0.286 |
| web portals required | 0.352 | 0.238 | 4 | 0.953 |
| | 0.334 | 0.226 | 3 | 0.678 |
| | *1.477* | *1.000* | | **2.905** |

## 3.7 Example of Deriving Beta Testing Metrics

In this section, the metrics are utilized to demonstrate the application of the software metric functions on a real medium application. The purpose is to show how the metrics provide results for enterprise level software. The first experiment is on a medium size accounting application designed to support a major energy company. The software was designed to support end-to-end processing of lease and royalty distribution for land owners that have a contract with the energy company. The contract allows the energy company to drill oil and other natural resources from the land.

The software is designed to support the energy company's finance business unit. This form-intensive application facilitates the payment process handling calculations, currency transfers, balancing accounts, and other finance operations. The current application supports up to 50 consecutive end-users to include web-based users. This application was designed using a culmination of tools to include a major database application, multiple systems, and a web (based development tool) to support remote users. The data for this sample was collected from the project development manager who is also responsible for managing quality control efforts. The product has just completed a beta test for version 4.0 of this application. The new version supports more users and new database application.

### *3.7.1 Weights Initialization Process for Accounting Software*

This step in the process was for the purpose of collecting the attribute values for each objective (See Table 12). Reviewing the data from past beta tests the project development manager provided minimum and maximum values for each attribute. After the data was collected Algorithm 1 was employed because weights initialization is required. In Table 13 E. Weight represented the estimated weight based on experience with the product. The weights were initialized using step 5 in the Algorithm. N. Weight represents the weight for this project. After weights are initialized the objective function value was used to calculate the predicted risk level for each (Table 13). The predicted risk values are in bold.

Table 12. Accounting Software Actual Attribute Values

| **Actual Attribute Value** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **CI** | **CS** | **CL** | **CU** | **DC** | **DD** | **DL** | **FE** |
| 60 | 43 | 3 | 40 | 0 | 140 | 3 | 1 |
| **FP** | **HR** | **IC** | **ID** | **LS** | **MB** | **MC** | **MS** |
| 4 | 7 | 535 | 1 | 3 | 38 | 4 | 2 |
| **NC** | **OA** | **PP** | **SA** | **SC** | **SF** | **SH** | **ST** |
| 3 | 5 | 16 | 75 | 1200 | 1 | 1 | 37 |
| **SU** | **SW** | **TG** | **UC** | **UI** | **UL** | **UR** | **UT** |
| 3 | 2 | 55 | 10 | 535 | 4 | 15 | 120 |
| **WP** | **WS** | | | | | | |
| 3 | 3 | | | | | | |

Table 13. Accounting Application Prediction Data Set

| Environmental Dependency | | | | | |
|---|---|---|---|---|---|
| Attribute Name | Weight | N. Weight | Value | N. Value | Total |
| amount of threads generated | 5 | 0.116 | 55 | 1 | 0.116 |
| CPU utilization | 5 | 0.116 | 40 | 2 | 0.233 |
| dynamic link library requirements | 2 | 0.047 | 3 | 1 | 0.047 |
| Firewall port required. | 4 | 0.093 | 4 | 2 | 0.186 |
| incoming client service request | 4 | 0.093 | 43 | 3 | 0.279 |
| multiple system requirements | 2 | 0.047 | 2 | 1 | 0.047 |
| network connections required | 4 | 0.093 | 3 | 3 | 0.279 |
| open API | 4 | 0.093 | 5 | 3 | 0.279 |
| physical processors required | 5 | 0.116 | 16 | 3 | 0.349 |
| special user accounts required. | 1 | 0.023 | 3 | 3 | 0.070 |
| web portals required | 3 | 0.070 | 3 | 2 | 0.140 |
| web service request requirements | 4 | 0.093 | 3 | 1 | 0.093 |
| | 43 | 1.000 | | 25 | **2.116** |

| Function Coverage Completeness | | | | | |
|---|---|---|---|---|---|
| Attribute Name | Weight | N. Weight | Value | N. Value | Total |
| hardware and software requirements | 4 | 0.286 | 7 | 1 | 0.286 |
| software wizards | 3 | 0.214 | 2 | 1 | 0.214 |
| unrestricted text fields | 3 | 0.214 | 120 | 1 | 0.214 |
| use case totals | 4 | 0.286 | 10 | 2 | 0.571 |
| | 14 | 1.000 | | 5 | **1.286** |

| Localization | | | | | |
|---|---|---|---|---|---|
| Attribute Name | Weight | N. Weight | Value | N. Value | Total |
| code locale | 5 | 0.263 | 3 | 2 | 0.526 |
| fonts effect on UI | 3 | 0.158 | 1 | 1 | 0.158 |
| languages supported | 5 | 0.263 | 3 | 2 | 0.526 |
| special input devices | 2 | 0.105 | 1 | 1 | 0.105 |
| screens affected by font adjustments | 4 | 0.211 | 75 | 2 | 0.421 |
| | 19 | 1.000 | | 8 | **1.737** |

| Robustness | | | | | |
|---|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **Value** | **N. Value** | **Total** |
| number of lines of source code | 2 | 0.054 | 1200 | 1 | 0.054 |
| amount of threads generated | 3 | 0.081 | 16 | 1 | 0.081 |
| changes to dynamic data | 4 | 0.108 | 140 | 1 | 0.108 |
| class inheritances | 5 | 0.135 | 60 | 2 | 0.270 |
| multitask command buttons | 5 | 0.135 | 4 | 1 | 0.135 |
| program switches lacking default clause | 3 | 0.081 | 1 | 3 | 0.243 |
| use case totals | 3 | 0.081 | 10 | 2 | 0.162 |
| unrestricted text fields | 5 | 0.135 | 120 | 1 | 0.135 |
| user interface complexity | 4 | 0.108 | 535 | 3 | 0.324 |
| user-levels | 3 | 0.081 | 4 | 1 | 0.081 |
| | 37 | 1.000 | | 16 | **1.595** |

| Software Vulnerability | | | | | |
|---|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **Value** | **N. Value** | **Total** |
| Firewall port required | 5 | 0.294 | 4 | 2 | 0.588 |
| network connections required | 4 | 0.235 | 3 | 3 | 0.706 |
| open API | 2 | 0.118 | 5 | 3 | 0.353 |
| special user accounts required | 5 | 0.294 | 3 | 3 | 0.882 |
| web portals required | 1 | 0.059 | 3 | 2 | 0.118 |
| | 17 | 1.000 | | | **2.647** |

| UI Accessibility | | | | | |
|---|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **Value** | **N. Value** | **Total** |
| special fonts | 5 | 1.000 | | 1 | 1.000 |
| special hardware requirements | 0 | 0.000 | | 0 | 0.000 |
| | 5 | 1.000 | | | **1.000** |

| UI Usability | | | | | |
|---|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **Value** | **N. Value** | **Total** |
| message boxes | 3 | 0.136 | 38 | 1 | 0.136 |
| multitask command buttons | 3 | 0.136 | 4 | 1 | 0.136 |
| screen traversal | 5 | 0.227 | 15 | 5 | 1.136 |
| software wizards | 1 | 0.045 | 2 | 1 | 0.045 |
| UI responses required | 3 | 0.136 | 15 | 1 | 0.136 |
| User interface controls | 3 | 0.136 | 320 | 2 | 0.273 |
| unrestricted text fields | 4 | 0.182 | 120 | 1 | 0.182 |
| | 22 | 1.000 | | | **2.045** |

After applying the attributes values to the proper objective metrics, the risk levels points are calculated for comparison. The predicted risk values for each objective are: e = 2.116, f = 1.286, l = 1.737, r =1.595, s = 2.647, a = 1.000, u = 2.045, predicting a higher risk value for software vulnerability and enterprise dependency than any other objective.

The project manager provided a set of actual risk values based on feedback received from the current beta testing. The actual risk values are highlighted in Table 14. The project manager received favorable results from the group of beta testers only handling a small set of issues and a few usability changes.

**Table 14. Actual Risk Values vs. Predicted Risk Values for Accounting Application**

| | Environmental Dependency | Function Coverage Completeness | Localization | Robustness | Software Vulnerability | UI Accessibility | UI Usability |
|---|---|---|---|---|---|---|---|
| **PRV** | ***2.116*** | ***1.286*** | ***1.737*** | ***1.595*** | ***2.647*** | ***1*** | ***2.045*** |
| ARV | 3 | 3 | 2 | 2 | 3 | 1 | 3 |

### *3.7.2 Training Future Functions For Accounting Software*

This section will employ Algorithms to demonstrate the effectiveness of the training process. Algorithm 2 (in section 3.71) was executed. The actual risk values (ARV) are compared to the predicted risk values (PRV) to determine the correct training formula. In this case, PRV < ARV for all objectives, so the formula: $\mathbf{y_i = r.w_i + (1\text{-}r)(x_i / 5)}$, where $y_i$ is the new attribute weight, r = .95, $w_i$ is the weight from past beta tests, and $x_i$ is the normalized attribute value for current project. The calculations for this experiment are highlighted in Table 16. The results are presented in Table 15 demonstrating an improvement in the risk predictions.

**Table 15. PRV vs. ARV Comparisons for Accounting Software after Training the Functions**

| | Environmental Dependency | Function Coverage Completeness | Localization | Robustness | Software Vulnerability | UI Accessibility | UI Usability |
|---|---|---|---|---|---|---|---|
| **PRV** | ***2.184*** | ***1.291*** | ***1.738*** | ***1.653*** | ***2.657*** | ***1*** | **2.134** |
| ARV | 3 | 3 | 2 | 2 | 3 | 1 | 3 |

## Table 16. Accounting Software Function Training Data Set

**Environmental Dependency**

| Attribute Name | L Weight | A Weight | Value | Total |
|---|---|---|---|---|
| amount of threads generated | 0.120 | 0.100 | 1 | 0.100 |
| CPU utilization | 0.130 | 0.109 | 2 | 0.217 |
| dynamic link library requirements | 0.054 | 0.045 | 1 | 0.045 |
| firewall port required. | 0.108 | 0.090 | 2 | 0.181 |
| incoming client service request | 0.118 | 0.099 | 3 | 0.296 |
| multiple system requirements | 0.054 | 0.045 | 1 | 0.045 |
| network connections required | 0.118 | 0.099 | 3 | 0.296 |
| open API | 0.118 | 0.099 | 3 | 0.296 |
| physical processors required | 0.140 | 0.117 | 3 | 0.351 |
| special user accounts required. | 0.052 | 0.043 | 3 | 0.130 |
| web portals required | 0.086 | 0.072 | 2 | 0.144 |
| web service request requirements | 0.098 | 0.082 | 1 | 0.082 |
|  | 1.200 | 1.000 |  | **2.184** |

**Robustness**

| Attribute Name | L Weight | A Weight | Value | Total |
|---|---|---|---|---|
| number of lines of source code | 0.061 | 0.055 | 1 | 0.055 |
| amount of threads generated | 0.087 | 0.078 | 1 | 0.078 |
| changes to dynamic data | 0.113 | 0.102 | 1 | 0.102 |
| class inheritances | 0.148 | 0.134 | 2 | 0.267 |
| multitask command buttons | 0.138 | 0.125 | 1 | 0.125 |
| program switches lacking default clause | 0.107 | 0.096 | 3 | 0.289 |
| use case totals | 0.097 | 0.087 | 2 | 0.175 |
| unrestricted text fields | 0.138 | 0.125 | 1 | 0.125 |
| user interface complexity | 0.133 | 0.120 | 3 | 0.359 |
| user-levels | 0.087 | 0.078 | 1 | 0.078 |
|  | 1.110 | 1.000 |  | **1.653** |

**Function Coverage Completeness**

| Attribute Name | L Weight | A Weight | Value | Total |
|---|---|---|---|---|
| hardware and software requirements | 0.281 | 0.281 | 1 | 0.281 |
| software wizards | 0.214 | 0.214 | 1 | 0.214 |
| unrestricted text fields | 0.214 | 0.214 | 1 | 0.214 |
| use case totals | 0.291 | 0.291 | 2 | 0.583 |
|  | 1.000 | 1.000 |  | **1.291** |

**Localization**

| Attribute Name | L Weight | A Weight | Value | Total |
|---|---|---|---|---|
| code locale | 0.270 | 0.262 | 2 | 0.524 |
| fonts effect on UI | 0.160 | 0.155 | 1 | 0.155 |
| languages supported | 0.270 | 0.262 | 2 | 0.524 |
| special input devices | 0.110 | 0.107 | 1 | 0.107 |
| screens affected by font adjustments | 0.220 | 0.214 | 2 | 0.427 |
|  | 1.030 | 1.000 |  | **1.738** |

## Software Vulnerability

| Attribute Name | L Weight | A Weight | Value | Total |
|---|---|---|---|---|
| firewall port required | 0.299 | 0.277 | 2 | 0.554 |
| network connections required | 0.254 | 0.235 | 3 | 0.704 |
| open API | 0.142 | 0.131 | 3 | 0.394 |
| special user accounts required | 0.309 | 0.286 | 3 | 0.859 |
| web portals required | 0.076 | 0.070 | 2 | 0.141 |
|  | 1.080 | 1.000 |  | **2.653** |

## UI Accessibility

| Attribute Name | L Weight | A Weight | Value | Total |
|---|---|---|---|---|
| firewall port required | 0.960 | 1.000 | 1 | 1.000 |
| network connections required | 0.000 | 0.000 | 0 | 0.000 |
|  | 0.960 | 1.000 |  | **1.000** |

## UI Usability

| Attribute Name | L Weight | A Weight | Value | Total |
|---|---|---|---|---|
| message boxes | 0.140 | 0.130 | 1 | 0.130 |
| multitask command buttons | 0.140 | 0.130 | 1 | 0.130 |
| screen traversal | 0.266 | 0.249 | 5 | 1.243 |
| software wizards | 0.053 | 0.050 | 1 | 0.050 |
| UI responses required | 0.140 | 0.130 | 1 | 0.130 |
| user interface controls | 0.150 | 0.140 | 2 | 0.280 |
| unrestricted text fields | 0.183 | 0.171 | 1 | 0.171 |
|  | 1.070 | 1.000 |  | **2.134** |

# Chapter 4

# Software Metrics-Based Approach to Beta Testing Design

## 4.1 Major Components of a Beta Testing Design

Currently, there are no known formal industry standards or models used by enterprise software manufacturers to manage beta testing software. Most modern beta testing process models are proprietary and change based on historical data (the results of prior beta tests). Enterprise software developers rely on strict time duration and a set amount of beta testers (clientele) as a model to guide the beta testing process. Good beta testing design is accomplished by an understanding of important areas that have the potential to cause problems in a production environment and have “real-world” testers validate potential issues and provide feedback. This study provides a software metrics-based approach to generating a formal design for enterprise software beta testing.

This research introduces five major components of beta testing design (see Figure 6). The five components are influenced by the predicted risk values derived from the metric functions for each beta testing objective. The first component is selecting the proper group of beta testers (beta testing clientele) based on results received from the software metrics. The second component involves understanding important software components that require special attention based on the predicted value levels. Constructing a questionnaire, as an instrument to collect feedback and to guide beta testers, is the third component of the beta testing design provided in the study. The fourth component of the beta testing design is determining the size of beta testers required to validate the product in beta. The final (fifth) component is to determine the duration of beta test answering the question: *How long should the product remain in the beta testing phase?*.

Beta testing design provides a formal framework used to guide software manufacturers in planning the beta testing process. In addition, the beta testing design assists software manufacturers with properly focusing time and tasks, in preparation for beta testing a product. The outcome of this methodology is improved quality in the final product, in addition to saving cost and building positive relationships with the user base community.

