Testing in the Early Phases of the Software Development Life Cycle

Sarah B. Lee  
*University of Memphis*  
sblee@memphis.edu

Hima Puppala  
*University of Memphis*  
hbpuppal@memphis.edu

Linda B. Sherrell  
*University of Memphis*  
sherrell@cs.memphis.edu

Sajjan Shiva  
*University of Memphis*  
sshiva@memphis.edu

Abstract

The aim of this research is to design a testing process for FedEx Corporation that more thoroughly integrates testing activities throughout the software development life cycle. With the new process, testing personnel will be actively involved in the planning, requirements, and design of the project. This should have the side-benefit of increasing morale among testing personnel as they are committed to the project from its inception to completion. In other words, testing personnel are not treated as a necessary evil “to find bugs” but as critical contributors to the software team.

1. Introduction

Throughout the information technology (IT) industry, there is increasing interest in improving the effectiveness of software testing in the software development lifecycle (SDLC). In addition to the automation of testing techniques, there is much interest in the relationship of quality requirements in the requirements development and management phases of the SDLC to effective testing.

In the SDLC, requirements inspection is recommended early to reveal defects in the requirements specification. Execution-based testing traditionally is done later in the SDLC with an objective of finding defects in the code [2]. According to National Aeronautics and Space Agency (NASA) findings, “problems that are not found until testing are at least 14 times more costly to fix than if the problem was found in the requirements phase”[19]. Furthermore, a James Martin study revealed that the root cause of 56% of all defects is errors introduced in the requirements phase. Approximately half of these defects are a result of poorly written, ambiguous, and incorrect requirements [12]. A Standish Group study in 2000 quantified the cost to U.S. business for software projects that are cancelled or exceed time and budget estimates. The reasons for these failures included requirement specifications that are incomplete or change too often [24].

With this type of knowledge, many researchers are proposing that testing techniques be applied in the requirements phase of the lifecycle, with a goal of uncovering requirements defects well before programming starts. This goal is based on the assumption that if software development is based on incomplete or incorrect specifications, then the resulting product will be unsatisfactory and unable to fulfill the user requirements no matter how well-written the code may be [2]. Additionally, involvement of testing resources in the technical design within iterative development life cycles is appearing in research proposals [18].

While the benefits for testing early in the SDLC are obvious in theory, it is frequently easier to quantify the cost of detecting and correcting defects after implementation. Quantifying the benefits of early testing is not as easy. It is hard to estimate the potential propagation of a defect that is prevented from ever occurring. Additionally, the scenario that would cause a defect to be revealed is difficult to assess. If a defect would rarely occur in its intended environment, then it may have little impact on product costs. However, a defect that appears only when a rare set of conditions occur could be serious enough to have a large impact if initiated [6].

Section 2 of this paper reviews software life cycles. Sections 3 and 4 elaborate on current research regarding testing and its relationship to the requirements and design phases. Section 5 discusses the Global Development Process (GDP), which serves
as a development process guideline at FedEx, with particular attention to GDP activities that occur before the actual development phase.

2. Software Life Cycle Models

Both the management information systems writings and the software engineering literature contain many examples of software life cycle models. Some models prescribe recommended procedures for software development, whereas others describe actual customs [16]. This research will define a software life cycle model and associated process that will bridge the gap between the existing theory and the current practice at FedEx Corporation.

The software life cycle model that is currently in use at FedEx is called Global Development Process (GDP). Because this model has documentation and deliverables associated with each phase and requirements are frozen after the specification phase, the current life cycle model is best described as a variant of the Waterfall Model. It is well known that the Waterfall model has several limitations [17, 20, 22]; the most notable one being that it is inflexible to changing requirements.

A model that incorporates testing earlier in the software development lifecycle is the V Model [5]. In particular this model emphasizes the correlation between development phases (requirements analysis, system design, and program code) and testing phases (acceptance testing, system testing, and unit testing). However, the V Model is still seen as a variant of the Waterfall Model.

