Journal of the Association for Information Systems (2020) **21**(4), 1045-1071
doi: 10.17705/1jais.00628

RESEARCH ARTICLE

ISSN 1536-9323

Performance Outcomes of Test-Driven Development: An Experimental Investigation

Vikram S. Bhadauria[1], RadhaKanta Mahapatra[2], Sridhar P. Nerur[3]
[1]Texas A&M University Texarkana, USA, vbhadauria@tamut.edu
[2]University of Texas Arlington, USA, mahapatra@uta.edu
[3]University of Texas Arlington, USA, snerur@uta.edu

Abstract

Despite the growing popularity of test-driven development (TDD), there is no empirical confirmation of the benefits that this contemporary practice confers on its users. Prior research findings on its efficacy have largely been inconclusive. We conducted a laboratory experiment to assess the impact of TDD on software quality and task satisfaction. Additionally, we investigated the productivity aspect of TDD as compared to the traditional test-last method of software development. Results indicate that software quality and task satisfaction are significantly improved when TDD is used. Despite the additional requirements of testing, TDD is not more resource intensive than the test-last method. We also examined TDD's impact on learning post hoc and discuss the implications of our findings and directions for future research.

Keywords: Test-Driven Development, Software Quality, Developer Satisfaction, Learning, Experimental Design

Sandeep Purao was the accepting senior editor. This research article was submitted on March 25, 2017, and underwent two revisions.

1 Introduction

Ever since the agile manifesto was articulated (Beck et al., 2001), there has been a proliferation of development methods (e.g., Scrum and Extreme Programming) and their attendant practices aimed at enhancing the quality of software while satisfying the constraints of time and cost. Arguably, the most celebrated of these "best" practices is test-driven or test-first development, which advocates continuous cycles of test-code-refactor rather than the traditional,[1] linear approach of testing after performing analysis, design, and implementation. Not only does test-driven development (TDD) alter the workflow of development activities that were dominant for several decades, but it also forces developers to continually adapt their design strategies and the code that follows. An integral part of TDD is rapid feedback on the system being developed, which provides an opportunity to frequently inquire into what works and what doesn't, and to evolve appropriate designs based on this reflection.

TDD's emphasis on evolving test cases prior to coding is a significant departure from erstwhile approaches to software development. It must be noted that upfront testing is not simply a reordering of the phases of development but rather a design strategy (see Janzen & Saiedian, 2006) that reduces the time between thought and action, thereby fostering a climate for reflective

[1] The traditional test-last approach to software development has been variously referred to as *test-last, test after coding,* and *traditional/classical approach* in the extant literature on software development. In our manuscript, we use the terms traditional and test-last interchangeably.

practice (Schon, 1983). The test-code-refactor-test cycle provides immediate feedback on actions, is more conducive to opportunistic designs, and facilitates continual framing and reframing of the problem and its attendant solution. As the primary cornerstones of the TDD practice, immediate feedback and the capability to redesign rapidly not only engage developers but also enhance their satisfaction (see Tripp & Riemenschneider, 2014).

Empirical studies on the efficacy of test-driven development (TDD) abound (Rafique & Misic, 2013; Wilkerson, Nunamaker, & Mercer, 2012; Madeyski, 2005; Erdogmus, Morisio, & Torchiano, 2005). However, the findings are inconsistent, perhaps because of the varied methods used to study the effects of TDD. For the most part, these studies have focused on the outcomes of TDD, such as external solution quality measured in terms of defect reduction, rather than on the internal processes that this approach facilitates. In order to fill this void, our research uses an experimental study to investigate whether TDD does indeed outperform the traditional approach of software development or not. We measure the performance outcomes of the technique in terms of the quality of the code produced and task satisfaction achieved by the software developer. Specifically, our paper addresses the following twin research questions:

1. Does TDD outperform the test-last approach to software development in terms of the quality of the software produced?
2. Do software developers engaged in TDD experience higher levels of satisfaction than those who use the test-last approach?

In addition to addressing the above research questions, we also explore the effect of TDD on learning outcomes and assess its impact on productivity. Our study makes several contributions to the extant literature on TDD. First, it employs rigorous means (i.e., randomized experimental design with adequate sample) to clarify the relationship between TDD and the quality of software produced. This is particularly useful because prior studies have been largely inconclusive in this regard. Second, while past empirical works have looked at the effect of TDD on quality and productivity, scant attention has been paid to the satisfaction that developers might derive from the use of a test-code-refactor-test cycle of software development. Our study fills this gap by investigating the impact of TDD on the satisfaction of the developer with a programming task. Studying developer satisfaction is critical as it influences job satisfaction, lowers job-related stress, and promotes retention in a profession that is plagued by employee burnout and high turnover rates. Third, researchers have bemoaned the fact that there is a dearth of studies on the learning effects of software development approaches (Avgar, Tambe, & Hitt, 2018; Singh, Tan, & Youn, 2011; Wastell, 1999). With this void in mind, additional analysis was performed to understand the relative impact of TDD (vis-à-vis the test-last approach) on learning outcomes. Thus, although not measured longitudinally, we provide some insight into what are generally regarded as aspects of learning (Gemino, 1999). Finally, this study also contributes to our understanding of the effect of TDD on productivity, something that is not entirely clear from prior studies on TDD.