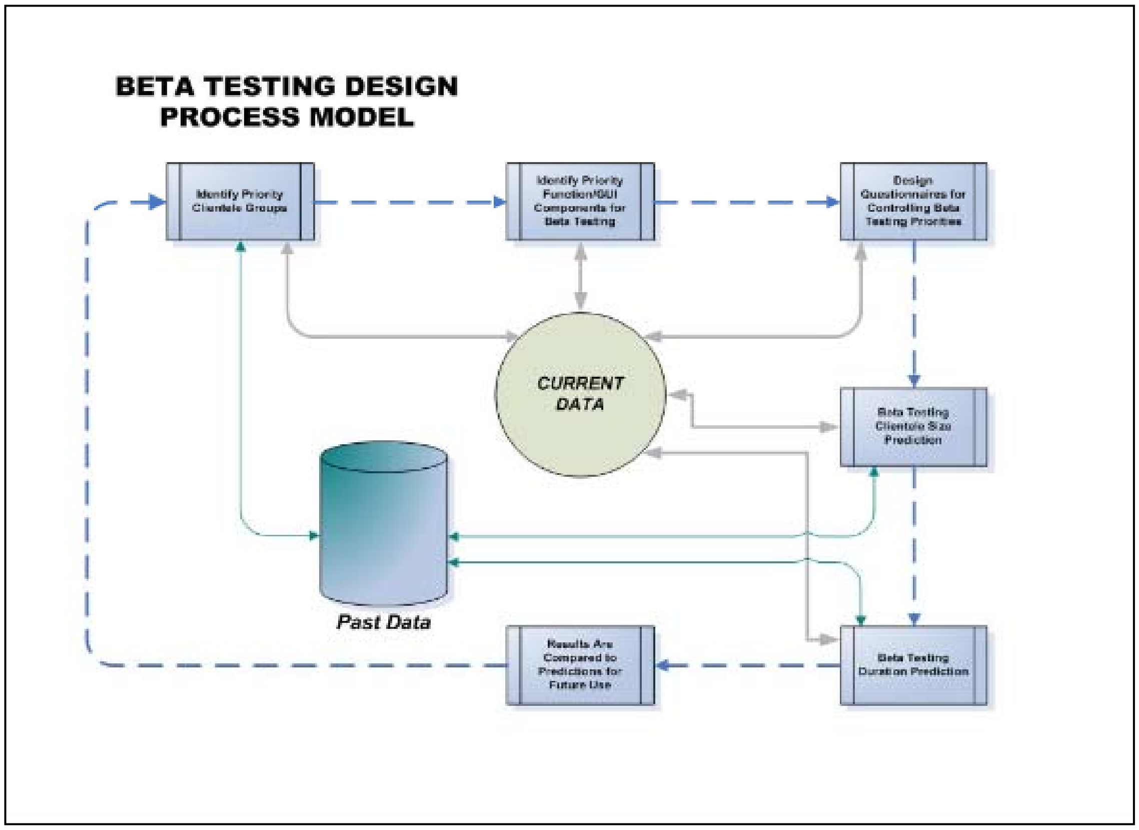
BETA TESTING DESIGN PROCESS MODEL
Identify Priority Clientele Groups
Identify Priority Function/GUI Components for Beta Testing
Design Questionnaires for Controlling Beta Testing Priorities
CURRENT DATA
Beta Testing Clientele Size Prediction
Past Data
Results Are Compared to Predictions for Future Use
Beta Testing Duration Prediction

**Figure 6. Beta Testing Design Process Model**

### 4.2 Software Metrics-Based Beta Testing Design

Prior to beta testing a product, beta test managers (or lead software developers) must predict areas of the software that have potential to create problems or issues if introduced to a production environment without proper testing.  After beta test managers determine the product is complete and ready for beta testing, attribute data is collected from the current product to apply to software metrics. In cases where products have completed beta testing in the past, beta test managers will review data from prior projects (or similar) for trending purposes. The Enterprise Software Profile (See Appendix A) is used to collect the required product data and the appropriate algorithms are employed to obtain predictions. Based on prediction results (received from objective metric functions), beta test managers will proceed with the design of the beta testing. The next set of sub-sections outlines the impact software metrics has on the beta testing design.

### *4.2.1 Identify Beta Testing Priority Clientele Groups*

Identifying the type of beta testing clientele groups is essential to the testing process. The concept of beta testing relies on validating the software in a “real" environment to assure the product functions as designed. In addition, the product gains final approval from the target group of end-users. Craig and Jaskiel stated in their book, *Systematic Software Testing*, “Many users will not have any experience in testing, but their knowledge of the business function may make them valuable additions in spite of their lack of experience[13].” It is important to have beta testers from multiple tiers and level of experience with the application.

The software metrics results assist beta test managers in prioritizing the type of testing clientele to solicit for the beta test. For example, if the objective metric value for localization is at a level where it presents a high potential of problems, the beta test managers will recruit a class of beta testers from target foreign countries (i.e. if the product is localized for Germany, German end-users will be recruited to beta test the product). The same holds true in a similar scenario where the objective metric value is high for environmental dependency. This indicates testers with administrator or engineering backgrounds are required. Other metric results will provide guidance in additional types of testers required for the beta test.

The testing user group is an organization of qualified end-users offers some influence of the product in beta. Testing user groups also provides support during the testing phase and provides a forum where testers can exchange ideas, offer information, and give advice that will improve the product in beta[34]. In this study, beta testers are classified into several categories. There are several groups of end-users targeted for beta testing. The testing groups are:

- System Engineers
- New end user groups
- Experienced user groups
- Regional experience end user Group
- Regional system engineers
- Accessibility User Group

System Engineers are users with special administrator rights to the environment the product will be installed. System engineers have administrator privileges (or special access) to hardware, software, secured areas, network components, and databases. In an enterprise environment, the system administrator may be several people or a special department[15]. New end user groups are software end-users with no formal experience with the product. Most new end-users have direct interest in the product to meet a specific need, training requirements, or other significant interest.

Experienced user groups are end-users with prior or current knowledge of the software and software domain. The experienced user group understands the functionality and uses the product in production environment. Regional experience end user groups are end-users in target countries with prior or current knowledge of the application and application domain. The regional experience user group has used the product for a significant period and clearly understands the functionality of the product. Regional based system engineers are end-users with administrator or system rights in the target country the application will be installed. Regional based system engineers must have rights (or access) to hardware, software, secured areas, databases, and network resources. Accessibility specific user Group is the end-user base that has special challenges such as sight, hearing, and physical disabilities that hinder usage of the application. This base of end-users requires the adjustments in order operate the software.

### *4.2.2 Identify Priority Functions/GUI Components for Beta Testing*

When planning a beta test, testing managers' focus on areas that require the most attention during the testing process. Currently, beta test managers focus on "new" features or use cases as priority. The software metric based approach will guide beta test managers with prioritizing the features, functions, and software sectors of an application requiring validation in the beta testing process. In a scenario where the objective metric value is high for software vulnerability, the beta tester will need to add a higher priority to assure security features are not vulnerable. If there are several areas that indicate the potential for risk, beta test managers will provide high priority to the trouble areas.

### *4.2.3 Design Questionnaires for Controlling Beta Testing Priorities*

Beta testing products are implemented with minimum intervention from the software manufacturer's developers. However, beta test managers design testing questionnaires to provide testers with a level of guidance in areas with high significance and to generate feedback from the testing group. Software metrics are effective in assisting beta testing managers in customizing questionnaires based on the level of results received from metrics. Today, questionnaires are designed by past tests, where problems were introduced during testing, new features, components and use cases. Software metrics provide additional information to assist in prioritizing information and controlling the design of the questionnaire. To illustrate this point, if environmental dependency yields results that indicate a high risk of problems, beta test managers will design questionnaires to focus on deployability of the product.

### *4.2.4 Minimal Beta Testing Clientele Size Prediction*

Today, industry software manufacturers use fixed beta testing size constraints to validate a successful beta test. Software manufacturers are focused on eliminating the amount of bugs in applications prior to release believing the more testers the higher chance of eliminating bugs. Most software manufacturers continue the beta testing process until a set amount of testers is participating in the beta test (Table 1.).

Software metrics provide a more structured method for predicting the clientele size requirements for beta testing enterprise software. If a software metric-based design model were used for prior beta testing, beta testing managers could use trending (past results) to predict the size required for the new product or products from a similar application domain. For example, if past tests yielded higher or similar numbers for the UI usability objective, and the number of beta testers that are actual or potential end-users (non-administrators) of the product was low, the new metric results will indicate that current and future tests will require more end-users to beta test the product. Past results impact the number of beta testers required for current project.

Table 17. Sample Beta Testing User Group

| Estimated User Base | Beta Testers Required |
|---|---|
| | Percentage of user base |
| Below 500 users | 5% |
| 500 – 2500 | 10% |
| 2500 or More | 15% |

### *4.2.5 Minimal Beta Testing Duration Prediction*

Beta testing time is important to the success of the beta process. However, predicting the duration of a beta test requires historical data and the expected amount of feedback. Historical data influences the results received from prior tests. Beta test managers use both the current and prior data derived from software metrics to set the testing duration. The amount of feedback required is another variable that impacts the duration of a beta test. In the case where products that yield a high vulnerability metric result for robustness and usability for past and current project, the beta managers can use results from past beta test to predict the length of time. For instance, if the past test was 3-months and beta test managers believe more time was needed to receive better feedback, the new test should be planned for longer.

## 4.3 Formal Procedure for Beta Testing Design

### Step 1. Identify beta testing priority clientele groups

Review the PRV for each objective. Risk levels determine the importance of engineers that should be targeted for the beta testers. Start with the objective that yields the highest predicted risk level and match the required user group for beta testing the objective. Below is a list of required engineers based on the outcome of risk levels.

*Environmental Dependency*

Systems Engineers – are significant because end-users with systems experience have special rights to target environment and will provide the most effective feedback system related issues, concerns, and/or enhancement recommendation.

Experienced Users Group - is required because experienced end-users use advanced features in the application that could potentially uncover software errors. The software errors in this case may be dependency related.

*Function Coverage Completeness*

Experienced User Group – are vital because they are capable of providing complete feedback on the software coverage because of tenure with the product.

New Users Group – are significant because expectations of the product are set with need based on the product description. The new end-user group provides novel feedback on current features versus expected and desired functionality.

*Localization*

Regional Experience User Group – are best suited for localization beta testing because they are more skilled with the product and can test localized functionality to reveal potential issues in the product.

Regional Systems Engineers – are most efficient when beta testing localization because system experience may be required to validate some system issues experienced with the product in test. The experience base of foreign systems engineers will have the access to infrastructure resources to effectively implement the software product into production.

*Robustness*

New User Group – are significant to this process because they will introduce the majority of usage errors (operator errors). The objective captures usage patterns to improve error handling in the software.

Experienced User Group – are best suited for robustness provided this base of users may introduce usage patterns not covered in the design phase, which may result in the discovery of errors. Experienced users also provide a more seasoned level of usage patterns.

*Software Vulnerability*

System Engineers – are essential to beta testing software vulnerability because of their access and experience with the infrastructure. Systems engineers will provide good feedback on enhancements, requirements or features in the application that will reduce the level of vulnerability introduced by the application.

Experienced User Group – is important to this objective based on their understanding of the product (and product domain). The experienced users will provide valuable feedback on requirements and desired functionality. This information will assist developers in planning ways to lower vulnerability based on need and demand.

*UI Accessibility*

Accessibility User Group – is required for beta testing this objective because users with special challenges will provide the essential feedback required.

*UI Usability*

Experienced User Group – is always essential when beta testing usability because tenure with the product provides the comparison feedback. Comparison feedback is based on past products versus new product functionality.

New User Group – is vital when testing usability features in the application providing integral feedback on GUI, features, and ease of use. The ease of use feedback influences the current and future release of the product in test.

Table 18 is useful to quickly identify the type of user group required based on PRV.

**Table 18. Required Type of User Group for Beta Test Based on PRV**

| Clientele Groups | | | | | | |
|---|---|---|---|---|---|---|
| Objectives | System Engineers | Experience User Group | New User Group | Regional Experienced User Group | Regional New User Group | Accessibility User Group |
| Environmental Dependency | ● | ● | | | | |
| Function Coverage Completeness | | ● | ● | | | |
| Localization | | | | ● | ● | |
| Robustness | | ● | ● | | | |
| Software Vulnerability | ● | ● | | | | |
| User Interface Accessibility | | | | | | ● |
| User Interface Usability | | ● | ● | | | |

**Step 2. Identify priority functions/GUI components**

Again, the PRV for the current beta testing project is reviewed. The risk levels will reveal areas that require the most focus.

If PRV reveal priority should be given to:

- Environmental Dependency - the following product function requires special attention:
    - Product installation
    - Advance features

  - Benchmark product for degradation issues (i.e. bottlenecks, high utilization)
  - Validate access to resources (i.e. resources, sub networks, etc).
  - Web based resources are functional (i.e. portals, web-based forms, etc)
  - Web services are functional
- Function Coverage Completeness - the following product functions must be validated:
  - Software wizards
  - Graphical user interface is comprehensive
  - Test each product use case
- Localization - the following product functions require validation:
  - Font changes maintain organization of all forms
  - Language changes are effective
  - Special devices are operable
  - Time/Country/Currency changes are functioning
- Robustness - the following function requires attention:
  - Graphical user interface handles input errors
  - Error message are comprehensive and user understands the nature of the problem
  - All user groups are working according to access levels

- Software Vulnerability - the following software components must be validated:
    - User access levels are secure
    - Application access to bridges and routers are not creating vulnerabilities
    - Web portals are operable and not introducing security issues
    - Network protocols are encrypting data correctly and not exposing the current infrastructure
    - APIs are secure
- UI Accessibility - the following software components require validation:
    - Test fonts are readable to users with limited sight
    - Special peripherals are operable at the same accuracy and response time as original peripherals (i.e. special carpal tunnel mouse works the same as regular mouse)
- UI Usability - the following software components require special attention
    - Test GUI to assure ease of use
    - Message boxes provide accurate instructions
    - Software wizards are operable and yield exact results
    - GUI controls are operable and well organized
    - Navigation thorough the software provides hints and does not confuse the end user.

**Step 3. Design questionnaires for controlling beta testing priorities**

Review the PRV for each objective. Place objectives in order by risk level. The questionnaire will be outlined based on the required feedback. This information will guide the end-user in testing features of the application that provided a higher risk value. Below are examples of questions and components that can be used to build the testing questionnaire.

Environmental Dependency solicits feedback on:

- Product installation: Capture user experiences with installing the product
- Usage of Advanced Features: Suggest information on advanced features
- Utilization: Did the product introduce any high utilization issues
- Bottlenecks: Did the product introduce degradation to the current system or network
- Access to Resources: Did the product gain access to additional resources, networks, or databases with ease?
- Web resources: Capture users experience with web based resources, etc.

Function Coverage Completeness solicits feedback on:

- Software wizards: Does the software wizard yield the appropriate outcomes? Or do software wizards assist in easing the experience with the application?
- Graphical user interface: Are the screens providing the appropriate amount of information to receive the desired outcome?
- Use Case: Are the product specifications accurate and complete?

Localization solicits feedback on:

- Font Change: Does the character changes for desired country maintain organization to all screens?
- Language Support: Is the desired language correct and effective?
- Special Localized Peripherals: Does the specially mapped peripherals operate effectively?
- Conversion Changes: Does the Time/Country/Currency provide accurate translations?

Robustness solicits feedback on:

- Input Errors: Does the graphical user interface alert user of errors?
- Message Boxes/Alerts: Are the error message boxes comprehensive and does the end-user understand the nature of the problem?