Most modern software development life cycle models embody incremental development in some fashion [22]. These models facilitate the delivery of a product that meets the expectations of the customer as valuable feedback is gained after the delivery of each increment. One such model is the Unified Process [8], a two-dimensional life cycle model that incorporates the advantages of both the waterfall and the incremental models. In particular, UP has the phases Inception, Elaboration, Construction, and Transition, which satisfy the business needs of management with deliverables at the end of each phase, along one axis. The workflows or disciplines, which encompass the traditional phases such as Business Modeling, Requirements, Design, Implementation, and Testing, are positioned along the other axis. While the activities of a particular discipline may dominate a phase, developers are not constrained by documentation that is frozen at the end of a phase, and are even encouraged to revisit as many disciplines as necessary during each phase. Within each phase, incremental and iterative development occurs.

The previous approaches are described as disciplined or plan-driven methods. However, some companies are now adapting agile software development methodologies, as these techniques are better suited to rapidly changing requirements and provide more interaction with the client. The agile models support incremental development with builds usually ranging from 2 weeks to a month. In addition to describing the life cycle model, agile methods such as eXtreme Programming (XP) [1], Scrum [21], and Crystal Clear [4] provide process guidelines with special attention to team interactions. Furthermore, XP [23] guidelines describe test-driven development where test cases are written and executed before the actual program code is written.

Although the above life cycle models are worth our investigation, they do not directly address process improvement or team building activities. The Team Software Process (TSP) [7] provides strategies for building self-directed teams, which establish their own goals and plans. A four-day set of activities known as the Launch is especially helpful as stakeholders meet and establish project objectives, roles and responsibilities, and outlines for quality and risk management plans. In addition, two-day Relaunches, which are planning meetings for the next phase, are conducted after each phase. The use of TSP can improve quality and result in cost savings. For example, a pilot study of TSP at Microsoft resulted in a significant reduction of unit test defects (from 25 to 7 defects/KLOC) after TSP training. The pilot team also spent 11.5% of their effort in traditional testing as compared to the usual 40-60% of development time [9].

Processes that deal specifically with testing are not as prevalent in the literature as those encompassing the entire software development life cycle. Guidance on testing methods can be gleaned from [13], [14] and the Capability Maturity Model Integrated (CMMI) [3]. A summary of testing practices primarily from the earlier works of Perry and Myers appears in [10], providing a good outline of the testing practices and deliverables for each phase of the software development life cycle model.

3. Merging Requirements Inspection and Testing

A variety of approaches are found for merging requirements inspection with software test planning, design, and execution. Perspective-based reading
(PBR) focuses on the perspectives that people have based on their role in the SDLC. In reviewing the requirements, the tester will review from the testing perspective, with a focus on test case generation, while another reviewer brings the software designer perspective. Benefits of this technique include providing an opportunity to detect any potential defects that could evolve into real defects, and test cases generated at this early stage can be used for actual testing later [2].

In [2], the Classification Tree Method (CTM) is recommended for black-box test case generation in support of PBR. This method involves decomposing each specification into functional units that can be tested independently. A test case table is generated from the intermediate classification tree that is developed. Test cases are generated from the test case table. The PBR-CTM method combines test case generation with requirements inspection, helping the reviewer detect requirements defects effectively and generating test cases for later use [2].

Mogyorodi proposes a simple Requirements-Based Testing (RBT) approach to the SDLC where:

- as soon as requirements are complete, they are tested,
- as soon as design is complete, the requirements are walked through the design to ensure they are met,
- as soon as code is constructed, it is reviewed and tested as usual.

Because testing begins in the requirements phase, many defects are avoided later.

In this approach, an ambiguity review is conducted to identify and eliminate ambiguous words, phrases, and constructs in the requirements, producing a high-quality set of requirements. A cause-effect graph is built from the requirements, which among other benefits, enables notation of precedence rules within the requirements. Test cases are generated from the logic present in the cause-effect graph, and these test cases are then reviewed by the requirements authors. The RBT approach is designed to ensure maximum coverage by testing and to prevent testing from being a bottleneck within the SDLC [12].

Ramachandran proposes an approach for integration of the testing phase with the other phases of the SDLC in order to detect errors earlier. With test engineers involved from the beginning, requirements are verified and validated to eliminate ambiguity. Requirements-based and design-based test cases are performed before coding begins [18]. Lutz provides a Safety Checklist for use with safety-critical, embedded systems that targets two main categories of software errors. These include inadequate interface requirements and discrepancies between documented requirements and the requirements actually needed for correct functioning [11].