The remainder of the paper is organized as follows. The next section reviews the literature that provides the conceptual foundation of our study. This is followed by a discussion of our model and the hypotheses associated with it. In the subsequent section, the methodology used to test our hypotheses is presented. Next, we present our results. Following this, we include two additional analyses as a post hoc investigation—one assessing developer productivity and the other exploring learning outcomes of using the TDD approach. We then present the results and the implications of the findings for research and practice. Finally, the paper concludes with a discussion of the limitations of our research and directions for future research.

2 Background Literature

This section provides a brief description of TDD, which contrasts it with the traditional test-last approach to software development, and offers a review of the empirical research on TDD and its impact on software quality and productivity. We also discuss developer satisfaction and Kolb's experiential learning model, which provides the foundation for additional analysis.

2.1 Test-Last and Test-Driven Development

The waterfall model and its variants have guided software development for many decades. In this model, software development proceeds in a linear sequence, starting with planning, analysis, design and coding, followed by testing (Pressman, 2005). There is an implicit assumption that requirements are unvarying and that the development process, including the problems that may arise, can be anticipated ahead of time. With this approach, a separate quality assurance or testing group is given the responsibility to test and ensure that the quality of the software produced meets expectations. This approach to software development is called the traditional or test-last approach.

Figure 1a provides an overview of the application development process in the traditional/test-last approach. The planning stage, which is carried out at the organizational level, is not shown here. After receiving the written description for an application, developers in the test-last approach first perform analysis and design to determine the classes, their methods, and the interactions among them to fulfill the stated requirements.

