

## **The Development of a Thorough Test Plan in the Analysis Phase Leading to more Successful Software Development Projects**

**Jim Nindel-Edwards**

Seattle Pacific University  
e-mail: [nindej@spu.edu](mailto:nindej@spu.edu)

**Gerhard Steinke**

Seattle Pacific University  
e-mail: [gsteinke@spu.edu](mailto:gsteinke@spu.edu)

### **ABSTRACT**

*A critical problem in many software development projects is missing some important requirements until late in the development life cycle. The impact of a missing or misunderstood requirement can add to the project cost and delay product launch due to rework both of the application code, documentation updates, and addition test passes. Building a thorough test plan very early in the product development cycle has the potential for early discovery of missing requirements with attendant reduction in project costs and schedule improvement. In this paper we discuss some of the causative factors of how and why requirements can be missed in software development projects and show how the early formulation of a test plan can directly address these factors. Key to the success of this approach is the understanding that mature tests often go beyond the documented requirements for applications and into exploring the boundaries of the application domain where missing requirements can often be uncovered. In addition the test methodology of looking at features from multiple users' perspectives and less common use cases, shed light on misinterpreted and/or misunderstood requirements.*

Key words: Requirements management, test plan, test plan checklist, functional requirements, missing requirements.

### **INTRODUCTION**

Software development projects invariably work better when requirements are thoroughly documented, designed and developed,. Yet in many cases the users and stakeholders of projects are not entirely clear in expressing their product requirements (Jones, 1997). The difficulty in visualizing the end product makes it challenging, if not downright difficult, to capture and document all the software product requirements. Without clear and complete requirements documentation developers often must build a complete, complex system from the only view of the product that typically has any real meaning to the user, the ubiquitous *feature list*. This list – hopefully found in the project proposal –becomes the de facto description of the new, or newly enhanced, software product.

Yet even complete, approved, requirements documentation often miss items critical to the success of a software project. The requirement may simply be assumed by the customer, may have been consider and not documented, or at the start of a project simply unknown. The missing requirements may be discovered in the build process or test cycle. Sadly some requirements go totally undiscovered until the product is released or implemented only to fail or be seriously defective in its first use due to “bugs”, some of which may in fact be missed requirements.

While it is both possible and likely to miss some product functionality requirements, the bulk of truly missed requirements lie outside the realm of the function/features of a software product (Leffingwell and Widrig, 2003). Factors such as product performance, load limits, target installation environments, and applicable standards are not generally considered primary requirements of a software product yet play a key role in the success (or failure) of a software product. While good and well structured requirements documentation will address many of these environmental requirements, those that are missed in the requirements stage often still are missing at product launch. To the extent they truly are requirements for product success early discovery of these in the project can be a critical success factor in helping to create an “on time, on budget” software development effort that meets both customer and stakeholders' expectations.

Various methodologies, Object Oriented, Spiral (Boehm, 2000), Unified Development, Extreme (Jeffries, 2001), and many other development lifecycles (Kay, 2002) address requirement elaboration in different ways and while

typically not focused on requirements management recognize the importance of the base requirements. Regardless of the chosen methodology recognition of important requirements late in the product lifecycle is an expensive consequence.

In this paper we look at the foundational elements of project requirements and suggest that a thorough test plan will provide the environment for determining and classifying a more complete list of requirements beyond the commonly used functional requirement orientation. This enhanced emphasis on the test plan will be used to help us understand key strengths and key weaknesses of typical requirement methodologies. We suggest completing a test plan template that should allow the software development manager to better understand the full requirements of the software project going beyond functional requirements—and thus lead to more successful projects.

## GENERAL APPLICATION REQUIREMENTS

The starting point for our research starts with the definition of “requirements”. A typical usage of this term *requirement* is “a statement of need, something that some class of user or other stakeholder wants.” (Alexander and Stevens, 2002) Another definition of requirements is offered by requirements engineering authors Dorfman and Thayer (1990) in two parts:

1. A software capability needed by the user to solve a problem to achieve an objective.
2. A software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documentation.

In both of these definitions the primary emphasis is on the functionality of the software product, those features that satisfy a user’s needs. Alexander and Stevens (2002) go on to define a *function* as “... something that a system or subsystem does, presumably because a requirement made it necessary.” They also state that the commonly used term “functional requirement” “in practice ... means the same as function.”