An Error Abstraction Process (EAP) is proposed in [26] as an approach to ensuring a sound verification process for the requirements phase. A Requirement Error Classification Taxonomy (RET) is used to analyze and abstract requirements errors that may lead to defects and failures in the target product. While results are promising in the academic setting in which this was prototyped, there are no published results from an industrial setting [26].

4. Design-based Testing

In the design phase of the SDLC, the technical architecture is identified and the detailed design is developed. Often, designers perform their tasks without detailed documentation of rationale for design decisions. In [18], testers are recommended as active participants on the design team to assess the testability of the technical architecture and the detailed design rationale.

Following XP guidelines, design occurs throughout the development process. In [25], the author recommends an approach coined a “less eXtreme approach” which combines early prototyping with emergent design.

Prototypes provide early feedback to the customer which enables them to identify if the original requirements are correct. Early prototyping also provides an opportunity for the designers to determine if a chosen technical architecture and detailed design approach will be sufficient. While early prototyping serves to identify potential functional and non-functional design decisions, it appears at first glance to be at odds with the emergent design approach of XP which recommends against spending time in the beginning of a project getting the architecture correct.

In [25], a solution of early prototyping and up-front design in parallel is recommended. This prototyping along with repeatable design techniques is proposed to get the technical architecture right early, and to continue to evolve the detailed design as the project progresses. Involvement of testing resources throughout this “less eXtreme approach” to design will maximize the benefits outlined in [18].

In [15], integration of a model-based design process using levels of design abstraction with a test process is introduced. For testing, the different design abstraction levels are used in different ways. The more abstract design models reflect the user requirements and are
used in test case specifications, while the more concrete models include more detailed aspects of the technical realization of those requirements that are used to derive test cases. With different levels of abstraction focused on different design decisions, test cases are not limited to requirements coverage but also include design rationale which may potentially be error-prone.

By tightly integrating design and testing efforts, a better design process is achieved along with an improved testing process. An additional benefit is documented traceability between the generated test cases and user requirements [15].

5. The Global Development Process Model

The GDP model reviewed in this study consists of five phases: Concept, Definition, Planning, Development, and Launch. With the scope of this research being bound by the requirements and design phases of the SDLC, the Concept, Definition, and Planning Phases of GDP will be explored in more detail. Process improvement in later phases will be left for further research.

In the Concept phase, the development of the Preliminary Business Justification occurs. The Preliminary Business Justification is a high-level overview of the proposed product/service/project including a plan to deliver. The Definition phase identifies the business rules, procedures, actions, and information flow required to implement and support the requested features. The resulting deliverable is a comprehensive Business Requirements Specification (BRS). In these two phases, where requirements specifications are born, there is typically no involvement of testing resources and no explicit effort to assess the quality of the requirements.

In the Planning phase, the high level design occurs along with at least one interim work product review. Test Planning typically begins around the middle of the Planning phase. The objective of the Test Planning step is to create a test plan and related test scenarios for a project including test schedules, identification of additional resources (people, training, and costs), test types to be executed in test environments, and guidance for test design. This process spans the Planning and Development phases. A best practice noted at this step states that requirements should be reviewed to ensure that the different types of tests have been identified to satisfy both business and software requirements. Observations reveal that this best practice is not consistently being applied throughout the software development landscape at the corporation under study. Test Design, where actual test cases are written, does not appear until the second quarter of the Development Phase timeline.

6. Recommendations

To more thoroughly integrate testing activities throughout the software development life cycle at FedEx, a method similar to PBR-CTM should be adopted that will involve testing resources in the Definition phase of GDP. By combining test case generation with requirements inspection, requirements defects will be detected earlier and test cases will be identified for later use.

With the static load schedule that many projects must adhere to, the opportunity still exists for FedEx to embrace an iterative design and development approach that involves testing resources more actively in the design of software products. Internal development deliverables can be cycled towards the final product using early prototyping and repeatable design techniques, while remaining aligned with the enterprise load schedule.

To accomplish the cultural shift from testing as a bottleneck phase at the end of coding, training on the current and improved role of testing in the SDLC should be conducted for all staff involved in the life cycle. It is noted that this is currently being implemented at FedEx through the System Testing Excellence Program.

7. Further Research

Additional effort is required to build a tactical plan for implementing the tasks that will support the recommendations outlined in this document. Further research is needed to explore the additional GDP phases and associated deliverables in terms of the role of testing resources.

8. References