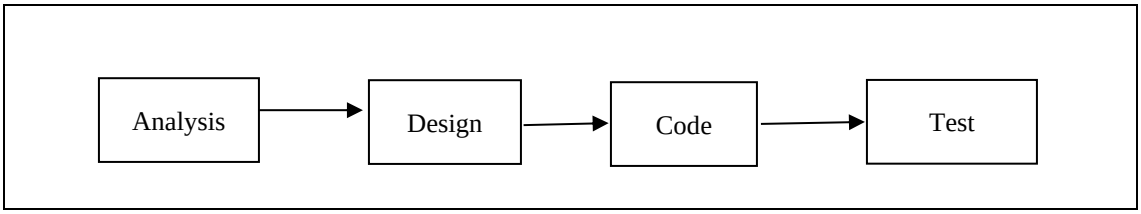

Figure 1a. Test-Last Method of Software Development

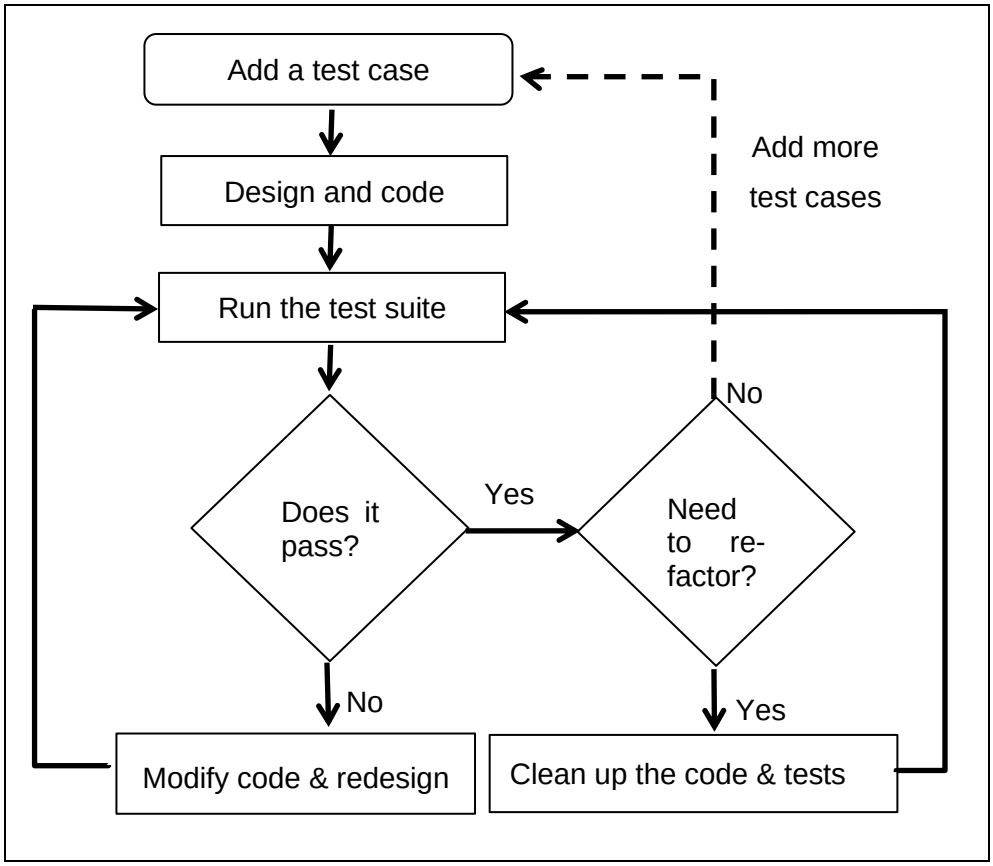

Figure 1b. TDD Process

After conceptualizing and finalizing their design, they implement their solution, followed by testing to ensure that the code works as anticipated. Should errors be reported during testing, the developers typically return to their code to fix the problems rather than questioning the efficacy of their designs. This practice of not questioning design assumptions and not using insights from testing to opportunistically improvise makes the test-last approach different from the test-driven development method described next (Bhat & Nagappan, 2006; Aniche & Gerosa, 2015).

The agile software development (ASD) methodology (Beck et al., 2001; Cockburn & Highsmith, 2001; Nerur, Mahapatra, & Mangalaraj,, 2005) evolved with the intention of overcoming the limitations of the traditional software development method. Central to ASD is an iterative, incremental approach that is largely consistent with the notion of the evolutionary delivery model enunciated by Gilb (Gilb 1989; Larman & Basili, 2003). Many practices such as pair programming, test-driven development (TDD), and continuous code integration have evolved within this framework. Among these, TDD represents a significant departure from erstwhile software development practices because it positions testing as the precursor to coding rather than the other way around. In contrast to the traditional method described earlier, TDD uses cycles of test-design-code-refactor to develop software. Figure 1b shows a typical cycle of the TDD process. Specifically, developers produce progressively useful software by continually iterating through the following steps (Beck, 2002):

1. Adding a test case that not only reflects a requirement from the perspective of a user, but also incorporates the acceptance criteria for that requirement.
2. Writing code to ensure that the test "passes."
3. Refactoring the code to eliminate redundancies and to improve the quality of software being developed while ascertaining that these endeavors don't "break" the code and cause it to fail.

It can be readily appreciated that Step 2 itself proceeds in an iterative manner, as developers repeatedly reformulate their design strategies based on immediate feedback they receive from the code when it fails. It must also be noted that prior to Step 2, the required functionality to make the test pass does not exist. Furthermore, Steps 2 and 3 occur repeatedly until the refactored code (i.e., code that has been modified and/or refined) satisfies the test requirements.

The promotion of testing ahead of coding purportedly has serious implications for design (Janzen & Saiedian, 2006). The primary objective of TDD is "clean code that works" (Ron Jeffries, as cited by Beck, 2002). Automated tests drive software development, ensuring that additions to the growing codebase are made only when tests "fail." The tests reflect a design strategy that is immediately implemented and evaluated, and the feedback that the developer gets from repeatedly carrying out this process is invaluable for developing "clean code." Beck (2002, pp. ix) makes the following observation about TDD: "You must design organically, with running code providing feedback between decisions." These cycles of design-code-refactor lead to reflective action, affording the benefits of opportunistic and improved designs that can result in fewer defects and better software quality. Thus, TDD fosters a climate that is conducive to enhancing developer satisfaction.

As discussed earlier, coding and testing have traditionally been distinct and sequential phases in the software development process. The primary purpose of testing in the test-last approach is to detect errors in the code, whereas TDD strives to anticipate and prevent defects through confirmatory testing of requirements reflected in the test cases (Shalloway, Beaver, & Trott, 2009; Ambler & Lines, 2012). In the latter, developers have to iteratively evolve "executable specifications" in the form of test cases, create "good enough" models and then code, and, finally, confirm their design through testing the program (Ambler & Lines, 2012). Thus, TDD is not merely a testing method that focuses on reducing errors and rework, but an approach that facilitates better designs.