Note that the second part of the Dorfman and Thayer (1990) definition of requirements goes beyond the needs of the software product, moving into the “contract, standard, specification, or other formally imposed documentation.” Typically it is in these documents – the contracts, standards, specifications, etc., that the hidden or missed requirements will be found. Certainly we can miss functional requirements, but it is the area of form, fit, finish, and integration where we are more likely to overlook project requirements and it is in the “testing” of these less well documented requirements that the test plan creation process can help uncover missing project requirements.

Humphrey Watts (1990) refers to *unknown requirements* and later *misunderstood requirements*. A misunderstood requirement often is recorded, but missing some contextual reference that would allow the requirement to be fully *understood*.

Why should we be concerned about these other requirements that do not directly affect application functionality? Simply, meeting the second class of requirements is a prerequisite for acceptance of the project. Failure to meet a specification can preclude payment for a project. Rework may be required to fulfill a standard. Missing a contractually defined delivery date may reduce payment and/or impose late delivery payments from the supplier. Clearly it is our best interest as project managers to fully understand *all* the requirements of a project and in doing so lessen the risk of missing or misinterpreting any requirements. We suggest that these requirements will not be forgotten if a thorough test plan is included in the requirements definition phase.

Requirements come in various forms and a clear, comprehensive understanding of all the project requirements is a prerequisite for the thorough test plan. Looking broadly we can classify requirements in to at least six different types.

### *Functional Requirements - What an application must do*

*Functional* requirements are the most basic of all software and project requirements and many books and articles, including *Managing Software Requirements* (Leffingwell and Widrig, 2003) and *Writing Better Requirements* (Alexander and Stevens, 2002), have been written on this subject. Simply stated, the functional requirements of a software product must answer the question: “What must the software product do?” to be a successful application. This seemingly simple question becomes increasingly complex as more stakeholders and users get involved in requirements process. Functional requirements can be, and often are, in conflict with one another. An example of a potential requirement conflict might be: “easy to use” and “comprehensive in features.”

Having collected all the requirements there can and should be a process of requirements conflict resolution to, where possible, we mold the functional requirements into the best possible package. For example, given the “feature rich” and “easy to use” requirements previously mentioned, a new requirement entitled: “allow customized user levels” might be introduced to resolve the apparent requirements conflicts. In other cases some features may simply be dropped as redundant in a comprehensive requirements review session.

Since virtually all software product development efforts have resource constraints, software product managers typically will segregate functional requirements into the “needs” and “wants.” Balancing out the needs, wants, delivery date/time, and project costs is the very essence of software project management. If we are missing requirements or requirement priorities the process of selecting critically important functional requirements must be flawed as we do not have all of the information required for to make a fully informed selection. But herein lies the importance of creating a test plan which could identify all the requirement priorities for a successful software project.

And assuming that we have a full set of requirements, are they fully understood? In the end there is a *test* for good requirements – are they themselves *testable*? Using the test plan approach can enhance the functional requirements process by assuring that the requirements are more precise, i.e. testable. Good functional requirement testability is an important contribution to the requirements process -- helping to flush out ambiguities in function requirements.

#### *Application Architecture – How does this fit?*

Software architectural and environmental requirements typically address the target computing environment for the software product. Ideally this information is included in the project requirements and happily, in most cases this is addressed. The full ramification of the target environment however may not be considered, particularly for some statistical outliers in the target environment. An example might be a target of a certain class of workstations only to find out in the implementation phase that not all departments have the specified or hoped for hardware. And, worse yet, some other business mandated software package precludes upgrading their hardware.

If this architecture requirement is included in the test plan, then the “missed” requirement here might be easy to rectify early in the development cycle, instead of being discovered at later phases of implementation which would cause serious, and costly, deployment delays and/or rework of the application to accommodate the “required” hardware configuration.

#### *Environmental Requirements – Where will it run?*

Environmental factors can follow a similar path of being included in some form in the initial project requirements yet incomplete in some critical regard at the time for beta testing and/or implementation. Referring to Myers’s Project Objective list (Myers, 2003), application efficiency, compatibility, security, and several of the reliability attributes can be classified as environmental features, or requirements of software applications.