Software Vulnerability solicits feedback on:

- User Access: Do the user level requirements create vulnerabilities in the product? Are the access levels secure?
- Router/Bridge Access: Does the number of port requirements create software and/or network vulnerabilities?
- Web Accessible Features: Are the web accessible features operable? Do the web features create security concerns/issues?
- Network Connections: Do network protocols communicate correctly? Does the level of network protocols required creating network vulnerabilities or violations?

- API: Does utilization of the current APIs create network issues?

UI Accessibility solicits feedback on:

- Accessibility Fonts: Do accessibility fonts create a readable user display?
- Accessibility Hardware: Are the special accessibility peripherals operable with the same accuracy and response time as original peripherals?

UI Usability solicits feedback on:

- GUI: Does the graphical user interface provide ease of use?
- Message Boxes: Do the message boxes provide accurate instructions?
- Software Wizards: Do the software wizards assist in complex tasks? Are the software wizards useful in yielding exact results?
- GUI Controls: Are GUI controls operable and well organized?
- Application Navigation: Is navigating through the software complex?

Additional questions may be added based on level and experience with the application to enhance the questionnaire.

**Step 4. Predict minimal beta testing clientele size**

Review PRV for the current project. Find the mean of predicted risk levels using the following formula

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

Add all PRV to get the total sum. Divide total by 7. The answer is rounded to the nearest whole number.

Use the following matrix to match the mean of predicted risk levels for the current project to obtain minimum beta testing clientele size.

| Average Risk Level | Current user Base |
|---|---|
| 1 | 1% |
| 2 | 4% |
| 3 | 6% |
| 4 | 8% |
| 5 | 10% |

This is just a base set of data. The actual percentages will change with experience and past data.

Step 5. Predict minimal beta testing duration

Review the PRV for the current project. Match the current PRV with the matrix provided below. Add the number of weeks to get the minimum predicted number of weeks for the current project. The information provided in this matrix can be changed to match past data.

| Predicted Risk Level | Minimum Weeks |
|---|---|
| 1 | 2 |
| 2 | 2.5 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |

## 4.4 An Example of the Beta Testing Design

Beta testing design is essential to outlining and organizing a beta test. In the case of the accounting software (used as an example in section 3.7), the predicted risk levels for each objective are highlighted in Table 19. Beta test managers must pay attention to these objectives when designing the beta test.

Table 19. Accounting Software - Predicted Objective Metrics Results

| Environmental Dependency | Function Coverage Completeness | Localization | Robustness | Software Vulnerability | UI Accessibility | UI Usability |
|---|---|---|---|---|---|---|
| 2.17 | 1.29 | 1.74 | 1.60 | 2.65 | 1 | 2.05 |

Applying the five components of beta testing design, the first area of focus is to identify and prioritize beta testing clientele groups. In this scenario, software vulnerability and environmental dependency objectives yield risk values higher than others, although for this application it has moderate risk levels the beta testing design is still effective.

**Step 1. Identify beta testing priority clientele groups**

After reviewing the PRV for the current project, results suggest focus on the software vulnerability, environmental dependency, and UI usability objectives. Reviewing the suggested users in Table 18, experienced users and systems engineers are the most targeted clientele for this beta test. A small group of new end-users is required for UI usability testing objective and recommended group of Regional Experienced testers are required for the localization objective.

**Step 2. Identify priority functions/GUI components**

In the second step, PRV levels for software vulnerability, environmental dependency, and UI usability objectives suggest priority for the following components:

*Software vulnerability*

- User access levels are secure

- Application access to bridges and routers are not creating vulnerabilities
- Web portals are operable and not introducing security issues
- Network protocols are encrypting data correctly and not exposing the current infrastructure
- APIs are secure

*Environmental dependency*

- Product installation
- Advance features
- Benchmark product for degradation issues (i.e. bottlenecks, high utilization)
- Validate access to resources (e.g. resources, sub networks, etc).
- Web based resources are functional (e.g. portals, web-based forms, etc)
- Web services are functional

*UI usability*

- Test GUI to assure ease of use
- Message boxes provide accurate instructions
- Software wizards are operable and yield exact results
- GUI controls are operable and well-organized
- Navigation through the software provides hints and does not confuse the end user.

The beta testing design suggests priority should focus on the aforementioned components. However, other components are tested at the discretion of the beta test manager.

**Step 3. Design questionnaires for controlling beta testing priorities**

In Step 3. A customized questionnaire is created based on the severity and level of PRV. In this case, The PRV for the software vulnerability, environmental dependency, and UI usability required more focus (yielding PRV values of 2.56, 2.17, and 2.05 respectively). Attention is also given to other objectives to provide a comprehensive questionnaire. Using these guidelines, the Sample Beta Testing Questionnaire (Figure 7) is used to demonstrate how the PRV impact the design of this document.

Figure 7. Sample Beta Testing Questionnaire

| | |
|---|---|
| **Product Name:** | |
| **Date:** | |
| **Beta Tester Name:** | |
| **Beta Tester Company Name:** | |
| **Beta Tester Role:** | |
| **Years of Experience with Product:** | |

| **Product Description** |
|---|
| |

## Section 1. Environmental Dependency

The focus of the environmental dependency objective is to measure software dependence on hardware, shared software components, and other supplemental software to confirm that the required dependency does not impede operation. After installing the product in your environment, test the product to assure that the product has successfully installed without introducing any problems. Please answer the questions below and rate the product based on your experience during implementation.

| **Installation Questions** | **Yes** | **No** | **Not Applicable** |
|---|---|---|---|
| Did you install the product on each of the supported platforms (i.e. windows, UNIX, etc)? | ☐ | ☐ | ☐ |
| Did the product introduce degradation after installation? | ☐ | ☐ | ☐ |
| Did the product increase system utilization after installation? | ☐ | ☐ | ☐ |
| Does the advance install feature function correctly? | ☐ | ☐ | ☐ |

| **Rating** | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** |
| Rate your overall experience installing the product. | ☐ | ☐ | ☐ | ☐ | ☐ |
| Rate your experience using the advanced install features included with this application. | ☐ | ☐ | ☐ | ☐ | ☐ |
| Rate the usefulness of the wizards during the installation of the application. | ☐ | ☐ | ☐ | ☐ | ☐ |
| Rate the product's overall utilization of CPU. | ☐ | ☐ | ☐ | ☐ | ☐ |

| **Based on your experience please provide additional information that will enhance the installation process.** |
|---|
| |

| System Integration Questions | Yes | No | Not Applicable |
|---|---|---|---|
| Did you encounter major issues integrating the current product to your current environment? | ☐ | ☐ | ☐ |
| *If yes, please explain the problem(s):* | | | |
| Does the product integrate with external systems? | ☐ | ☐ | ☐ |

| Rating | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Rate the product's integration and ability to communicate with current systems and applications. | ☐ | ☐ | ☐ | ☐ | ☐ |

| Resource Dependency Questions | Yes | No | Not Applicable |
|---|---|---|---|
| Do the web-based resources function as designed? | ☐ | ☐ | ☐ |
| Do the web-based resources provide ease of use? | ☐ | ☐ | ☐ |
| Do the web-based resources support the current thin client infrastructure? | ☐ | ☐ | ☐ |

| Rating | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Rate the overall usefulness of the web based resources | ☐ | ☐ | ☐ | ☐ | ☐ |

| Please provide any additional comments. |
|---|
| |

## Section 2.  Software Function Coverage

The focus of software function coverage is to confirm that the software meets customer expectations. Based on your operational experience with this application, please answer the questions below and rate the product.

| Software Wizard Questions | Yes | No | Not Applicable |
|---|---|---|---|
| Do the software wizards assist in ease of use with the application? | ☐ | ☐ | ☐ |
| Do the software wizards increase productivity? | ☐ | ☐ | ☐ |
| Do the software wizards enhance overall operation of the application? | ☐ | ☐ | ☐ |
| Does each software wizard yield the expected outcome based on the wizard's description? | ☐ | ☐ | ☐ |

| Rating | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Rate the usefulness of the wizards during the installation of the application. | ☐ | ☐ | ☐ | ☐ | ☐ |
| Rate the overall effectiveness of the software wizards based on your experience with each section. | ☐ | ☐ | ☐ | ☐ | ☐ |

| Forms Questions | Yes | No | Not Applicable |
|---|---|---|---|
| Do the screens and forms provide the appropriate amount of information to receive the desired outcome? | ☐ | ☐ | ☐ |
| Are the screens easy to use? | ☐ | ☐ | ☐ |
| Are the screens/forms easy to navigate? | ☐ | ☐ | ☐ |

| Rating | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Rate your overall experience using the forms in this application. | ☐ | ☐ | ☐ | ☐ | ☐ |

| Application Specification Questions | Yes | No | Not Applicable |
|---|---|---|---|
| Are the product specifications comprehensive? | ☐ | ☐ | ☐ |

| Rating | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Rate how the product meets your overall expectation. | ☐ | ☐ | ☐ | ☐ | ☐ |

| **Based on your experiences, what enhancements would you recommend to improve your expectations of this product.** |
|---|
| |

## Section 3. Software Globalization/Localization

The software localization section collects feedback on adjustments to the software to prepare the application for deployment in foreign countries. Based on adjustments to language changes, conversion changes, and hardware mapping, please answer the questions below and provide a rating for the localized features in this product.

| **Localization Questions** | **Yes** | **No** | **Not Applicable** |
|---|---|---|---|
| Does the semantic change to the appropriate country language create organizational issues on the graphical user interface (form change)? | ☐ | ☐ | ☐ |
| *If yes, please describe the issue* | | | |
| Are there any language issues in the application? | ☐ | ☐ | ☐ |
| *If yes, please describe the issue.* | | | |
| Do the specially mapped peripherals operate effectively? | ☐ | ☐ | ☐ |
| Are the time/date changes correct in the application after installation? | ☐ | ☐ | ☐ |
| Are there any conversion issues (date/currency/metric)? | ☐ | ☐ | ☐ |

| **Rating** | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** |
| Rate the overall application adaptation to the targeted country infrastructure. | ☐ | ☐ | ☐ | ☐ | ☐ |

| **Based on your experiences, what enhancements would you recommend to improve the localization/globalization features in this application** |
|---|
| |

## Section 4. Software Robustness

The focus of the software robustness is to recognize incorrect application data and the impact of user errors, and to capture usage patterns to measure how it impact the software. In this section, feedback is required on how the applications respond to errors. In addition, feedback is required on warning and error messages. Please answer the questions below and provide a rating based on the robustness features in this application.

| **Error-handling Questions** | **Yes** | **No** | **Not Applicable** |
|---|---|---|---|
| Does the graphical user interface alert the end-user of an error? | ☐ | ☐ | ☐ |
| In the event of an error, does the application respond appropriately? | ☐ | ☐ | ☐ |
| Is the description of the error understandable? | ☐ | ☐ | ☐ |
| If the application provides a warning, is it comprehensive? | ☐ | ☐ | ☐ |

| **Describe the type of error encounter and the action taken to resolve the error.** |
|---|
| |

| **Rating** | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** |
| Rate the application's ability to handle errors. | ☐ | ☐ | ☐ | ☐ | ☐ |

| **Based on your experiences, what enhancements would you recommend to improve how the application handles errors or warnings.** |
|---|
| |

## Section 5. Software Vulnerability

The software vulnerability validations are potential security violations focusing on vulnerabilities in communication and other components in the application which may introduce or create security violations. Please answer the questions below and provide a rating based on security vulnerabilities in this application.

| **Software Security Questions** | **Yes** | **No** | **Not Applicable** |
|---|---|---|---|
| Do the user level requirements create vulnerabilities in the product? | ☐ | ☐ | ☐ |
| *If yes, please explain the security violations.* | | | |
| Are the port access levels secure? | ☐ | ☐ | ☐ |
| Does the number of port requirements create software and/or network vulnerabilities? | ☐ | ☐ | ☐ |
| *If yes, please explain the nature of the vulnerability and the port number..* | | | |
| Are the web accessible features operable? | ☐ | ☐ | ☐ |
| If yes, do the web features create security concerns/issues? | ☐ | ☐ | ☐ |
| *Please explain the security concerns* | | | |
| Do the network protocols communicate correctly? | ☐ | ☐ | ☐ |
| Does the level of network protocols required create network vulnerabilities and/or violations? | ☐ | ☐ | ☐ |
| *If yes, please explain the vulnerabilities and/or violations.* | | | |
| Does the utilization of the API required by this application create infrastructure security violations? | ☐ | ☐ | ☐ |
| Does this product create or introduce any security violations? | ☐ | ☐ | ☐ |
| *If yes, please explain the security violations.* | | | |

| Rating | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Rate the overall security of this application. | ☐ | ☐ | ☐ | ☐ | ☐ |

| Based on your experiences, what enhancements would you recommend to improve the security of this application. |
|---|
| |

## Section 6. User Interface Accessibility

The user interface accessibility (UIA) are features of an application that are designed to assist end-users with special physical needs. Please answer the questions below and provide a rating for the accessibility features in this application.

| Accessibility Questions | Yes | No | Not Applicable |
|---|---|---|---|
| Does the accessibility fonts included with this application create a more readable user display? | ☐ | ☐ | ☐ |
| Are the usability peripherals operable? | ☐ | ☐ | ☐ |

| Rating | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| Rate the accuracy of the accessibility peripherals. | ☐ | ☐ | ☐ | ☐ | ☐ |
| Rate your overall experience with the accessibility features in this application. | ☐ | ☐ | ☐ | ☐ | ☐ |

| Based on your experiences, what enhancements would you recommend to improve the accessibility features of this application. |
|---|
| |

## Section 7. Software Usability

The software usability section is used to validates that the graphical user interface is simple and promotes ease of use. The usability feedback is used to enhance the overall utilization of the software. Based on your experience with the application, please answer the questions below and provide a rating for the usability features in this application.

| **User Interface Questions** | **Yes** | **No** | **Not Applicable** |
|---|---|---|---|
| Does the graphical user interface provide ease of use? | ☐ | ☐ | ☐ |
| *If no, please explain why:* | | | |
| Do the message boxes provide accurate instructions? | ☐ | ☐ | ☐ |
| Are the GUI controls operable and well organized? | ☐ | ☐ | ☐ |
| Do the software wizards assist in complex tasks? | ☐ | ☐ | ☐ |
| Are the software wizards useful in yielding exact results? | ☐ | ☐ | ☐ |
| *If no, please explain issues with results provided by use of software wizards:* | | | |
| Does the help menu prove clear instructions on how to perform a task? | ☐ | ☐ | ☐ |
| Is navigating through the software complex? | ☐ | ☐ | ☐ |

| **Rating** | Poor | Below Average | Average | Good | Excellent |
|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** |
| Please rate the overall layout of screens and menus. | | | | | |
| Rate the overall experience with form instructions. | | | | | |
| Rate how user friendly the application is. | | | | | |

| **Based on your experiences, what enhancements would you recommend to improve the usability features of this application.** |
|---|
| |

**Step 4. Predict minimum beta testing clientele size**

The PRV was reviewed for current project. The following mean formula is employed to receive the average risk level for this project.

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

= (2.17 + 1.29 + 1.74 + 1.60 + 2.65 + 1 + 2.05) / 7
= 12.5 / 7
= 1.79
= ***2***
The total is rounded to nearest whole number.

Using the matrix provided below, the PRV suggests a minimum size of *4%* of the current customer base.

| Average Risk Level | Current user Base |
|---|---|
| 1 | 1% |
| 2 | 4% |
| 3 | 6% |
| 4 | 8% |
| 5 | 10% |

This is just a base set of data. The actual percentages will change with experience and past data.