2.2 Software Quality and Productivity

Several researchers have examined the impact of TDD practice on software quality. Lui and Chan (2004) found that TDD greatly improves the software development process by enabling objective task estimation and progress tracking through rapid feedback. The test suite created early on in the development process provided an early alert system and made it easier for developers to take corrective action when they deviated from their goals. Overall, this resulted in a superior quality of software as the end result. A meta-analysis investigated the impact of TDD on external code quality and productivity and found marginal improvement in quality and little to no change in productivity (Rafique & Misic, 2013). Wilkerson et al. (2012) did a quasi-experiment to compare TDD with code inspection. Using a 2x2 factorial design, they compared four conditions: code inspection alone, TDD alone, both, and none. They found code inspection to be more effective in reducing defects. Another study found that developers using TDD produced code of higher quality that passed 18% more functional tests than code developed using the traditional approach (George & Williams, 2004). This increase in quality, however, was associated with a slight reduction in productivity. Software developed using TDD has also been found to have lower computational complexity and higher test volume and coverage, as compared to that developed using the traditional approach (Janzen & Saiedian, 2006). Crispin (2006) also reports a reduction in the defect rate of as much as 62% in projects that used TDD. Muller and Hagner (2002), on the other hand, did not find any change in quality or productivity; however, they found the resultant code to be better suited for reuse.

TDD has also been studied in an industrial setting. In two case studies conducted at Microsoft, Bhat and Nagappan (2006) found that the TDD approach reduced the number of defects per KLOC (thousand lines of code) by nearly four times, while the effort required went up by only 15%. Test coverage increased by 88%, thus significantly enhancing software quality. The results from different studies on TDD are summarized in Table 1 and show that the research findings regarding the impact of TDD on software quality and productivity are inconclusive: almost half show quality improvement, whereas the other half found no change or a drop in quality. There are several plausible reasons for the inconclusive findings in this body of literature. First, some of the studies used small sample sizes, which might have resulted in low statistical power. Second, several studies used self-reported data, which can potentially lead to weak control and may thus affect outcomes. To overcome these limitations and to shed light on this phenomenon, we employed a robust experimental design that uses randomization and an adequate sample size.

Table 1. Summary of Empirical Research on Test-Driven Development

Study	Impact of test-driven development				
	S/W quality	**Productivity**	**Setting / method**	**Sample size**	**Benchmark**
Bhat & Nagappan, 2006	Improvement by a factor of 2	-15% to -35%	Industry (case study)	Team 1 = 6 Teams 2 = 5-8	Non-TDD projects
Canfora et al., 2006	Inconclusive	-65%	Industry (experiment: 2 tasks, each 5 hours long)	28	Test after coding group (TAC)
Edwards, 2004	+45%	-90%	Academic (year long experiment)	59 students first traditional, then next year TDD	The same students did both. TL was control.
Erdogmus et al., 2005	No difference	+22% (though not statistically significant)	Academic (experiment: take home task)	TF= 11 TL = 13	TL was the control group
Fucci et al., 2017	Granularity and uniformity influence S/W quality. Sequencing and refactoring as other independent variables.	Granularity and uniformity influence productivity. Regression model is significant.	Industry workshops (3 tasks at 2 places, multiple runs)	Company A: 17 Company B: 22 (Collected 82 data points from 39 participants)	No control group
George & Williams, 2004	+18%	-16% (minor correlation reported statistically)	Industry (structured experiments)	TDD: 6 pairs; TL: 6 pairs; Total = 24	TL was the control group
Janzen & Saiedian, 2006	+16%	+57%	Academic (experiment: take home project)	3 teams of 3-4 students each, total = 10	TL was the control group
Madeyski, 2010	No statistically significant improvement	Not reported	Academic (experiment: take home assignment)	TF = 10; TL = 9, total = 19.	TL group
Madeyski, 2005*	-38%	N/A	Academic (experiment)	TF: 28, classic approach: 28, total = 56	Classic TL approach was used as benchmark
Muller & Hagner, 2002	No difference	No difference, but TDD was more efficient in implementation phase	Academic (experiment)	TDD: 10, traditional: 9; total 19	Traditional was the control group
Pancur & Ciglaric, 2011*	No difference	No difference	Academic (experiment: first part take home assignment, then final exam)	Part 1 (home assignment 5 weeks): TDD =14, ITL = 9; Part2 (final exam 4 hrs): TDD=14, ITL = 18	ITL was used as control group. Different number of stories to different groups.
Rafique & Misic, 2013	A little improvement	Little to no difference	Both (metastudy)	Meta analysis of 27 studies	Not applicable
Wilkerson et al., 2012	TDD results in inferior quality compared to code inspection	Code inspection is more expensive than TDD	Academic	7 (neither), 9 (TDD), 6 (code inspection), 7 (both), total = 29	TDD compared with code inspection

Notes: Acronyms used: ITL = iterative test-last approach, TAC = test after coding, and TL = test-last, TF = test-first, TDD = test-driven development. All terms other than TDD were used in papers to refer to the traditional approach of code development
*Pancur & Ciglaric (2011) and Madeyski (2005) used individuals and pairs, we have only included samples on individuals to maintain comparison equivalence