In the end analysis, this branch of the requirements project tree is the most productive area to specify in a test plan, particularly as our computing environments grow more complex (Schneberger, 1995). An intentional inclusion or exclusion of a specific environment for a software product in our test plan would represent a known and accommodated requirement. Software test plans which clearly state “included” and “excluded” environmental factors help to bring focus to this area.

#### *Hardware and Software Standards – Does it comply with standards?*

Standards represent an abstraction of hardware and software operating environments. Calling out applicable standards for any software product should be a part of the project requirements (Patton, 2001) and most requirement engineering templates provide for citation of applicable standards. Virtually no software product exists without some standards based context, if nothing else, the target operating system(s). When no standards are either specified or acknowledged the software development project is in jeopardy as any change to the deployment target outside of the development environment could be considered as either a failed implementation or uncontrolled change to the project requirements.

Citing of applicable hardware/software standards will reduce project risk factors. Citing of standards in project requirements therefore is necessary to a robust software product development project.

A robust test plan provides two valuable functions in the compliance area. First is the validation that the software product in fact meets its target standards. Second, typical robust test plans and associated test cases will push the edges of the compliance requirements to determine how the application responds outside of the standard settings. Typically referred to as edge and boundary tests the test plan and associated test results can help product managers understand how the software product performs when applied outside the required standard.

#### *Functional Attributes – Does it satisfy secondary user groups?*

There exists a whole host of other functional requirements for software products to be successful, many of which are not specifically identified by the target user group for a specific product. The primary focus of most requirements sets is the functionality required causing a specific user group(s) to be interested in buying, switching to, or to otherwise accepting a specific application package. But there are secondary user groups who will be impacted—and they have their own requirements. Their set of functional requirements is too often overlooked in the requirements process.

In this area several of Myers's other "Product Objectives" (Myers, 2003) fit, including documentation, efficiency, installation, and several reliability attributes. When working from high-level project requirements templates often these areas will in fact be addressed – at least they might have a subject heading in the requirements document. But too often they are simply addressed at the high level and thus are lost as the software development project unfolds.

Vague and ambiguous efficiency and reliability requirements can easily be lost in an application design process. Product documentation too often is grafted onto software development projects or – worse yet – eliminated entirely, sacrificed to reduce project costs, shorten the initial product development cycle, or to allow including more functionality into the product. If efficiency, reliability, and documentation requirements are not clearly articulated and made part of the testing plan for the software product very likely they too, may be found missing when the product is delivered to the customer.

Likewise software products that are installed in a shared environment (like an end user workstation) that either cause another application to fail, and/or are themselves "damaged" by the installation of another software package are of low quality and reliability. Unless we are building truly turn key hardware/software applications the install ability of an application should be part of the test plan.

The test plan's role in this area has the highest likelihood of revealing missing requirements. Test plans or test scenarios depicting usage by secondary user groups can point out glaring holes in the product requirements. Robust test plans will look at the software product from all perspectives, primary users, secondary users, hard core users, and casual users. All of these scenarios and the corresponding test cases can yield both insights into missed or potentially missed requirements as well as providing validation for existing features included in the requirements.

#### *Product and System Security Requirements – Is it Secure?*

Security is in some regards similar to the previous group in that security is both functional, and necessary for most software applications. The *functional* aspect is that most software applications must consider and provide for application and data security. The project requirement here however is a little tenuous – must the application provide for the security? Or the environment where the applications operates, i.e. the operating system?

Certainly we could consider application and data security to be functional – a feature the application must provide. It could be considered environmental – some combination of application functionality and operating system features could make an application "secure." Or even a functional requirement of a secondary user group – security being the responsibility of system administration rather than the primary user of an application. Sadly if left to any of these categories security features may be found to be lacking until late, perhaps very late, in the development cycle. Security requirements are particularly important to be well known and well understood early in the development cycle.

Security is another area where the test plan can help drive out both the comprehensive security needs of a software product. The peculiar attribute of computing security is that a functional requirement is quite easily stated – the application and its data must be secure. We can even go onto defining some of the attributes – data can be accessible by those who need it, but not to those that do not. Detailed test plans that include the methods of attack for an application can be a significant aid in the application design process to highlight the various possibilities for

application and data attack that the designs might not consider until too late in the development project, or too late after a successful data attack on the released application.