**Step 5. Predict minimal beta testing duration**

The PRV was reviewed for the current project and matched to recommend minimum number of weeks. The matrix provided below is used as an instrument to calculate the minimum predicted beta testing duration. Add the number of weeks to get the minimum predicted number of weeks for the current project. The information provided in this matrix will improve with experience.

| Predicted Risk Level | Minimum Weeks |
|---|---|
| 1 | 2 |
| 2 | 2.5 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |

| | | | |
|---|---|---|---|
| Environmental dependency | *2* | 2.5 | |
| Function coverage completeness | *1* | 2 | |
| Localization | *2* | 2.5 | |
| Robustness | *2* | 2.5 | |
| Software Vulnerability | *3* | 3 | |
| UI Accessibility | *1* | 2 | |
| UI Usability | *2* | 2.5 | |
| | | 17 | weeks |

# Chapter 5

# Beta Testing Case Studies

In this chapter, two real-world enterprise products were used to demonstrate the effectiveness of the software metrics based beta testing design provided in this study. The two products are a major enterprise infrastructure monitoring application and a similar enterprise virtualization monitoring software. The products have participated in a beta test in the past 8 months and feedback from the test was used correctly measure the actual metric values. The data used in this chapter was collected using a survey and face-to-face interviews. Also, issue logs were used to measure historical data. This experiment was conducted over a two-month period and closely monitored by the researcher.

## 5.1 Enterprise Infrastructure Monitoring Application

Enterprise infrastructure monitoring is a growing trend in business application because of the demand created by solution services such as ERP, CRM, and other business applications. The increase in data exchange has increased the number of bottlenecks because of the sheer demand of data exchange and network communication. The product used in this section is a robust application that helps corporations in managing network resources, providing a large number of services to assist in better managing the network.

### *5.1.1 Product Overview*

The first enterprise product is infrastructure management application, responsible for monitoring the health and availability of resources for robust environments.

The product uses agents' technology to monitor hardware and software in the environment to provide real-time feedback in the event of issues. The product also includes neural nets technology used to implement predictive management useful in predicting system bottlenecks, possible software and hardware failure, security issues and other infrastructure related issues. In addition, the agents' technology sends alerts supporting rules-based methodology.

This enterprise product was designed to support a diverse environment supporting nearly every available operating system and environment, major enterprise databases, hardware platforms, messaging infrastructures, etc. The application is designed to support out-of-the-box functionality and collect information automatically. However, installation may require assistance from the support group for complex environments. In addition, the product requires a SQL database to store information, and a separate server to manage the alerts. The application managers require WINTEL or LINUX servers but may be administered remotely using a java based application or web browser.

As a major competitor in the EIM market space, this product supports over 6000 different companies in 15 different countries. The product is also built to support every major business sector providing real-time analysis for a host of different applications.

Data for this experiment was collected using the Enterprise Product Profile in Appendix A, the company's internal issue log, and face-to-face interviews. The beta test manager used several resources to provide accurate information including polling several senior developers and benchmark utility. This enterprise application has just completed a 5-month beta test that validated a new version of the software. An interview was conducted with the beta test manager after receiving the data to discuss the information in the Enterprise Product Profile, which provided attributes values and actual risk values for each objective and reviewed minimum and maximum values for this application based on past data and similar application in this domain.

*5.1.2 Metric Function Outcomes for Enterprise Infrastructure Monitoring Software*

In this experiment the product is using the software metrics-based process for the first time. Based on the information collected from the developers for this product, Algorithm 1 is utilized to calculate the data that will result in predicted risk values for each objective.

The experiment began with collecting the actual attributes values for the EIM software and normalizing each attribute value to a positive range 1 to 5. The Min/Max formula was used to normalize each value.

$$x_i = \left( \frac{a_i - a_{min}}{a_{max} - a_{min}} \right) \times \left( r_{max} - r_{min} \right) + r_{min}$$

The normalized attribute values are collected in Table 20, which provided the attribute values and normalized attribute values. Column 1 – Attribute Names are the name of each attribute for this application. Column 2 – Max is the maximum values this application based on past data and other applications in the industry. Column 3 – is the actual attribute value collected from the software manufacturer using the Enterprise Software Profile in Appendix A. Column 4 – is the normalized attribute value after applying the Min/Max formula.

**Table 20. Enterprise Monitoring Software Maximum Values And Normalization Data Set**

| Attribute Name | Max | Value | Normalized |
|---|---|---|---|
| amount of source code | 5E+07 | **20000000** | 2 |
| amount of threads generated | 1500 | **834** | 3 |
| changes to dynamic data | 100 | **20** | 1 |
| class inheritances | 200 | **60** | 2 |
| code locale | 15 | **3** | 1 |
| CPU utilization | 100 | **90** | 5 |
| dynamic link library requirements | 1000 | **800** | 4 |
| firewall port required. | 20 | **15** | 4 |
| fonts effect on UI | 5 | **3** | 3 |
| hardware and software requirements | 10 | **10** | 5 |
| incoming client service request | 900 | **420** | 3 |
| languages supported | 20 | **9** | 3 |
| message boxes | 2100 | **160** | 1 |
| multiple system requirements | 10 | **3** | 2 |
| multitask command buttons | 200 | **13** | 1 |
| network connections required | 7 | **4** | 3 |
| open API | 9 | **3** | 2 |
| physical processors required | 32 | **4** | 1 |
| program switches lacking default clause | 30 | **7** | 2 |
| screen traversal | 15 | **7** | 3 |
| screens affected by font adjustments | 500 | **170** | 2 |
| software wizards | 30 | **1** | 1 |
| special fonts | 10 | | 0 |
| special hardware requirements | 10 | **0** | 0 |
| special input devices | 10 | **3** | 2 |
| special user accounts required. | 7 | **5** | 4 |
| UI responses required | 100 | **48** | 3 |
| unrestricted text fields | 1500 | **320** | 2 |
| use case totals | 40 | **7** | 1 |
| user interface complexity | 2000 | **950** | 3 |
| user interface controls | 1400 | **295** | 2 |

| Attribute Name | Max | Value | Normalized |
|---|---|---|---|
| user-levels | 25 | **7** | 2 |
| web portals required | 12 | **1** | 1 |
| web service request requirements | 15 | **0** | 0 |

After normalizing the attribute values, the weight initialization function is used to surmise weights for this product. The developers, based on empirical evidence, provide the estimated weights that were converted into tangible weights.

The calculations for this project are outlined in Appendix B1. Column 1 – is the formal attribute name for the software in test. Column 2 – E. Weight - is the estimated (guessed) weight provided by the beta test manager based on the potential for problem in production. Column 3 – Weight - is the calculated weight that was obtained using the following formula:

$$W_i = \frac{y_i}{\left( \frac{\sum_{i=1}^{n} y_i}{n} \right)}$$

Column 4 – N. Value is the normalized values calculated earlier. And Column 5 – Total is the metric function for the specific objective providing the predicted risk value.

$$objective\_metric\_value = \sum_{i=1}^{n} w_i x_i$$

The objective risk functions were used to calculate risk prediction for each objective (results outline in Table 21). The risk values demonstrate a priority to software vulnerability (3.35), environmental dependency (3.27), and function coverage completeness objectives (3.11) all indicating that the level of risk for these objectives is average. The UI accessibility function was not used in this experiment because the product does not provide custom accessibility features relying on the operation platform to provide and manage these attributes. The predictions provided by the objective functions will assist with building the beta testing design for the current project.

Table 21. Predicted Risk Value For Infrastructure Monitoring Software

| Environmental Dependency | Function Coverage Completeness | Localization | Robustness | Software Vulnerability | UI Accessibility | UI Usability |
|---|---|---|---|---|---|---|
| 3.273 | 3.111 | 2.250 | 1.912 | 3.353 | 0 | 2.125 |

This experiment closely matched actual risk levels provided by the beta test director. The actual risk values (ARV) for each objective was based on results of the actual beta test. The values were:

- Environmental Dependency = 4
- Function Coverage Completeness = 3
- Localization = 3
- Robustness = 3
- Software Vulnerability = 4
- UI Usability = 3

Prediction of risk values matures with experience, as stated earlier in this study. A demonstration of the training process for this product is discussed in Section 5.1.4.

### *5.1.3 Applying Beta Testing Design to Enterprise Infrastructure Monitoring Software*

There are five steps in the beta testing design and the predicted values guides in carefully planning this activity. In this scenario, the major focus is validating and eliminating the potential for software vulnerabilities, testing environmental dependencies and validating user inference. Following the steps of effective beta testing design, the predictions suggest:

The risk values for environmental dependency (3.27) and software vulnerability (3.35) suggest a large majority of the beta testers for this application should have engineering backgrounds with experience in enterprise management applications. Engineers must have administrator privileges in the current environment to test software dependency, firewall ports, network connection, and other connectivity features. In addition, the engineers must provide feedback on the function coverage of the application and make suggestions on areas of improvement.

**Step 1. Identify beta testing priority clientele groups**

In step one, the PRV was reviewed for each objective and focus was devoted to the environmental dependency, function coverage completeness, software vulnerability, and localization. To best prepare the for the current beta test, Table 18 is used to map the required types of clientele required which are (in order of importance)

a. Experienced user group (environmental dependency, function coverage completeness, and software vulnerability)
b. System engineers (environmental dependency and software vulnerability)
c. Regional Experience user group (localization)
d. Regional new user group (localization)
e. New user group (function coverage completeness)

Fundamentally, the openness of this application sparks a great concern for the security of environment when using a tool that requires an immense level of communication to function optimally. Additionally, the large level of dependencies, such as the CPU utilization (peeks to 90%), creates an elevated risk in production, and priority to these issues is important. Priority must also focus on environmental dependency related activity providing attention to attributes in this objective yielding a normalized value greater than 3. Special environmental and security validation is required for the number of firewall ports, network connections, special user accounts, etc. The large number of unrestricted test fields (320) impact the function coverage completeness objective and may create issues with new and experienced users, and a small set of regional based beta testers are required to support localization objective initiatives.

**Step 2. Identify priority functions/GUI components**

The second step in the beta testing design process, the PRV levels for software vulnerability, environmental dependency, and function objectives suggest priority for the following components:

*Environmental dependency*

- Product installation
- Benchmark product for degradation issues (e.g. bottlenecks, high utilization)
- Validate access to resources (e.g. resources, sub networks, etc).
- Web services are functional

*Software vulnerability*

- User access levels are secure
- Application access to bridges and routers are not creating vulnerabilities – currently 15 ports are required for optimal operation of this product
- Network protocols are encrypting data correctly and not exposing the current infrastructure – a large number of different network connections are required

Function Coverage Completeness

- Software wizards

- Graphical user interface is comprehensive there are high number of unrestricted text fields
- Test each product use case

The application provided risk values suggesting component priority must focus on security attributes, such as reviewing whether the firewall requirements creates additional security risk.  Also, attention must be devoted to validating whether the amount of required special user accounts create security vulnerabilities possibly violating infrastructure access standards.  There are also environmental dependencies and function coverage completeness attributes that require special attention because normalized values were high.

**Step 3.  Design questionnaires for controlling beta testing priorities**

The third step of the beta test design process is building a testing questionnaire to assist in guiding testing priorities.  In this step of the process, focus is on the environmental dependency, software vulnerability, and function coverage completeness first.  Secondary focus must validate other objectives.  Based on the importance of issues, this step requires questions in the following areas:

Environmental Dependency

- Product installation: Capture user experiences with installation of the product
- Usage of Advanced Features: Suggest information on advanced features

- Utilization: Did the product introduce any high utilization issues
- Bottlenecks: Did the product introduce degradation to the current system or network
- Access to Resources: Did the product gain access to additional resources, networks, or databases with ease?

Function Coverage Completeness:

- Software wizards:  Do the software wizards yield the appropriate outcomes?  Or do software wizards assist in easing the experience with the application?
- Graphical user interface:  Are the screens providing the appropriate amount of information to receive the desired outcome? Did the end-user find many unrestricted fields to be cumbersome?
- Use Case:  Are the product specifications accurate and complete?

Software Vulnerability:

- User Access:  Do the user level requirements create vulnerabilities in the product?  Are the access levels secure?
- Router/Bridge Access:  Does the number of port requirements create software and/or network vulnerabilities?
- Network Connections:  Do network protocols communicate correctly?  Is the level of network protocols required creating network vulnerabilities or violations?

Additional questions may be added based on level and experience with the application to enhance the questionnaire.

Design of the questionnaires is critical in any beta test design. The predicted values assist in providing signs of vulnerabilities in the application that leads to guidance in areas of significance. In addition, the predicted values provide a strategy for promoting feedback in important areas. In this case, risk values suggest building a questionnaire to spawn feedback in the areas of weakness in security and dependencies. In addition, the questionnaire must assure the product meets the customers' expectations, strategically requesting feedback on the design and operation of the software in test. A sample questionnaire is provided in Chapter 4, Section 4.4 (An Example of the Beta Testing Design) on page 172.

**Step 4. Predict minimal beta testing clientele size**

The fourth step of this methodology uses all PRV for each objective in the current project to predict minimum beta testing clientele size for a successful test. However, past data improves the accuracy of determining minimum size. However in this case we use the formula:

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

$$\bar{X} = (3.27 + 3.11 + 2.25 + 1.91 + 3.35 + 0 + 2.13)/7$$
$$\bar{X} = (16.02)/7$$
$$\bar{X} = 2.28$$
or ***2***

Use the following matrix to decide the minimum beta testing clientele size.

| Average Risk Level | Current user Base |
|---|---|
| 1 | 1% |
| 2 | 4% |
| 3 | 6% |
| 4 | 8% |
| 5 | 10% |

This is just a base set of data. The actual percentages will change with experience and past data.

In this scenario, the minimum client size prediction is impacted by the predicted values for each objective. The values predict 4% of the total user base assures the product is being tested by the optimal number of engineers to enrich the feedback required for a product of this size. However, past data plays an important role in understanding the size requirements and helps build on the success of future predictions. In this experiment, the risk values suggest using beta testers with current software experience making it easier to limit the types of clients to testers with engineering experience. A percentage of the current user base may be targeted to participate in this beta testing process. If future risk values yield similar results, the current test will assist with providing a more precise size for other tests.

**Step 5. Predict minimal beta testing duration**

The fifth step collects the PRV for each objective to estimate the minimum beta testing duration. The values are matched with matrix provided below to obtain the estimated weeks per objective. The estimated weeks for each objective are added to get the total minimum of weeks for the entire project.

| Predicted Risk Level | Minimum Weeks |
|---|---|
| 1 | 2 |
| 2 | 2.5 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |

| | | | |
|---|---|---|---|
| Environmental dependency | *3.27* | 3 | |
| Function coverage completeness | *3.11* | 3 | |
| Localization | *2.25* | 2.5 | |
| Robustness | *1.91* | 2.5 | |
| Software Vulnerability | *3.35* | 3 | |
| UI Accessibility | *0* | 0 | |
| UI Usability | *2.13* | 2.5 | |
| | | 16.5 | weeks |

The estimated average time to test a product of this size is 16.5 weeks or approximately 4 months. However, past data assist with better predictions of time and manufacturers must pay close attention to results from previous tests to manage future tests.

Here, the risk level predictions provide a foundation for adapting the beta test to better manage and improve the process through the metrics.

### *5.1.4 Applying the Function Training Process to Enterprise Infrastructure Monitoring Software*

The training process is essential in the model, learning from past data. This example demonstrates how the learning process improves the prediction by adjusting the weights to conform to historical data. The past data is adjusted by taking a percentage of the past weights from the completed projects and adding or subtracting the answer by a percentage of the normalized value of the current project.