2.3 Developer Satisfaction

Software development is a cognitively challenging task and developers are known to experience burnout caused by job stress (Sonnentag, Broadbeck, & Stolte, 1994). Furthermore, given the shortage of talent in the software industry and the high turnover that the industry experiences, it is difficult for organizations to retain competent developers (Westlund & Hannnon, 2008). The key to mitigating turnover and enhancing organizational commitment is to ensure that developers are satisfied with their jobs, which is likely to occur if they derive satisfaction from engaging tasks that are both challenging and motivating (Locke & Latham, 1990). While satisfaction as a consequence of job characteristics has been extensively studied (Melnik & Maurer, 2006; Morris & Venkatesh, 2010; Tripp, Riemenschneider, & Thatcher, 2016), there is a dearth of empirical studies in information systems (IS) that examine individual satisfaction at the task level. Notable exceptions are the studies on pair development by Balijepally et al. (2009) and Mangalaraj et al. (2014). Our study extends this stream of research by investigating the impact of TDD on developer task satisfaction. In the context of our study, developer satisfaction is defined as the affective response of the programmer to the overall task of software development. In other words, our study assesses how developers feel about the tasks in which they engage.

2.4 TDD and Kolb's Experiential Learning Model

In a knowledge-driven economy, the long-term viability of an organization depends on its ability to learn, adapt, sense, and anticipate threats and opportunities in the marketplace. Practices such as TDD implicitly subscribe to the view that design evolves through discourse rather than being an a priori commitment to a given end. An important consequence of this perspective is that there is almost immediate feedback that either affirms or disconfirms the effectiveness of the design alternatives being considered. Such an approach not only helps detect and fix design and programming errors early in the development process but also provides an environment in which developers can collectively engage, learn, and grow because they immediately observe the results of their design choices and understand the efficacy of their actions. Given the imperative for organizations to evolve epistemically, it is critical to investigate the potential of contemporary software development practices such as TDD for conferring learning capabilities on knowledge workers engaged in cognitively demanding tasks.

Constant testing, which entails a continually evolving software because developers get rapid feedback from creating and running test cases, provides a mechanism for experiential learning through heightened developer involvement. Experiential learning is a useful synthesizing approach for building not only technical skills but also business dexterity to solve complex problems (Cameron & Purao, 2010). Kolb's experiential learning theory (ELT) (Kolb, 1976) provides an appropriate theoretical foundation for understanding TDD's impact on learning. According to ELT, learning occurs as a consequence of a continuous circular loop that has four distinct stages: concrete experience, reflective observation, abstract conceptualization, and active experimentation (Kolb, 1976) (see Figure 2).

Concrete experience concerns a new experience encountered by a learner in a specific situation. It could also involve a reinterpretation of an existing experience. Reflective observation entails the sensemaking stage during which the new experience is compared with existing understanding, with particular emphasis on the inconsistencies between the two. The abstract conceptualization stage is a creative stage that builds upon the previous two stages to envisage a novel solution. The novel solution could be an entirely new idea or a modification of an existing abstract concept. During the active experimentation stage, the solution developed in the previous stage is applied to a real-world scenario. The active experimentation stage then generates input for a new concrete experience stage.

The four stages of Kolb's ELT, namely, concrete experience, reflective observation, abstract conceptualization, and active experimentation are reflected to a large extent in the iterative process advocated by TDD. Developers using the TDD approach have to: (1) continually evolve designs to solve complex and often novel problems, (2) test their designs immediately by coding their design solutions, (3) repeatedly use feedback and reflection to reconceptualize their design strategies and test the efficacy of these strategies by implementing solutions, and (4) streamline and improve the quality of the software. Recognition and correction of errors based on immediate feedback provide an opportunity for learning and reflection (Argyris & Schon, 1978). These cyclical steps in TDD are conducive to Schon's notion of "reflection-in-practice" (Schon, 1983), thus providing a climate for learning as the software is progressively elaborated.

From the perspective of Kolb's ELT, learning is said to occur when the learner oscillates between the roles of an involved actor and a detached observer as he or she moves from specific instances to abstract generalizations. The learning cycle continues when these generalizations guide the decisions and actions toward specific tasks. The TDD technique forces such oscillation of roles at the cognitive level of the developer. Thus, much like ELT, TDD follows a cyclical process of development in which developers

continually play the dual roles of coder/tester as they develop code, receive feedback, reflect on their actions, and improve the quality of the software system. Learning, based on active experimentation guided by rapid feedback, is therefore an integral part of the TDD process.