### **TEST PLAN PERSPECTIVE – VERIFYING FUNCTIONALITY, FINDING DEFECTS**

We can apply a thorough test plan, identifying all requirements in this test plan during the requirements engineering phase. This approach reduces the likelihood of missing requirements to being discovered late in the development effort when they are more expensive to repair.

#### *Using a test plan checklist*

Using checklists is a common practice used in most formal development methodologies and is a well-accepted technique to insure all important project requirements have been considered in a software product's development plan. The process of gathering functional requirements generates the features list that becomes the functional checklist for both software quality activities and user acceptance. A comparable test plan checklist can be used to reasonably assure that all project requirements have been considered in the project plan.

Functional tests mapped directly or indirectly to product objectives and functional requirements often will make up the bulk of the software test plan. Certainly a software product must meet functional requirements to be acceptable. The bulk of the test plan will most likely be directed at the classical areas of unit test, [unit] integration testing, functional/black box testing, etc. By in large, all of these tests are of course necessary and will serve to validate both the application design, and verify the application meets the functional requirements.

Once we move beyond the function requirements other software product attributes are called for in the test plan. Generally termed "system test" in test plan documents these areas have a greater probability of catching missed product requirements. Unfortunately however in a typical product life cycle these tests are not normally scheduled until late in the project, since they normally require a functioning application to perform the test. This relegation to late in the life cycle carries the risk of increased cost to repair, and higher likelihood of impacting the product delivery schedule. But looking at the software product from the software test perspective yields a different vision that goes beyond the function product attributes, as seen below in Figure 1. That is why clearly articulating the test plan early on is so valuable.

Naturally a test plan initiated early in the product development cycle must keep pace with the application being developed. As the application progresses through the design, develop, test, change, re-release, re-test cycle new areas for system level tests will be exposed. A complete and well developed test plan will need maintenance just as the application it verifies is being developed and maintained.

<p><b>Functional Requirements Testing</b></p> <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> All documented user requirements, primary and other required use case scenarios</li> <li><input checked="" type="checkbox"/> Human Factors – Identification of processes and procedures that will be difficult or inconvenient for the users/operators of the application <ul style="list-style-type: none"> <li>• User’s manual and/or on-line documentation</li> <li>• Help files</li> <li>• Samples and examples</li> <li>• Icons and artwork</li> </ul> </li> </ul> <p><b>Application Architecture Testing</b></p> <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> Configuration: Determine the planned legal hardware and/or software configuration on which the system will, and will not operate correctly</li> <li><input checked="" type="checkbox"/> Compatibility – Test and expose areas where system interfaces that are possible and likely are not fully functional</li> <li><input checked="" type="checkbox"/> Installability – Test for proper installation of the application and determine situations where correct installation can yield incorrect results from the application and/or adverse affects on the installation target</li> </ul> <p><b>Environmental Requirements Testing</b></p> <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> Load/Stress Testing – Objective: To identify the peak load conditions at which the system fails.</li> <li><input checked="" type="checkbox"/> Volume – Objective: Test to determine the level of continuous heavy load at which the system will fail</li> <li><input checked="" type="checkbox"/> Performance – Determine and benchmark application performance under nominal and peak conditions</li> <li><input checked="" type="checkbox"/> Reliability/Availability – Records the uptime performance of the application under nominal and peak loads <ul style="list-style-type: none"> <li>• Setup and installation</li> </ul> </li> </ul> <p><b>Hardware and Software Standards Testing</b></p> <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> Application deployment on all specified hardware and using requirement specified operation systems, including where applicable different versions of operating systems</li> <li><input checked="" type="checkbox"/> Verification against various system interfaces, including current industry standards for information exchange (HTML, XML, SOAP, UDDI, etc.) <ul style="list-style-type: none"> <li>• Error messages</li> </ul> </li> </ul> <p><b>Functional Attributes, Secondary User Groups Testing</b></p> <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> Serviceability – Identify the application’s diagnostic results helping application administrators to determine corrective procedures, both proactive and reactive</li> <li><input checked="" type="checkbox"/> Testing common use case scenarios of secondary user groups <ul style="list-style-type: none"> <li>• Product support information (contact #'s, e-mail addresses, etc.)</li> <li>• Ads and marketing material</li> </ul> </li> </ul> <p><b>Product and System Security Requirements Testing</b></p> <ul style="list-style-type: none"> <li><input checked="" type="checkbox"/> Security – Testing various methods to break through the security provisions of the application</li> <li><input checked="" type="checkbox"/> Recovery – Examines application behavior after encountering errors or other abnormal conditions.</li> </ul>
--

Figure 1: Sample Project Test Plan.