Prior to using the training process, the predicted risk value (PRV) from the previous beta test must be compared to the actual risk value (ARV) determined at the end of the previous project. In this case, an interview was conducted with the beta test manager of this product to validate the actual risk value was accurate based on the outcome of the past beta test. The results of the actual risk values are:

| **Objective** | **PRV** | **ARV** |
|---|---|---|
| Enterprise Dependency | 3.27 | **4** |
| Function Coverage Completeness | 3.11 | **3** |
| Localization | 2.25 | **3** |
| Robustness | 1.91 | **3** |
| Software Vulnerability | 3.35 | **4** |
| UI Usability | 2.13 | **3** |

To train the weights, the learning function formula $y_i = r \bullet w_i \pm (1 - r)(\frac{x_i}{5})$ is used. In this scenario r = .95, representing past weights. The two formulas are applied based on the comparison of PRV to ARV. If PRV > ARV then Formula A

- ***$y_i$ = .r.$w_i$ – (1 – r)($x_i$ / 5)*** is used. In this scenario, this applies to the function coverage completeness objective. If PRV < ARV then

Formula B - ***$y_i$ = .r.$w_i$ + (1 – r)($x_i$ / 5*** is used to adjust weights for the current project, which applies to all other objectives.

The outcome of these formulas is provided in Appendix B3. Column 1 – Attribute Name is the name of the actual software attribute. Column 2 – is the outcome of applying either formula A or Formula B.

After calculating the trained weights for the current project, the values are scaled up or down so that the sum of all weights equal one. The following formula was used to scale the weights:

$$W_i = \frac{y_i}{\left(\frac{\sum_{i=1}^{n} y_i}{n}\right)}$$

The outcomes are in Column 3 – A. Weight, which represents the adjusted weight, in Appendix B3. After applying the adjusted weights to the correct objective function, the predicted risk values are calculated, applying the applicable objective metric functions. The calculation totals are outlined in Column 5 – Totals representing the results of using the correct objective metric function. The last column in each objective represents the experienced predicted risk value.

This experiment demonstrated an improvement in the prediction of risk value for all objectives, with the prediction for software vulnerability and localization objectives demonstrated the closest to actual value (see Table 22). This learning process continues to improve over time and the weights become more stable with each application of these functions.

Table 22. Comparison of Risk Values For EM Software – Training Process

| | Environmental Dependency | Function Coverage Completeness | Localization | Robustness | Software Vulnerability | UI Usability |
|---|---|---|---|---|---|---|
| ARV | 4 | 3 | 3 | 3 | 4 | 3 |
| **PRV** | **3.314** | **3.054** | **2.408** | **1.953** | **3.879** | **2.138** |

## 5.2 Enterprise Virtualization Monitoring Software

This section presents the outcome of the second real-world experiment. The second enterprise product is a virtualization monitoring application used to support enterprise virtualization infrastructure, virtual private networks, SAN's. Virtualization involves combining software, network, and hardware resources to emulate a computer system. The concept allows end-users to get the most value out of a computer system without focusing on special implementation, physical location, and presentation[45].

### *5.2.1 Product Overview*

The second test was conducted on an enterprise virtual infrastructure management application designed to support corporate virtualization projects and infrastructure. Virtualization supports a technology shift to an on-demand computing trend. This trend offers corporations the capability of multi-execution environments, resource sharing, and system emulation.

This product optimizes virtual resources providing dynamic reconfiguration of resources, utilization monitoring, managing dynamic resources, automatic discovery of resource utilization, business process mapping, and session policy configuration. Dynamic reconfiguration is the process of reallocation onboard resources in Sun environments. This process will detach and reattach resources without requiring a system reboot. The product also supports utilization monitoring, providing metrics to a single interface, which may be accessed via a web interface. The application automatically manages dynamic resources in real time, allocating resources when services are required. In addition to managing resources, the application discovers new dynamic resources and provides instant metrics (both hardware and software). Resource allocation is managed by business process maps to assure the correct amount of resources are available for a business unit. The application is also intelligent enough to manage allocation of resources by custom policies.

This product was designed to support out-of-the-box functionality providing seamless installation and integration. The product requires a Windows NT Based, UNIX, LINUX servers. Additional proprietary agents are required to manage monitoring and alerts. This product is designed to support enterprise-level organizations with an approximate user base of 300 end-users for a single installation. The product is also developed to support nine different foreign languages and countries.

The data for this application was collected from a group of software developers responsible for full life-cycle development. The development director verified the information and interview was conducted to review the information. This product had just completed a 3-month beta testing process sent out to 30 different beta testing sites.

*5.2.2 Metric Function Outcomes for Enterprise Virtualization Monitoring Software*

Using the same approach and methodology identical to the product in Section 5.1.2, this product is using the software metrics-based process for the first time. Based on the attribute information provided by the software development team, Algorithm 1 is employed to calculate the data that will result in predicted risk values for each objective.

The experiment began with collecting the actual attributes values for the current project and normalizing each attribute value to a positive range 1 to 5. The Min/Max formula was used to normalize each value.

$$x_i = \left( \frac{a_i - a_{min}}{a_{max} - a_{min}} \right) \times \left( r_{max} - r_{min} \right) + r_{min}$$

Again, the normalized attribute values are collected in Table 23, which provided the attribute values and normalized attribute values. Column 1 – Attribute

Names are the official attribute name for the application in test. Column 2 – Max is the maximum values of this application based on past data and other applications in the industry. Column 3 – is the actual attribute value collected from software manufacturer using the Enterprise Software Profile in Appendix A. Column 4 – is the normalized attribute value after applying the Min/Max formula.

**Table 23. Enterprise Virtualization Monitoring Software Maximum Values and Normalization Data Set**

| Attribute Name | Max | Value | Normalized |
|---|---|---|---|
| amount of source code | 3E+07 | **3000000** | 1 |
| amount of threads generated | 750 | **395** | 3 |
| changes to dynamic data | 50 | **0** | 0 |
| class inheritances | 100 | **60** | 3 |
| code locale | 7 | **3** | 3 |
| CPU utilization | 100 | **60** | 3 |
| dynamic link library requirements | 500 | **80** | 1 |
| firewall port required. | 20 | **8** | 2 |
| fonts effect on UI | 5 | **3** | 3 |
| hardware and software requirements | 10 | **3** | 2 |
| incoming client service request | 900 | **140** | 1 |
| languages supported | 20 | **9** | 3 |
| message boxes | 1200 | **80** | 1 |
| multiple system requirements | 5 | **2** | 2 |
| multitask command buttons | 100 | **9** | 1 |
| network connections required | 7 | **4** | 3 |
| open API | 5 | **1** | 1 |
| physical processors required | 4 | **2** | 3 |
| program switches lacking default clause | 15 | **7** | 3 |
| screen traversal | 10 | **5** | 3 |
| screens affected by font adjustments | 500 | **70** | 1 |
| software wizards | 30 | **9** | 2 |
| special fonts | 10 | **0** | 0 |
| special hardware requirements | 10 | **0** | 0 |
| special input devices | 10 | **0** | 0 |
| special user accounts required. | 7 | **5** | 4 |
| UI responses required | 50 | **24** | 3 |
| unrestricted text fields | 750 | **78** | 1 |

| Attribute Name | Max | Value | Normalized |
|---|---|---|---|
| use case totals | 20 | **3** | 1 |
| user interface complexity | 1000 | **380** | 2 |
| user interface controls | 600 | **150** | 2 |
| user-levels | 10 | **4** | 2 |
| web portals required | 12 | **1** | 1 |
| web service request requirements | 15 | **2** | 1 |

After normalizing the attribute values, the first experiment for this product is to initialize the weights for this product. The developers provided estimated weights, which then were converted into real numbers for use with objective functions.

The calculations for this project are outlined in Appendix C2. Column 1 – is the formal attribute name. Column 2 – E. Weight - is the estimated weight provided by the beta test manager based on the potential for problems in production. Column 3 – Weight - is the calculated weight that was obtained using the following formula:

$$W_i = \frac{y_i}{\left( \frac{\sum_{i=1}^{n} y_i}{n} \right)}$$

Column 4 – N. Value is the normalized values calculated earlier. Column 5 – Total is the metric function for the specific objective providing the predicted risk value.

$$objective\_metric\_value = \sum_{i=1}^{n} w_i x_i$$

After employing the appropriate objective metric value function, the results predicted an average potential of problems for activities in the software vulnerability objective (2.65). Other moderate problems were predicted in the localization objective and environmental dependency objectives (Outlined in Table 24).

Table 24. Predicted Risk Values For Virtualization Monitoring Software

| Environmental Dependency | Function Coverage Completeness | Localization | Robustness | Software Vulnerability | UI Accessibility | UI Usability |
|---|---|---|---|---|---|---|
| 2.200 | 1.778 | 2.500 | 1.931 | 2.637 | 0 | 1.700 |

Again, this experiment closely matched actual risk levels provided by the beta test director. The actual risk values (ARV) for each objective was based on results of the actual beta test. The values were:

- Environmental Dependency = 3
- Function Coverage Completeness = 2
- Localization = 3
- Robustness = 3
- Software Vulnerability = 3
- UI Usability = 3

A demonstration of the training process for this product is discussed in Section 5.2.4.

### *5.2.3 Applying Beta Testing Design to Enterprise Infrastructure Monitoring Software*

The PRV obtained for this project is applied to the beta testing design to assist in planning the pending beta test. Because of the experience of this product, there are no objectives providing the prediction of significant issues in production. However, there are few objectives that require special attention, which are the software vulnerability (2.65), localization (2.50), and environmental dependency (2.20) objectives. This indicates that testing priority should concentrate on validating potential security issues and testing the product in global environments.

**Step 1. Identify beta testing priority clientele groups**

In step one, the PRV was reviewed for each objective and focus was devoted to the environmental dependency, software vulnerability, and localization. To best prepare for the current beta test, Table 18 is used to map the required types of clientele required which are (in order of importance)

(1) Experienced user group (environmental dependency, function coverage completeness, and software vulnerability)

(2) System engineers (environmental dependency and software vulnerability)

(3) Regional experience user group (localization)

(4) Regional new user group (localization)

(5) New user group (function coverage completeness)

**Step 2. Identify priority functions/GUI components**

The second step in the beta testing design process, the PRV levels for software vulnerability, environmental dependency, and function objectives suggest priority for the following components:

*Environmental dependency*

- Product installation
- Benchmark product for degradation issues (i.e. bottlenecks, high utilization)
- Validate access to resources (e.g. resources, sub networks, etc).
- Web services are functional

*Software vulnerability*

- User access levels are secure
- Application access to bridges and routers are not creating vulnerabilities – currently 15 ports are required for optimal operation of this product
- Network protocols are encrypting data correctly and not exposing the current infrastructure – a large number of different network connections are required

Localization - the following product functions require validation:

- Font changes maintains organization of all forms
- Language changes are effective
- Special devices are operable
- Time/Country/Currency changes are functioning

**Step 3. Design questionnaires for controlling beta testing priorities**

The third step of the beta test design process is building a testing questionnaire to assist in guiding testing priorities. In this step of the process, focus is on the environmental dependency, software vulnerability, and function coverage completeness first. Secondary focus must validate other objectives. Based on the importance of issues, this step requires questions in the following areas:

Environmental Dependency

- Product installation: Capture user experiences with installing the product
- Usage of Advance Features: Suggest information on advance features
- Utilization: Did the product introduce any high utilization issues
- Bottlenecks: Did the product introduce degradation to the current system or network
- Access to Resources: Did the product gain access to additional resources, networks, or databases with ease?

Software Vulnerability :

- User Access: Do the user level requirements create vulnerabilities in the product? Are the access levels secure?
- Router/Bridge Access: Does the number of port requirements create software and/or network vulnerabilities?
- Network Connections: Do network protocols communicate correctly? Is the level of network protocols required to create network vulnerabilities or violations?

Localization:

- Font Change: Do the character changes for desired country maintain organization to all screens?
- Language Support: Is the desired language correct and effective?
- Special Localized Peripherals: Do the specially mapped peripherals operate effectively?
- Conversion Changes: Does the Time/Country/Currency provide accurate translations?

Additional questions may be added based on level and experience with the application to enhance the questionnaire. A sample questionnaire is provided in Chapter 4, Section 4.4 (An Example of the Beta Testing Design) on page 172.

**Step 4. Predict minimal beta testing clientele size**

The fourth step of this methodology, uses all PRV for each objective in the current project to predict minimum beta testing clientele size for a successful test. However, past data improves the accuracy of determining minimum size. However in this case we use the formula:

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

$\bar{X} = (2.2 + 1.78 + 2.5 + 1.93 + 2.65 + 0 + 1.7)/7$

$\bar{X} = (12.76)/7$

$\bar{X} = 1.82$

or ***2***

Use the following matrix to decide the minimum beta testing clientele size.

| Average Risk Level | Current user Base |
|---|---|
| 1 | 1% |
| 2 | 4% |
| 3 | 6% |
| 4 | 8% |
| 5 | 10% |

This is just a base set of data. The actual percentages will change with experience and past data.

In this scenario, the minimum client size prediction is impacted by the predicted values for each objective. The values predict a minimum of 4% of the total user base to assure the products beta test provides optimal acceptance by the current user group.

**Step 5. Predict minimal beta testing duration**

The fifth step collects the PRV for each objective to estimate the minimum beta testing duration. The values are matched with matrix provided below to obtain the estimated weeks per objective. The objective risk values are rounded to the nearest whole number to match predicted risk level with the suggested minimum weeks per objective. The estimated weeks for each objective are added to get the total minimum of weeks for the entire project.

| Predicted Risk Level | Minimum Weeks |
|---|---|
| 1 | 2 |
| 2 | 2.5 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |

| | | |
|---|---|---|
| Environmental dependency | *2.20* | 2.5 |
| Function coverage completeness | *1.78* | 2.5 |
| Localization | *2.50* | 3 |
| Robustness | *1.93* | 2.5 |
| Software Vulnerability | *2.65* | 3 |
| UI Accessibility | *0* | 0 |
| UI Usability | *1.70* | 2.5 |
| | | 16 weeks |

The estimated average time to test a product of this size is 16 weeks or approximately 4 months. However, past data assist with better predictions of time and manufacturers must pay close attention to results from previous tests to manage future tests.

### *5.2.4 Applying the Function Training Process to Enterprise Virtualization Monitoring Software*

Again, the training process is employed to demonstrate how the weights are stabilized and the functions improve the overall predictions. Here the objective risk value predictions are compared to actual risk provided by the results, feedback and information provided after the actual beta test. In this case, developers closely reviewed feedback from the users issues, and enhancements to provide the following actual risk values:

| **Objective** | **PRV** | **ARV** |
|---|---|---|
| Enterprise Dependency | 2.20 | **3** |
| Function Coverage Completeness | 1.78 | **2** |
| Localization | 2.50 | **3** |
| Robustness | 1.93 | **3** |
| Software Vulnerability | 2.65 | **2** |
| UI Usability | 1.70 | **3** |

To train the weights, the learning function formula $y_i = r \bullet w_i \pm (1 - r)(\frac{x_i}{5})$ is used. In this scenario r = .95, representing past weights. The two formulas are applied based on the comparison of PRV to ARV. If PRV > ARV then Formula A - ***$y_i = .r.w_i - (1 - r)(x_i / 5)$*** is used. In this scenario, this applies to the software vulnerability objectives.

If PRV < ARV then Fomula B - ***$y_i = .r.w_i + (1 - r)(x_i / 5)$*** is used to adjust weights for the current project, which applies to all other objectives.

The outcome of these formulas is provided in Appendix C3. Column 1 – Attribute Name is the actual attribute title. Column 2 – is the outcome of applying either formula A or Formula B.

After calculating the trained weights for the current project, the values are scaled up or down so that the sum of all weights equal one. The following formula was used to scale the weights:

$$W_i = \frac{y_i}{\left( \frac{\sum_{i=1}^{n} y_i}{n} \right)}$$

The outcomes are in Column 3 – A. Weight, which represents the adjusted weight, in Appendix C3. After applying the adjusted weights to the correct objective function, the predicted risk values are calculated, applying the applicable objective metric functions. The calculation totals are outlined in Column 5 – Totals representing the results of using the correct objective metric function. The last column in each objective represents the experienced predicted risk value.