3 Research Model

Our research model is presented in Figure 3. We compare TDD with the test-last approach using two dependent variables: software quality and task satisfaction of the developer. The goal of our study is to evaluate TDD as a software development approach with a focus on overall quality, including program design. While it is possible to compare TDD with other approaches such as test-last, coding with inspection (e.g., Fagan's approach as outlined in Wilkerson et al., 2012), and other variations, we chose to evaluate the performance of TDD vis-à-vis the test-last approach to make our findings comparable with much of what has been empirically tested previously. One of the reasons for not comparing it with Fagan's approach was to avoid introducing another source of variability into the study in the form of code reviewers and their abilities. Furthermore, the code review process has many variations in terms of team size and inspection method (Porter et al., 1997), which can also pose a challenge in developing a baseline to be used as a benchmark.

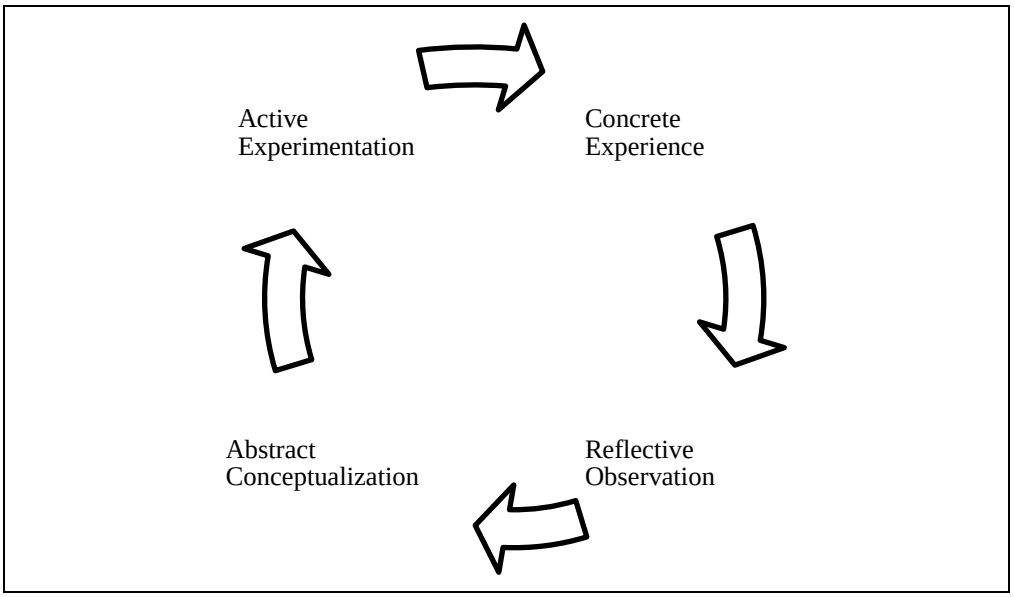

Figure 2. Experiential Learning Model (Adapted from Kolb, 1976)

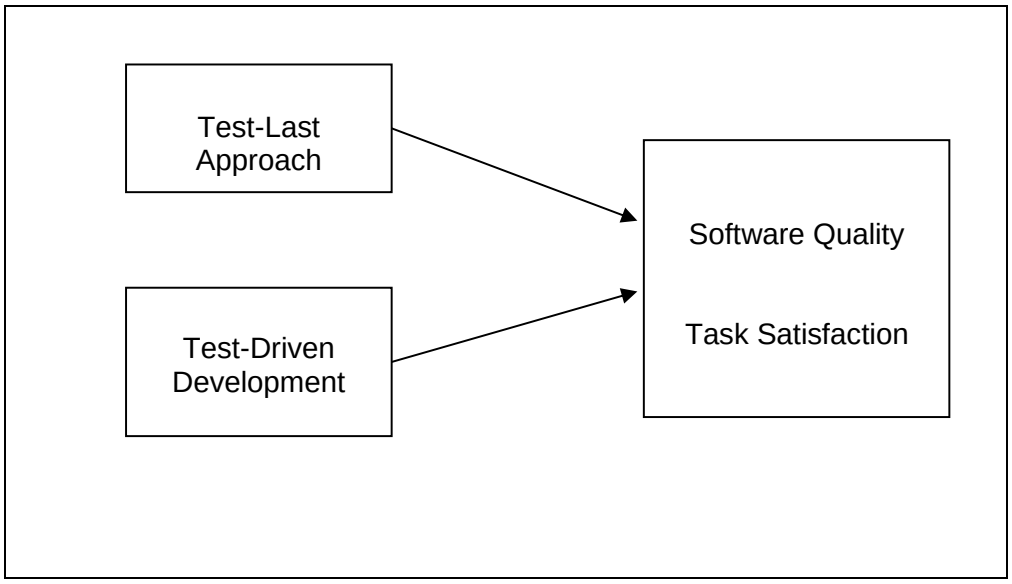

Figure 3. Research Model

Software quality is one of the main dependent variables used in prior empirical studies. However, software quality has not been measured consistently across all studies. A few have used the number of defects as a measure of quality (Bhat & Nagappan 2006; Edwards, 2004) and others have assessed functional correctness (Fucci et al., 2017; George & Williams, 2004) or acceptance testing (Pancur & Ciglaric 2011; Maydeski 2010, 2005; Erdogmus, Morisio, & Torchiano, 2005). While some studies view TDD as a defect-reduction technique, we take the more expanded view of TDD as a design strategy that can lead to superior program design. As Janzen and Saiedian (2006, p. 44) rightly note, "test-driven development focuses on how TDD leads analysis, design, and programming decisions." This view has also been endorsed by Wilkerson et al. (2012).