*Test at every step in the product life cycle*

“Testing” of a software application need not wait until the application is actually functioning. Part of the test plan created during the analysis phase, may have items to be tested during each of the following phases. Requirements, specifications, and application design are all testable even from documentation. Projects that use Rapid Application Development (RAD) or eXtreme Programming (XP) techniques actually will have testable software earlier in the

project lifecycle.

Even when RAD and XP techniques are not used every deliverable from a software development effort is testable in some sense. If by no other method than by document review the requirements of individual system components may be tested through scenario analysis to see the effects of these components on larger system. A simple example might be a design feature that allows a web service a limited time, several seconds delay even if the intranet or extranet is slow to respond to a web service request. If this same service were to be used repeatedly in a single transaction the overall response time for the transaction could exceed the design goals for the transaction. In this event we can “test” the transaction without any code even being built. The comparison of the design specifications to a perform requirement or goal is sufficient to see room for a possible transaction failure.

## CONCLUSION

Missing one or more software product requirements needed for a successful launch or version of a software product or system seems as inevitable as the knowledge that there is a “bug” in an application. Just as there can be minor defects in applications that do not prevent product release, there are classes of software requirements that, if missed, can either be gracefully deferred from a product release or renegotiated with the project sponsor. While we can and often do defer both requirements and bug fixes to later releases we need to occasionally ask ourselves, Why? Why did we miss the requirement? Or why was it that the requirement was misunderstood until too late in the project to be corrected prior to project launch? We suggest that the creation of a detailed test plan in the requirements phase will clarify all requirements and thereby lead to more successful software development projects.

A testing plan can help in the early discovery of missing and misunderstood requirements. Development of the product test plan early in the product life cycle, and constantly testing the product against this test plan can significantly improve the overall product quality. Some may ask “How can this be? Testing costs in test resources and adds nothing to the overall functionality of the application!” True perhaps, but the balance here is the cost of rework late in the development life cycle – which likewise adds no value to the software product – it is simply rework. Doing again what might have been done correctly the first time.

## REFERENCES

- Alexander, Ian F. and Stevens, Richard (2002). *Writing Better Requirements*, London, Addison-Wesley.
- Boem, Barry, edited by Hansen, Wilfred J (July 2002). *Spiral Development: Experience, Principles, and Refinements*, Pittsburgh, Carnegie Mellon, Software Engineering Institute. Website: [www.sei.cmu.edu/cbs/spiral2000/february2000/SR08.pdf](http://www.sei.cmu.edu/cbs/spiral2000/february2000/SR08.pdf)
- Dorfman, Merlin, and Thayer, Richard H. (1990). *Standards, Guidelines, and Example of System and Software Requirements Engineering*, Los Alamitos, CA: IEEE Computer Society Press.
- Jeffries, Ron (2001). *What is Extreme Programming?*, XProgramming.com. Website: <http://www.xprogramming.com/xpmag/whatisxp.htm>
- Jones, Casper (1997). *Software Quality, Analysis and Guidelines for Success*, London, Thomson Computer Press, page 67.
- Kay, Russel (May, 2002). *QuickStudy: System Development Life Cycle*, COMPUTERWORLD. Website: [www.computerworld.com/developmenttopics/development/story/0,10801,71151,00.html](http://www.computerworld.com/developmenttopics/development/story/0,10801,71151,00.html)
- Leffingwell, Dean and Widrig, Don (2003). *Managing Software Requirements, a Use Case Approach*, 2<sup>nd</sup> Ed, Boston, Addison-Wesley page 129.
- Myers, Glenford J. (1976) *Software Reliability*, New York, John Wiley & Sons.
- Patton, Ron (2001) *Software Testing*, Indianapolis, SAMS Publishing, page 59.

Schneberger, Scott L (1995). *Distributed Computer Environment Software Maintenance: System Complexity Versus Component Simplicity*, Proceedings from the Association for Information Systems Inaugural Americas Conference. See: <http://socrates.baylor.edu/ramsower/acis/sessions.htm> Watts, Humphrey (1990). *Managing the Software Process*, Addison-Wesley.