Again, this experimented confirmed improvement in the prediction of risk value for all objectives, with the prediction for software vulnerability and localization objectives demonstrated the closest to actual value (outlined in Table 25). This learning process continues to improve over time and the weights become more stable with each application of these functions.

Table 25. Comparison of Risk Values after Training Process

| | Environmental Dependency | Function Coverage Completeness | Localization | Robustness | Software Vulnerability | UI Usability |
|---|---|---|---|---|---|---|
| ARV | 3 | 2 | 3 | 3 | 3 | 3 |
| **PRV** | **2.283** | **1.803** | **2.472** | **1.986** | **2.625** | **1.764** |

## 5.3 Comparison of Current Beta Testing Design Process

Both products use similar methodologies when testing enterprise products. The current product utilizes a beta testing process that is a standard software improvement template. The process is used for every product in beta regardless of class. There are a set amount of clients required for each product, the questionnaire template focuses more on generic quality of service than seeking to solicit feedback based on the potential of problems in the test product. In addition, the current questionnaire has a major marketing focus. Time constraints are based on the amount of users actually participating in the program; since the product stays in beta until they have the required amount of users enrolled in the beta test. Another factor impacting time is the amount of “new” features in the product.

The beta testing design process in this study is far better for process improvements because it uses the current attributes of the application to expose weakness. This method provides a more customized beta testing, allowing the manufacturer to streamline areas of improvement, guide feedback in weak areas in the product, better measure test time durations, increase the amount and class of beta testers required, and gauge how to better manage feedback. In addition, the beta testing design process learns and adjusts to historical data in a more structured manner. This process actually improves the application beta testing process based on principles of the application and not a global standard template.

# Chapter 6

# Conclusion

The results of this study demonstrated how measuring software attributes of a finished enterprise application support the design of a beta test. The metric functions used in this study provided close risk predictions for each major objective and the functions demonstrated the ability to learn with experience, which is useful in providing closer predictions. The process serves as a major contribution in the improvement of software processes and the quality of enterprise applications. In addition, the objectives, attributes, and design components are valuable in channeling focus in the important areas of an application and assisting with prioritizing goals in a beta test practice.

## 6.1 Major Contributions

Today's beta testing methodologies only focus on the size of beta testing clientele, duration of the beta test, and the number of errors in the application. This study provides a quantitative approach to this problem. The methodology reveals the potential risk for a given sector of the application allowing adjustments to improve the testing process, leading to a quality application and better management of time and resources. The methodology provides seven beta testing objectives and a focused set of software validation goals. Each objective contains software attributes that highlight principle parts of the software that have the potential to create issues in a production environment. Software metric functions are used to calculate predicted risk values and a set of training metrics are used to demonstrate how the functions learn with experience. The training process improves predictions over time.

Another major contribution is the beta testing design. The design process provides a framework for planning a successful beta test. The beta testing design uses software metric functions to influence the five-step process. In addition, the beta testing design provided in this study improves with frequent use because of the impact of past data.

## 6.2 Limitations of the Study

There are several limitations in this study. These limitations will influence the maturity of this model with future research. The software metrics-based model supports only enterprise level software, which are applications designed to for a business entity requiring little to no modifications. The model was also designed for beta testing, not other forms of end-user product validation testing. The results of this study were limited to a small sample to test the efficiency of predicted risk values to influence beta testing design.

The beta testing design model in this study supports enterprise level applications, which are outlined in Section 3.1 of this dissertation. Beta testing is more common in the business community because of the use of enterprise level applications. There are other applications that could benefit from this model such as open source applications, customized educational and business applications, and system-based software.

This study focuses on beta testing as the most robust form of end-user software validation for enterprise level applications.  Other forms of end-user software validation such as agile testing and maturity models were not used to influence this study.  Agile testing focuses on agile development projects and maturity models concentrate on, but are not limited to, government and military process improvements.

This study provides only a framework to design a beta test.  The framework is proven to be effective, but only a sample of three products were utilized to demonstrate the effectiveness of the model.  Each limitation will impact the development of this model and improve the value of properly planning a beta test.

## 6.3 Future Works

There are several recommended projects that will extend this study and impact future research of software metrics-based beta design modeling.  Since this model was limited to a few enterprise testing samples, a more longitudinal study is recommended to prove the effectiveness of predicted risk values in beta testing design.  This requires implementing this methodology on different products and application domains over a finite period.  A larger study will improve the effectiveness of the functions and affects the accuracy of minimum and maximum attribute values.

After the process has matured, other objectives may be introduced to the process. Also, the function could be expanded to raise the range of the risk levels to illustrate more diversity in the perception of issues in an objective over a period of time. Future works may also integrate agile development projects to demonstrate the importance of agile testing as a compliment of beta testing. In addition, future studies may test the effectiveness of the model on open-source, customized business/education software, and system specific applications.

# Appendix A

## Sample Enterprise Software Profile

### ENTERPRISE SOFTWARE PROFILE

| | |
|---|---|
| **Product Name:** | |
| **Product Version:** | |
| **Date:** | |

| **Product Description** |
|---|
| |

| | |
|---|---|
| **Objective Name:** | Environmental Dependency |

| **Description** |
|---|
| The focus of the environmental dependency objective is to test the software reliance on hardware, shared software components, and other supplemental software to validate the required dependency does not impede operation. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Environmental Dependency | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of threads generated | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total amount of CPU utilization | ☐ | ☐ | ☐ | ☐ | ☐ | |

| | |
|---|---|
| **Objective Name:** | Function Coverage Completeness |

| **Description** |
|---|
| The focus of the function coverage completeness objective is to validate that the software in the beta testing cycle meets the customer expectations. This objective validates user inference, which is the essence of beta testing. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Function Coverage Completeness | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of hardware and supplemental software requirements | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of software wizards | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of unrestricted text fields | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of use case(s) | ☐ | ☐ | ☐ | ☐ | ☐ | |

| **Objective Name:** | Localization |
|---|---|

| **Description** |
|---|
| Enterprise software adjusted to function in foreign countries are properly localized. However, the extent of localization is managing language and conversation changes in the application. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Localization | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of code locales | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of fonts effecting UI | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of languages supported | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of special input devices required | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of screens affected by font adjustments | ☐ | ☐ | ☐ | ☐ | ☐ | |

| **Objective Name:** | Robustness |
|---|---|

| **Description** |
|---|
| The focus of the robustness objective is to identify incorrect data and how user errors and usage patterns impact software. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Robustness | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total lines of source code | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of threads generated | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of changes to dynamic data | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of class inheritances | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of multitask command buttons | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of program switches lacking default clause | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total software use cases | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of unrestricted text fields | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of input fields and command buttons on a user interface (UI complexity) | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of end-user accounts and/or user stack levels required (user-levels) | ☐ | ☐ | ☐ | ☐ | ☐ | |

| | |
|---|---|
| **Objective Name:** | Software Vulnerabilities |

| **Description** |
|---|
| The software vulnerability validation objective measures the application to exploit potential security violations focusing on vulnerabilities in communication. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Software Vulnerabilities | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of firewall ports required | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of network connections required | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of open API(s) required | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of special user accounts required | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of web portals required | ☐ | ☐ | ☐ | ☐ | ☐ | |

| | |
|---|---|
| **Objective Name:** | UI Accessibility |

| **Description** |
|---|
| The User interface accessibility (UIA) objective of beta testing validates the features of an application designed to assist end-users with special physical needs. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| UI Accessibility | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of special fonts required | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of special hardware requirements | ☐ | ☐ | ☐ | ☐ | ☐ | |

| | |
|---|---|
| **Objective Name:** | UI Usability |

| **Description** |
|---|
| This objective focuses on validating that the graphical user interface is simple and promotes ease of use. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| UI Usability | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of message boxes | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of multitask command buttons | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of screen traversals | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of software wizards | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of message boxes requiring response (or action) from user | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of user interface controls (i.e. requires manipulation by user such as control buttons, slide bars, etc.) | ☐ | ☐ | ☐ | ☐ | ☐ | |
| total number of unrestricted text fields | ☐ | ☐ | ☐ | ☐ | ☐ | |

# Appendix B1

## Enterprise Software Profile for Enterprise Management Software

### ENTERPRISE SOFTWARE PROFILE

| | |
|---|---|
| **Product Name:** | **Enterprise Infrastructure Monitoring Software** |

| | |
|---|---|
| **Product Version:** | **12.1** |

| | |
|---|---|
| **Date:** | **March 12, 2005** |

| **Product Description** |
|---|
| The software is a robust application that manages corporations network resources, providing a large number of system services to assist in better managing the network. |

| | |
|---|---|
| **Objective Name:** | Environmental Dependency |

| **Description** |
|---|
| The focus of the environmental dependency objective is to test the software reliance on hardware, shared software components, and other supplemental software to validate the required dependency does not impede operation. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Environmental Dependency | ☐ | ☐ | ☐ | ☐ | ● | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of threads generated | ☐ | ☐ | ● | ☐ | ☐ | 834 |

| Attributes | Level of Problem Relevant to the Attribute | | | | | Value |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | 1 | 2 | 3 | 4 | 5 | |
| total amount of CPU utilization | ☐ | ☐ | ● | ☐ | ☐ | 90 |
| total number of dynamic link library files required | ☐ | ☐ | ● | ☐ | ☐ | 800 |
| total number of firewall ports required | ☐ | ☐ | ☐ | ☐ | ● | 15 |
| estimated number of incoming client service request(s) | ☐ | ☐ | ● | ☐ | ☐ | 420 |
| total number of multiple systems required | ☐ | ● | ☐ | ☐ | ☐ | 3 |
| total number of network connections required | ☐ | ☐ | ☐ | ● | ☐ | 4 |
| total number of open API(s) required | ☐ | ● | ☐ | ☐ | ☐ | 3 |
| total number of physical processors required | ☐ | ● | ☐ | ☐ | ☐ | 3 |
| total number of special user accounts required | ☐ | ☐ | ☐ | ☐ | ● | 5 |
| total number of web portals required | ● | ☐ | ☐ | ☐ | ☐ | 1 |
| total number of web service request(s) required | ☐ | ● | ☐ | ☐ | ☐ | 2 |

| **Objective Name:** | Function Coverage Completeness |
|---|---|

| **Description** |
|---|
| The focus of the function coverage completeness objective is to validate that the software in the beta testing cycle meets the customer expectations. This objective validates user inference, which is the essence of beta testing. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Function Coverage Completeness | ☐ | ☐ | ☐ | ● | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of hardware and supplemental software requirements | ☐ | ☐ | ☐ | ● | ☐ | 5 |
| total number of software wizards | ☐ | ☐ | ☐ | ● | ☐ | 4 |
| total number of unrestricted text fields | ☐ | ☐ | ● | ☐ | ☐ | 320 |
| total number of use case(s) | ☐ | ● | ☐ | ☐ | ☐ | 7 |

| **Objective Name:** | Localization |
|---|---|

| **Description** |
|---|
| Enterprise software adjusted to function in foreign countries are properly localized. However, the extent of localization is managing language and conversation changes in the application. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Localization | ☐ | ☐ | ☐ | ● | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of code locales | ☐ | ☐ | ☐ | ● | ☐ | 3 |
| total number of fonts effecting UI | ☐ | ☐ | ☐ | ☐ | ● | 3 |
| total number of languages supported | ☐ | ☐ | ☐ | ● | ☐ | 9 |
| total number of special input devices required | ☐ | ● | ☐ | ☐ | ☐ | 3 |
| total number of screens affected by font adjustments | ☐ | ☐ | ☐ | ☐ | ● | 170 |

| **Objective Name:** | Robustness |
|---|---|

| **Description** |
|---|
| The focus of the robustness objective is to identify incorrect data and how user errors and usage patterns impact software. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Robustness | ☐ | ☐ | ☐ | ● | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total lines of source code | ☐ | ☐ | ☐ | ☐ | ● | 200 |
| total number of threads generated | ☐ | ☐ | ☐ | ☐ | ● | 834 |
| total number of changes to dynamic data | ☐ | ☐ | ● | ☐ | ☐ | 20 |
| total number of class inheritances | ☐ | ☐ | ● | ☐ | ☐ | 60 |
| total number of multitask command buttons | ☐ | ☐ | ● | ☐ | ☐ | 13 |
| total number of program switches lacking default clause | ☐ | ● | ☐ | ☐ | ☐ | 7 |
| total software use cases | ☐ | ☐ | ☐ | ● | ☐ | 7 |
| total number of unrestricted text fields | ☐ | ☐ | ☐ | ● | ☐ | 320 |
| total number of input fields and command buttons on a user interface (UI complexity) | ☐ | ● | ☐ | ☐ | ☐ | 950 |
| total number of end-user accounts and/or user stack levels required (user-levels) | ☐ | ☐ | ● | ☐ | ☐ | 7 |

| **Objective Name:** | Software Vulnerabilities |
|---|---|

| **Description** |
|---|
| The software vulnerability validation objective measures the application to exploit potential security violations focusing on vulnerabilities in communication. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Software Vulnerabilities | ☐ | ☐ | ☐ | ☐ | ● | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of firewall ports required | ☐ | ☐ | ☐ | ☐ | ● | 15 |
| total number of network connections required | ☐ | ☐ | ☐ | ● | ☐ | 4 |
| total number of open API(s) required | ☐ | ● | ☐ | ☐ | ☐ | 3 |
| total number of special user accounts required | ☐ | ☐ | ☐ | ☐ | ● | 5 |
| total number of web portals required | ● | ☐ | ☐ | ☐ | ☐ | 1 |

| **Objective Name:** | UI Accessibility |
|---|---|

| **Description** |
|---|
| The User interface accessibility (UIA) objective of beta testing validates the features of an application designed to assist end-users with special physical needs. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| UI Accessibility | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of special fonts required | ☐ | ☐ | ☐ | ☐ | ☐ | 0 |
| total number of special hardware requirements | ☐ | ☐ | ☐ | ☐ | ☐ | 0 |

| Objective Name: | UI Usability |
|---|---|

| Description |
|---|
| This objective focuses on validating that the graphical user interface is simple and promotes ease of use. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| Objective Name | Actual Rating | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | 0 | 1 | 2 | 3 | 4 | 5 |
| UI Usability | ☐ | ☐ | ☐ | ● | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| Attributes | Level of Problem Relevant to the Attribute | | | | | Value |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | 1 | 2 | 3 | 4 | 5 | |
| total number of message boxes | ☐ | ☐ | ● | ☐ | ☐ | 160 |
| total number of multitask command buttons | ☐ | ☐ | ● | ☐ | ☐ | 13 |
| total number of screen traversals | ☐ | ☐ | ☐ | ☐ | ● | 7 |
| total number of software wizards | ● | ☐ | ☐ | ☐ | ☐ | 1 |
| total number of message boxes requiring response (or action) from user | ☐ | ☐ | ☐ | ☐ | ● | 48 |
| total number of user interface controls (i.e. requires manipulation by user such as control buttons, slide bars, etc.) | ☐ | ☐ | ● | ☐ | ☐ | 295 |
| total number of unrestricted text fields | ☐ | ☐ | ☐ | ● | ☐ | 320 |