In addition to measuring software quality, our study distinguishes itself from prior research by assessing the influence of TDD on task satisfaction. In a field where developers are increasingly prone to burn-out, it is desirable to adopt approaches that can increase satisfaction at work, leading to greater engagement and commitment while reducing turnover intentions (Armstrong, Brooks, & Riemenschneider, 2015; Moore, 2000). Given this imperative, our study assesses the impact that TDD has on task satisfaction. We also measured learning outcomes and time to completion to assess learning and productivity, respectively; these analyses are presented in Section 7.

4 Hypotheses

In TDD, the processes of designing and coding are intertwined and code is developed in iterative cycles. Such incremental code development enables software developers to focus on one aspect of design (and its resultant code) at a time. Unit testing helps the developer to quickly identify not just errors in the code but also flaws in conceptualization and design (Beck, 1999; Dustin, 2002). It can therefore be argued that TDD enables developers to catch errors early in the development process, thus making it easier to identify the source of the problem. Repeated cycles of design-code-reflect-refactor ensure that working software gets tested frequently and is continually improved (Rafique & Misic, 2013). Furthermore, the TDD approach—often in combination with continuous integration—uses repeated testing and ensures complete test coverage, thus precluding new additions to the code from breaking the existing functionality.

Scott Ambler, a well-known methodologist, recommends TDD as a strategy for developing code that embodies good design and is easy to maintain (Ambler & Lines, 2012). He regards it as a critical practice that enhances the quality of code. Continual cycles of problem framing, code evolution, and problem reframing based on progressive insights lead to a reflective practice that yields better solutions (e.g., Schon, 1983).

In summary, the main distinction between TDD and the test-last approach is that the former requires the upfront development of test cases and use of the code-test-refactor cycle to successively develop and refine the code. On the other hand, a developer following the test-last approach may use a few iterations to modify the code in order to remove defects and meet stated requirements, but the overall software design is seldom refined based on the insight gained from testing. Refactoring, a practice that improves the quality of code and makes it more maintainable (see Ambler & Lines, 2012), is not an integral part of the test-last approach. In contrast, TDD is a design approach that repeatedly confirms that requirements embodied in test cases are satisfied because developers evolve code in test-code-refactor cycles (Shalloway et al., 2009; Ambler & Lines, 2012). Developers benefit from immediate feedback on the implementation of design choices, giving them an opportunity to improvise and refine their thinking in order to produce high-quality code.

In light of the preceding discussions, we hypothesize:

H1: While working on a programming task, programmers using TDD will produce software of higher quality than those using the test-last method of software development.

Locke and Latham (1990) use goal theory to assert that individuals working on a task experience satisfaction when they are successful in accomplishing task-related goals, and we argue that this can help improve the understanding of the influence of TDD on developer satisfaction. As has been argued in the literature, TDD is a design philosophy driven by test cases that embody functional requirements as well as user acceptance criteria (Crispin, 2006; Janzen & Saiedian, 2006). What distinguishes TDD from the test-last approach is that developers using TDD continually set achievable goals through test cases and write code that satisfies those tests. TDD facilitates the fulfillment of incremental goals as the project unfolds. In other words, developers using TDD repeatedly frame and reframe the problem through the articulation of small, clear goals in the form of test cases with well-defined acceptance criteria that they endeavor to satisfy through coding. The tangible fulfillment of each test case enables developers to confirm that their performance, in terms of the predefined goals, is successful, thereby leading to greater task satisfaction.

The theoretical underpinnings of self-determination theory (SDT) by Deci and Ryan (2000) lend further credence to the positive association between TDD and satisfaction. According to SDT, intrinsic motivation and its attendant benefits, such as well-being and

satisfaction, accrue when fundamental needs like autonomy, competence, and relatedness are fulfilled (Ilardi et al., 1993). While autonomy and relatedness may not be pertinent to our research context, competence is certainly a factor in promoting satisfaction among TDD subjects. It may be argued that TDD facilitates reflective practice (see Schon, 1983) because developers receive immediate feedback on the results of their design choices. Repeated feedback engendered by an inherently iterative process enables TDD developers to continually improvise and expand their capabilities, thus leading to greater confidence in their outcomes (i.e., competence). This should generate higher levels of motivation, which, in turn, should lead to greater satisfaction. This reasoning resonates with Buchan, Li, and MacDonell's (2011) finding that TDD users not only perceived improved quality of code and higher levels of productivity, but also experienced increased motivation and satisfaction.