## Appendix B2

## Calculations for Enterprise Management Software Weights Initialization Process

| Environmental Dependency | | | | |
|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **N. Value** | **Total** |
| amount of threads generated | 3 | 0.091 | 3 | 0.273 |
| CPU utilization | 3 | 0.091 | 5 | 0.455 |
| dynamic link library requirements | 3 | 0.091 | 4 | 0.364 |
| Firewall port required. | 5 | 0.152 | 4 | 0.606 |
| incoming client service request | 3 | 0.091 | 3 | 0.273 |
| multiple system requirements | 2 | 0.061 | 2 | 0.121 |
| network connections required | 4 | 0.121 | 3 | 0.364 |
| open API | 2 | 0.061 | 2 | 0.121 |
| physical processors required | 2 | 0.061 | 1 | 0.061 |
| special user accounts required. | 5 | 0.152 | 4 | 0.606 |
| web portals required | 1 | 0.030 | 1 | 0.030 |
| web service request requirements | 0 | 0.000 | 0 | 0.000 |
| | 33 | 1.000 | 32 | **3.273** |

| Function Coverage Completeness | | | | |
|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **N. Value** | **Total** |
| hardware and software requirements | 3 | 0.333 | 5 | 1.667 |
| software wizards | 1 | 0.111 | 1 | 0.111 |
| unrestricted text fields | 3 | 0.333 | 2 | 0.667 |
| use case totals | 2 | 0.222 | 3 | 0.667 |
| | 9 | 1.000 | 11 | **3.111** |

| Localization | | | | |
|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **N. Value** | **Total** |
| code locale | 4 | 0.200 | 1 | 0.200 |
| fonts effect on UI | 5 | 0.250 | 3 | 0.750 |
| languages supported | 4 | 0.200 | 3 | 0.600 |
| special input devices | 2 | 0.100 | 2 | 0.200 |
| screens affected by font adjustments | 5 | 0.250 | 2 | 0.500 |
| | 20 | 1.000 | 11 | **2.250** |

**Robustness**

| Attribute Name | Weight | N. Weight | N. Value | Total |
|---|---|---|---|---|
| number of lines of source code | 5 | 0.147 | 2 | 0.294 |
| amount of threads generated | 5 | 0.147 | 3 | 0.441 |
| changes to dynamic data | 3 | 0.088 | 1 | 0.088 |
| class inheritances | 3 | 0.088 | 2 | 0.176 |
| multitask command buttons | 3 | 0.088 | 1 | 0.088 |
| program switches lacking default clause | 2 | 0.059 | 2 | 0.118 |
| use case totals | 4 | 0.118 | 1 | 0.118 |
| unrestricted text fields | 4 | 0.118 | 2 | 0.235 |
| user interface complexity | 2 | 0.059 | 3 | 0.176 |
| user-levels | 3 | 0.088 | 2 | 0.176 |
|  | 34 | 1.000 | 19 | **1.912** |

**Software Vulnerability**

| Attribute Name | Weight | N. Weight | N. Value | Total |
|---|---|---|---|---|
| Firewall port required | 5 | 0.294 | 4 | 1.176 |
| network connections required | 4 | 0.235 | 3 | 0.706 |
| open API | 2 | 0.118 | 2 | 0.235 |
| special user accounts required | 5 | 0.294 | 4 | 1.176 |
| web portals required | 1 | 0.059 | 1 | 0.059 |
|  | 17 | 1.000 |  | **3.353** |

**UI Usability**

| Attribute Name | Weight | N. Weight | N. Value | Total |
|---|---|---|---|---|
| message boxes | 3 | 0.125 | 1 | 0.125 |
| multitask command buttons | 3 | 0.125 | 1 | 0.125 |
| screen traversal | 5 | 0.208 | 3 | 0.625 |
| software wizards | 1 | 0.042 | 1 | 0.042 |
| UI responses required | 5 | 0.208 | 3 | 0.625 |
| User interface controls | 3 | 0.125 | 2 | 0.250 |
| unrestricted text fields | 4 | 0.167 | 2 | 0.333 |
|  | 24 | 1.000 |  | **2.125** |

# Appendix B3

## Calculations for Enterprise Management Software Function Training Process

| Environmental Dependency | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| amount of threads generated | 0.116 | 0.092 | 3 | 0.275 |
| CPU utilization | 0.136 | 0.107 | 5 | 0.537 |
| dynamic link library requirements | 0.126 | 0.099 | 4 | 0.398 |
| Firewall port required. | 0.184 | 0.145 | 4 | 0.579 |
| incoming client service request | 0.116 | 0.092 | 3 | 0.275 |
| multiple system requirements | 0.078 | 0.061 | 2 | 0.122 |
| network connections required | 0.145 | 0.114 | 3 | 0.343 |
| open API | 0.078 | 0.061 | 2 | 0.122 |
| physical processors required | 0.068 | 0.053 | 1 | 0.053 |
| special user accounts required. | 0.184 | 0.145 | 4 | 0.579 |
| web portals required | 0.039 | 0.031 | 1 | 0.031 |
| web service request requirements | 0.000 | 0.000 | 0 | 0.000 |
| | 1.270 | 1.000 | | **3.314** |

| Function Coverage Completeness | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| hardware and software requirements | 0.267 | 0.317 | 5 | 1.587 |
| software wizards | 0.096 | 0.114 | 1 | 0.114 |
| unrestricted text fields | 0.297 | 0.353 | 2 | 0.706 |
| use case totals | 0.181 | 0.216 | 3 | 0.647 |
| | 0.840 | 1.000 | | **3.054** |

| Localization | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| code locale | 0.200 | 0.189 | 1 | 0.200 |
| fonts effect on UI | 0.268 | 0.252 | 3 | 0.803 |
| languages supported | 0.220 | 0.208 | 3 | 0.660 |
| special input devices | 0.115 | 0.108 | 2 | 0.230 |
| screens affected by font adjustments | 0.258 | 0.243 | 2 | 0.515 |
| | 1.060 | 1.000 | | **2.408** |

| Robustness | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| number of lines of source code | 0.160 | 0.140 | 2 | 0.280 |
| amount of threads generated | 0.170 | 0.149 | 3 | 0.447 |
| changes to dynamic data | 0.094 | 0.082 | 1 | 0.082 |
| class inheritances | 0.104 | 0.091 | 2 | 0.182 |
| multitask command buttons | 0.094 | 0.082 | 1 | 0.082 |
| program switches lacking default clause | 0.076 | 0.067 | 2 | 0.133 |
| use case totals | 0.122 | 0.107 | 1 | 0.107 |
| unrestricted text fields | 0.132 | 0.116 | 2 | 0.231 |
| user interface complexity | 0.086 | 0.075 | 3 | 0.226 |
| user-levels | 0.104 | 0.091 | 2 | 0.182 |
| | 1.140 | 1.000 | | **1.953** |

| Software Vulnerability | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| Firewall port required | 0.319 | 0.293 | 3 | 0.879 |
| network connections required | 0.254 | 0.233 | 5 | 1.163 |
| open API | 0.132 | 0.121 | 4 | 0.484 |
| special user accounts required | 0.319 | 0.293 | 4 | 1.172 |
| web portals required | 0.066 | 0.060 | 3 | 0.181 |
| | 1.090 | 1.000 | 2 | **3.879** |

| UI Usability | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| message boxes | 0.129 | 0.119 | 1 | 0.119 |
| multitask command buttons | 0.129 | 0.119 | 1 | 0.119 |
| screen traversal | 0.228 | 0.211 | 3 | 0.633 |
| software wizards | 0.050 | 0.046 | 1 | 0.046 |
| UI responses required | 0.228 | 0.211 | 3 | 0.633 |
| User interface controls | 0.139 | 0.128 | 2 | 0.257 |
| unrestricted text fields | 0.178 | 0.165 | 2 | 0.330 |
| | 1.080 | 1.000 | | **2.138** |

# Appendix C1

## Enterprise Software Profile for Enterprise Virtualization Monitoring Software

### ENTERPRISE SOFTWARE PROFILE

| | |
|---|---|
| **Product Name:** | **Enterprise Virtualization Monitoring Software** |

| | |
|---|---|
| **Product Version:** | **11.0** |

| | |
|---|---|
| **Date:** | **March 1, 2005** |

| **Product Description** |
|---|
| The software is a robust virtualization monitoring application used to support enterprise infrastructure, virtual private networks, on demand computing, and SAN's. |

| | |
|---|---|
| **Objective Name:** | Environmental Dependency |

| **Description** |
|---|
| The focus of the environmental dependency objective is to test the software reliance on hardware, shared software components, and other supplemental software to validate the required dependency does not impede operation. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Environmental Dependency | ☐ | ☐ | ☐ | ● | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of threads generated | ☐ | ☐ | ☐ | ● | ☐ | 395 |
| total amount of CPU utilization | ☐ | ☐ | ● | ☐ | ☐ | 60 |
| total number of dynamic link library files required | ☐ | ☐ | ● | ☐ | ☐ | 80 |
| total number of firewall ports required | ☐ | ☐ | ☐ | ● | ☐ | 8 |
| estimated number of incoming client service request(s) | ☐ | ☐ | ● | ☐ | ☐ | 140 |
| total number of multiple systems required | ● | ☐ | ☐ | ☐ | ☐ | 2 |
| total number of network connections required | ☐ | ● | ☐ | ☐ | ☐ | 4 |
| total number of open API(s) required | ☐ | ● | ☐ | ☐ | ☐ | 1 |
| total number of physical processors required | ☐ | ● | ☐ | ☐ | ☐ | 2 |
| total number of special user accounts required | ☐ | ☐ | ● | ☐ | ☐ | 5 |
| total number of web portals required | ● | ☐ | ☐ | ☐ | ☐ | 1 |
| total number of web service request(s) required | ☐ | ● | ☐ | ☐ | ☐ | 2 |

| **Objective Name:** | Function Coverage Completeness |
|---|---|

| **Description** |
|---|
| The focus of the function coverage completeness objective is to validate that the software in the beta testing cycle meets the customer expectations. This objective validates user inference, which is the essence of beta testing. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Function Coverage Completeness | ☐ | ☐ | ● | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of hardware and supplemental software requirements | ☐ | ☐ | ☐ | ☐ | ● | 3 |
| total number of software wizards | ☐ | ☐ | ☐ | ● | ☐ | 9 |
| total number of unrestricted text fields | ☐ | ☐ | ☐ | ☐ | ● | 78 |
| total number of use case(s) | ☐ | ☐ | ☐ | ● | ☐ | 3 |

| **Objective Name:** | Localization |
|---|---|

| **Description** |
|---|
| Enterprise software adjusted to function in foreign countries are properly localized. However, the extent of localization is managing language and conversation changes in the application. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Localization | ☐ | ☐ | ☐ | ● | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of code locales | ☐ | ☐ | ● | ☐ | ☐ | 3 |
| total number of fonts effecting UI | ☐ | ☐ | ☐ | ☐ | ● | 3 |
| total number of languages supported | ☐ | ☐ | ☐ | ● | ☐ | 9 |
| total number of special input devices required | ☐ | ☐ | ☐ | ☐ | ☐ | 0 |
| total number of screens affected by font adjustments | ☐ | ☐ | ☐ | ● | ☐ | 70 |

| **Objective Name:** | Robustness |
|---|---|

| **Description** |
|---|
| The focus of the robustness objective is to identify incorrect data and how user errors and usage patterns impact software. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Robustness | ☐ | ☐ | ☐ | ● | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total lines of source code | ☐ | ☐ | ● | ☐ | ☐ | 3M |
| total number of threads generated | ☐ | ☐ | ● | ☐ | ☐ | 395 |
| total number of changes to dynamic data | ☐ | ☐ | ☐ | ☐ | ☐ | 0 |
| total number of class inheritances | ☐ | ☐ | ● | ☐ | ☐ | 40 |
| total number of multitask command buttons | ☐ | ☐ | ● | ☐ | ☐ | 9 |
| total number of program switches lacking default clause | ☐ | ☐ | ☐ | ☐ | ● | 7 |
| total software use cases | ☐ | ☐ | ☐ | ● | ☐ | 3 |
| total number of unrestricted text fields | ☐ | ☐ | ● | ☐ | ☐ | 78 |
| total number of input fields and command buttons on a user interface (UI complexity) | ☐ | ☐ | ● | ☐ | ☐ | 380 |
| total number of end-user accounts and/or user stack levels required (user-levels) | ☐ | ● | ☐ | ☐ | ☐ | 4 |

| **Objective Name:** | Software Vulnerabilities |
|---|---|

| **Description** |
|---|
| The software vulnerability validation objective measures the application to exploit potential security violations focusing on vulnerabilities in communication. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| Software Vulnerabilities | ☐ | ☐ | ☐ | ● | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of firewall ports required | ☐ | ☐ | ☐ | ☐ | ● | 8 |
| total number of network connections required | ☐ | ☐ | ☐ | ● | ☐ | 4 |
| total number of open API(s) required | ☐ | ● | ☐ | ☐ | ☐ | 1 |
| total number of special user accounts required | ☐ | ☐ | ☐ | ☐ | ● | 5 |
| total number of web portals required | ● | ☐ | ☐ | ☐ | ☐ | 1 |

| **Objective Name:** | UI Accessibility |
|---|---|

| **Description** |
|---|
| The User interface accessibility (UIA) objective of beta testing validates the features of an application designed to assist end-users with special physical needs. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| UI Accessibility | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of special fonts required | ☐ | ☐ | ☐ | ☐ | ☐ | 0 |
| total number of special hardware requirements | ☐ | ☐ | ☐ | ☐ | ☐ | 0 |

| **Objective Name:** | UI Usability |
|---|---|

| **Description** |
|---|
| This objective focuses on validating that the graphical user interface is simple and promotes ease of use. |

**Step 1.**
Please provide an actual rating based on your experience with the product.

| **Objective Name** | **Actual Rating** | | | | | |
|---|---|---|---|---|---|---|
| | No Problem | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem |
| | **0** | **1** | **2** | **3** | **4** | **5** |
| UI Usability | ☐ | ☐ | ☐ | ● | ☐ | ☐ |

**Step 2.**
Please review the product attributes below. Based on your experience first provide the potential level of problems the attribute may create. Next, provide the numeric value based on what is requested in each section. (e.g. Total Number of Threads Generated = 7. This represents the maximum number of threads generated during a single instance of the application.)

| **Attributes** | **Level of Problem Relevant to the Attribute** | | | | | **Value** |
|---|---|---|---|---|---|---|
| | Low Problem | Moderate Problem | Average Problem | Significant Problem | Strong Problem | |
| | **1** | **2** | **3** | **4** | **5** | |
| total number of message boxes | ☐ | ☐ | ● | ☐ | ☐ | 80 |
| total number of multitask command buttons | ☐ | ☐ | ● | ☐ | ☐ | 9 |
| total number of screen traversals | ☐ | ● | ☐ | ☐ | ☐ | 5 |
| total number of software wizards | ● | ☐ | ☐ | ☐ | ☐ | 9 |
| total number of message boxes requiring response (or action) from user | ☐ | ☐ | ● | ☐ | ☐ | 24 |
| total number of user interface controls (i.e. requires manipulation by user such as control buttons, slide bars, etc.) | ☐ | ☐ | ● | ☐ | ☐ | 150 |
| total number of unrestricted text fields | ☐ | ☐ | ☐ | ☐ | ● | 78 |

# Appendix C2

## Calculations for Enterprise Virtualization Monitoring Software Weights Initialization Process

| Environmental Dependency | | | | |
|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **N. Value** | **Total** |
| amount of threads generated | 4 | 0.133 | 3 | 0.400 |
| CPU utilization | 3 | 0.100 | 3 | 0.300 |
| dynamic link library requirements | 3 | 0.100 | 1 | 0.100 |
| Firewall port required. | 4 | 0.133 | 2 | 0.267 |
| incoming client service request | 3 | 0.100 | 1 | 0.100 |
| multiple system requirements | 1 | 0.033 | 2 | 0.067 |
| network connections required | 2 | 0.067 | 3 | 0.200 |
| open API | 2 | 0.067 | 1 | 0.067 |
| physical processors required | 2 | 0.067 | 3 | 0.200 |
| special user accounts required. | 3 | 0.100 | 4 | 0.400 |
| web portals required | 1 | 0.033 | 1 | 0.033 |
| web service request requirements | 2 | 0.067 | 1 | 0.067 |
| | 30 | 1.000 | 25 | **2.200** |

| Function Coverage Completeness | | | | |
|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **N. Value** | **Total** |
| hardware and software requirements | 5 | 0.278 | 3 | 0.833 |
| software wizards | 4 | 0.222 | 2 | 0.444 |
| unrestricted text fields | 5 | 0.278 | 1 | 0.278 |
| use case totals | 4 | 0.222 | 1 | 0.222 |
| | 18 | 1.000 | 7 | **1.778** |

| Localization | | | | |
|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **N. Value** | **Total** |
| code locale | 3 | 0.188 | 3 | 0.563 |
| fonts effect on UI | 5 | 0.313 | 3 | 0.938 |
| languages supported | 4 | 0.250 | 3 | 0.750 |
| special input devices | 0 | 0.000 | 0 | 0.000 |
| screens affected by font adjustments | 4 | 0.250 | 1 | 0.250 |
| | 16 | 1.000 | 10 | **2.500** |

| Robustness | | | | |
|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **N. Value** | **Total** |
| number of lines of source code | 3 | 0.103 | 1 | 0.103 |
| amount of threads generated | 3 | 0.103 | 3 | 0.310 |
| changes to dynamic data | 0 | 0.000 | 0 | 0.000 |
| class inheritances | 3 | 0.103 | 3 | 0.310 |
| multitask command buttons | 3 | 0.103 | 1 | 0.103 |
| program switches lacking default clause | 5 | 0.172 | 3 | 0.517 |
| use case totals | 4 | 0.138 | 1 | 0.138 |
| unrestricted text fields | 3 | 0.103 | 1 | 0.103 |
| user interface complexity | 3 | 0.103 | 2 | 0.207 |
| user-levels | 2 | 0.069 | 2 | 0.138 |
| | 29 | 1.000 | 17 | **1.931** |

| Software Vulnerability | | | | |
|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **N. Value** | **Total** |
| Firewall port required | 5 | 0.294 | 2 | 0.588 |
| network connections required | 4 | 0.235 | 3 | 0.706 |
| open API | 2 | 0.118 | 1 | 0.118 |
| special user accounts required | 5 | 0.294 | 4 | 1.176 |
| web portals required | 1 | 0.059 | 1 | 0.059 |
| | 17 | 1.000 | | **2.647** |

| UI Usability | | | | |
|---|---|---|---|---|
| **Attribute Name** | **Weight** | **N. Weight** | **N. Value** | **Total** |
| message boxes | 3 | 0.150 | 1 | 0.150 |
| multitask command buttons | 3 | 0.150 | 1 | 0.150 |
| screen traversal | 2 | 0.100 | 3 | 0.300 |
| software wizards | 1 | 0.050 | 2 | 0.100 |
| UI responses required | 3 | 0.150 | 3 | 0.450 |
| User interface controls | 3 | 0.150 | 2 | 0.300 |
| unrestricted text fields | 5 | 0.250 | 1 | 0.250 |
| | 20 | 1.000 | | **1.700** |

# Appendix C3

## Calculations for Virtualization Monitoring Software Function Training Process

| Environmental Dependency | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| amount of threads generated | 0.157 | 0.131 | 3 | 0.392 |
| CPU utilization | 0.125 | 0.104 | 3 | 0.313 |
| dynamic link library requirements | 0.105 | 0.088 | 1 | 0.088 |
| Firewall port required. | 0.147 | 0.122 | 2 | 0.244 |
| incoming client service request | 0.105 | 0.088 | 1 | 0.088 |
| multiple system requirements | 0.052 | 0.043 | 2 | 0.086 |
| network connections required | 0.093 | 0.078 | 3 | 0.233 |
| open API | 0.073 | 0.061 | 1 | 0.061 |
| physical processors required | 0.093 | 0.078 | 3 | 0.233 |
| special user accounts required. | 0.135 | 0.113 | 4 | 0.450 |
| web portals required | 0.042 | 0.035 | 1 | 0.035 |
| web service request requirements | 0.073 | 0.061 | 1 | 0.061 |
| | 1.200 | 1.000 | | **2.283** |

| Function Coverage Completeness | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| hardware and software requirements | 0.294 | 0.288 | 3 | 0.864 |
| software wizards | 0.231 | 0.227 | 2 | 0.453 |
| unrestricted text fields | 0.274 | 0.269 | 1 | 0.269 |
| use case totals | 0.221 | 0.217 | 1 | 0.217 |
| | 1.020 | 1.000 | | **1.803** |

| Localization | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| code locale | 0.208 | 0.191 | 3 | 0.573 |
| fonts effect on UI | 0.327 | 0.300 | 3 | 0.900 |
| languages supported | 0.268 | 0.245 | 3 | 0.736 |
| special input devices | 0.000 | 0.000 | 0 | 0.000 |
| screens affected by font adjustments | 0.288 | 0.264 | 1 | 0.264 |
| | 1.090 | 1.000 | | **2.472** |

| Robustness | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| number of lines of source code | 0.108 | 0.097 | 1 | 0.097 |
| amount of threads generated | 0.128 | 0.115 | 3 | 0.344 |
| changes to dynamic data | 0.000 | 0.000 | 0 | 0.000 |
| class inheritances | 0.128 | 0.115 | 3 | 0.344 |
| multitask command buttons | 0.108 | 0.097 | 1 | 0.097 |
| program switches lacking default clause | 0.194 | 0.173 | 3 | 0.519 |
| use case totals | 0.141 | 0.126 | 1 | 0.126 |
| unrestricted text fields | 0.108 | 0.097 | 1 | 0.097 |
| user interface complexity | 0.118 | 0.106 | 2 | 0.211 |
| user-levels | 0.086 | 0.076 | 2 | 0.153 |
| | 1.120 | 1.000 | | **1.986** |

| Software Vulnerability | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| Firewall port required | 0.259 | 0.309 | 2 | 0.618 |
| network connections required | 0.194 | 0.230 | 3 | 0.691 |
| open API | 0.102 | 0.121 | 1 | 0.121 |
| special user accounts required | 0.239 | 0.285 | 4 | 1.140 |
| web portals required | 0.046 | 0.055 | 1 | 0.055 |
| | 0.840 | 1.000 | 2 | **2.625** |

| UI Usability | | | | |
|---|---|---|---|---|
| **Attribute Name** | **L Weight** | **A Weight** | **Value** | **Total** |
| message boxes | 0.153 | 0.141 | 1 | 0.141 |
| multitask command buttons | 0.153 | 0.141 | 1 | 0.141 |
| screen traversal | 0.125 | 0.116 | 3 | 0.347 |
| software wizards | 0.068 | 0.063 | 2 | 0.125 |
| UI responses required | 0.173 | 0.160 | 3 | 0.479 |
| User interface controls | 0.163 | 0.150 | 2 | 0.301 |
| unrestricted text fields | 0.248 | 0.229 | 1 | 0.229 |
| | 1.080 | 1.000 | | **1.764** |

## Bibliography

[1] Amazon.com, "Sports & Outdoors Beta", accessed on: July 29,2004, available: http://www.amazon.com/exec/obidos/subst/misc/sporting-goods-beta-launch-no-link-pop-.

[2] Ash, Lydia, *The Web Testing Companion - The Insider's Guide to Efficient and Effective Tests*. Indianapolis: Wiley Publishing, Inc., 2003.

[3] Beck, Kent, *Extreme Programming Explained:Embrace Change*: Addison-Wesley, 1999.

[4] BetaNews Inc., "The Betanews.com - website", accessed on: August 6,2004, available: www.betanews.com

[5] Bevan, Nigel, "Common Industry Format Usability Tests," presented at Usability Professionals Association, Scottsdale, Arizona, June 29 - July 2, 1999, pp. 1 - 6.

[6] Bray, Tim, and J. Paoli, at el., "Extensible Markup Language", World Wide Web Consortium (W3C), accessed on: April 20,2005, available: http://www.w3.org/TR/xml11/

[7] Carnegie Mellon Software Engineering Institute, "CERT Advisory CA-2002-03 Multiple Vulnerabilities in Many Implementations of Simple Network Management Protocol (SNMP)", CERT Coordination Center, accessed on: 12/2004,2004, available: http://www.cert.org/advisories/CA-2002-03.html

[8] Carzanigayz, Antonio, and A. Fuggetta, and at el., "A Characterization Framework for Software Deployment Technologies," CS Dept. Univ. of Colorado, Boulder 1998.

[9] Cianfrani, Charles A., and J. West, *ISO 9001: 2000 Explained (ISO 9000 Series)*, 2 ed. Portland: ASQ Quality Press, 2001.

[10] Collins, Rosann Webb, and, "Software Localization for Internet Software: Issues and Methods," *IEEE Software*, vol. March/April 2002, pp. 74 - 80, 2002.

[11] Computer Associates Inc., "CA Common Product Services Component - Getting Started", Computer Associates, accessed on: 11/2,2004,

[12] Computer Associates Inc., "CA Open and Selective Beta Programs", accessed on: August 22,2004, available: http://www3.ca.com/betas/

[13] Craig, Rick, and S. P. Jaskiel, <u>*Systematic Software Testing*</u>. Norwood, MA: Artech House Publishers, 2002.

[14] DeMarco, Tom, <u>*Controlling Software Projects: Management, Measurement and Estimation*</u>. New Jersey: Yourdan Press, 1982.

[15] Dyson, Peter, <u>*Dictionary of Networking*</u>, vol. 3. Alameda, CA: Sybex Incorporated, 1999.

[16] Fine, Michael R., <u>*Beta Testing for Better Software*</u>. New York: Wiley Computer Publishing, 2002.

[17] Gabbert, Paula, and, "Globalization and the Computing Curriculum," *ACM SIGCSE Bulletin*, vol. 35, 2003.

[18] Google Inc., "Google Alerts Beta Web Application", accessed on: Sept 15,2004, available: http://www.google.com/alerts

[19] Hetzel, Bill, <u>*The Complete Guide to Software Testing*</u>, vol. 2. New York: John Wiley & Sons, 1998.

[20] Hodge, Susan, <u>*Computers: Systems, Terms and Acronyms*</u>, 14 ed. Casselberry: SemCo Enterprises, Inc., 2003.

[21] IBM Corporation, "SNMP product vulnerabilities in IBM Tivoli Products", IBM Corporation, accessed on: 12/2004,2004, available: http://www-1.ibm.com/support/

[22] IBM Rational, "IBM Software Support: Rational Software Support: Beta programs", accessed on: 8/6,2004, available: http://www-306.ibm.com/software/rational/support/beta.html

[23] IEEE Std 610., and, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Computer Society*, 1990.

[24] Intel, "Taking Advantage of Usability Features of the Intel C++ Compliler 8.1 for Linux", Intel Corporation, accessed on: 1/29,2005, available: http://www.intel.com/cd/ids/developer/asmo-na/eng/microprocessors/pca/195905.htm?page=3

[25] Kropp, N.P., P. J. Koopman, "Automated Robustness Testing of Off-the-Shelf Software Components," presented at The Twenty-Eighth Annual Intertational Symposium on Fault Tolerant Computing, Munich, Germany pp. 230.

[26] Lethbridge, Timothy, and Robert Laganiere, *Object-Oriented Software Engineering*, vol. 1, 1 ed. Maidenhead: McGraw-Hill Education, 2001.

[27] Li, Eldon Y., and, "Software Testing In A System Development Process: A Life Cycle Perspective," *Journal of System Management*, vol. 41, pp. 23 - 31, 1990.

[28] LISA Organization, "Welcome to Localization Industry Standards Association:LISA", accessed on: 12/1,2004, available: http://www.lisa.org/

[29] Microsoft Corporation, "Article: 873165 ", Microsoft Corporation Knowledge Base, available: http://support.microsoft.com

[30] Microsoft Corporation, "The Customer Experience Improvement Program", accessed on: August 15,2004, available: http://www.microsoft.com/products/ceip/

[31] Microsoft Corporation, "Microsoft Technet Network", accessed on: October 6,2004, available: http://www.microsoft.com/technet/default.mspx

[32] Microsoft Corporation, "Microsoft TechNet Virtual Lab", accessed on: August 6, 2004,2004, available: http://www.microsoft.com/technet/traincert/virtuallab/default.mspx

[33] Microsoft Corporation, "System Requirements for Exchange Server 2003", Microsoft Corporation, accessed on: February 10,2005, available: http://www.microsoft.com/exchange/evaluation/sysreqs/2003.asp

[34] Microsoft Press, "Microsoft Press Computer and Internet Dictionary," vol. 2004, 3rd ed: Microsoft Corporation, 1997.

[35] Mossienko, Maxim, and at el., "Towards managing environment dependence during legacy systems renovation and maintenance," presented at Third IEEE International Workshop on Source Code Analysis and Manipulation, Amsterdam, The Netherlands pp. 10.

[36] Myers, Glenford J., *The Art of Software Testing*. New York: John Wiley & Sons, 1979.

[37] National Coordination Office for Information Technology Research and Development, "High Confidence Software and Systems Research Needs," High Confidence Software And Systems Coordinating Group, Ed.: Interagency Working Group on Information Technology Research and Development, 2004, pp. 48.

[38] Nilsson, Nils J., Learning Machines: Foundation of Trainable Pattern-classifying systems. Menlo Park, California: Stanford Research Institute, 1965.

[39] O.j.Dahl, and E. Dijkstra, Structured Programming. New York: Academic Press, 1972.

[40] Oracle, "Oracle Technology Network", accessed on: August 8,2004, available: http://www.oracle.com/webapps

[41] Pressman, Roger, and B. Jones, Software Engineering: A Practitioner's Approach, 5th ed. New York: McGraw-Hill, 2001.

[42] Raynus, Joseph, Software Process Improvement with CMM. Norwood: Artech House, 1999.

[43] Section508.gov, "508 Law - The Rehabilitation Act", accessed on: July 16,2004, available: http://www.section508.gov/index.cfm?FuseAction=Content&ID=3

[44] Shelton, Charles P., P. Koopman, K. Devale, "Robustness Testing of the Microsoft Win32 API," presented at International Conference on Dependable Systems and Networks (DSN 2000), New York, New York

[45] Singh, Amit, "An Introduction to Virtualization", kernelthread.com, accessed on: April,2005, available: http://www.kernelthread.com/publications/virtualization/

[46] Snedecor, George W, and W. Cochran, Statistical Methods. Ames, Iowa: Iowa State University Press, 1989.

[47] Sun Microsystems, Inc., "Java 2 SDK SE Developers Documentation", Sun Microsystems, accessed on: March 17,2005,

[48] Tang, Dong, M. Hecht, "Evaluation of Software Dependability Based on Stability Test Data," presented at Twenty-Fifth International Symposium on Fault-Tolerant Computing, Pasadena, California, June 27 - 30, 1995,

[49] Thibodeau, Patrick, "Users Begin to Demand Software Usability Tests", ComputerWorld, accessed on: 5/3/2003,2003, available: http://www.computerworld.com/printthis/2002/0,4814,76154,00.html

[50] U.S. Department of Health and Human Services, "Usability.gov:Usability Basics", accessed on: March,2005, available: http://www.usability.gov/index.html

[51] Whittaker, James A., and, "What is software testing?  And why is it so hard?" *IEEE Software*, vol. January/February 2000, pp. 77 - 78, 2000.

[52] Whyman, Edward K., and H. L. Somers, "Evaluation Metrics for a Translation Memory System," *Software-Practice and Experience*, vol. 29, pp. 1265 - 1284, 1999.

[53] WorldLingo, "Glossary of Terms", WorldLingo, accessed on: 12/1,2004, available: http://www.worldlingo.com/resources/glossary.html

[54] Yahoo Incorporated, "Yahoo.com", Yahoo! accessed on: February 1,2005, available: http://www.yahoo.com/