Unlike TDD, the test-last approach neither facilitates the incremental attainment of goals nor does it provide repeated feedback on design alternatives. Furthermore, the linear sequence of activities, from analysis to design to coding to testing, does not give subjects using the traditional approach an opportunity to progressively refine their design in light of errors they uncover during testing. Given this backdrop, we expect TDD to result in greater overall task satisfaction when compared with the test-last approach. Therefore, we hypothesize:

H2: While working on a programming task, overall task satisfaction of programmers using TDD is higher than the overall task satisfaction of those using the test-last method of software development.

5 Research Methodology

5.1 Experimental Design

We conducted a controlled laboratory experiment to validate the research model because it allows better control over potentially confounding extraneous factors, thus leading to precise measurements of the variables. The experiment involved two programming tasks—a warm-up task followed by the main task. The warm-up task required the participants to create an application for a movie rental business and the main task consisted of developing an application for a bookstore. Detailed task descriptions are provided in Appendices A and B.

Undergraduate and graduate students majoring in information systems or computer science participated in the experiment. Each participant was randomly assigned to one of the two groups. Participants in one group developed the solution using the traditional test-last method of software development, while participants in the other group used TDD for the same purpose. Randomization of the assignment was performed to ensure that the study was not influenced by any potential bias. Power analysis suggests a group size of 42 per condition for a large population effect size at a 0.05 significance level (Cohen, 1992).

Students participating in the experiment were already familiar with the traditional software development process but were not knowledgeable about TDD. In order to familiarize all students with the TDD approach, a tutorial session was offered by one of the authors. Following this session, the students completed an assignment on using JUnit test cases to verify that they had adequate knowledge and skills. Thereafter, they were allowed to participate in the experiment.

A total of 88 students participated in the experiment. Participation was completely voluntary. To encourage participation, extra credit was given to the students by their respective instructors. The students who chose not to participate in the experiment were allowed to complete an alternate assignment of equal credit. The experiment was conducted following a script that included informed consent and debriefing. Results from four participants were excluded from data analysis for various reasons. One person fell sick during the experiment and could not finish the main task. Three others did not completely respond to the questionnaire used for data collection. Thus, the final data analysis included responses from 84 subjects. The mean age of the participants was 26.06 years with a standard deviation of 5. Demographic details about the participants are shown in Table 2.

5.2 Experimental Setting and Procedure

Prior to the main experiment, we conducted a pilot test using four subjects to clarify the experimental protocol. Two of the participants used TDD, while the other two followed the traditional method of software development. Minor changes were made to the protocol based on the feedback received from the pilot study. During the main experiment, participants were supervised to ensure that no socializing occurred. Based on the observations from the pilot test, the participants were allowed up to 30 minutes for the warm-up task and up to two hours to complete the main task. Laptop computers with Eclipse IDE (integrated development environment) were provided to all participants. JUnit test cases were enabled only in machines that were used by the participants using TDD. Internet access was disabled to prevent subjects from searching for solutions online. However, participants did have access to the JAVA API provided by Eclipse. Subjects in the control group were specifically instructed to use the test-last approach, whereas those in the treatment group were told to use TDD. The latter were informed that the JUnit test suite was already installed within the Eclipse environment on their computers. The control group did not have access to the JUnit test suite. All subjects were instructed to submit working code that met the stated requirements.

Table 2. Demographic Details of Participants

Demographic variable	Number of subjects	Percentage
Gender		
Male	62	73.8%
Female	22	26.2%
Education		
Undergraduate	49	58.3%
Graduate	35	41.7%
Programming experience		
<1year	46	54.7%
1 year-2 years	22	26.2%
2 years-3 years	9	10.7%
> 3 years	7	8.3%
Java experience		
< 1year	56	66.7%
1 year-2 years	19	22.6%
2 years-3 years	3	3.6%
> 3 years	6	7.1%

5.3 Dependent Variables

Software quality and task satisfaction were used as the dependent variables in the main model. Learning outcomes and time taken to complete the main task were also measured and used for additional analyses presented in Section 7.

We assessed software quality based on quality of code developed during the main task and developed a rubric (see Appendix D) to guide code quality assessment. Consistent with our objective to evaluate quality holistically, we considered high-level abstractions (e.g., classes required for the solution) and appropriate methods for each class. We also evaluated syntactic correctness and the quality of design elements such as interfaces, maintainability, and functionality. Thus, our assessment rubric goes beyond counting defects. This is consistent with the evaluation procedure followed by Balijepally et al. (2009). Furthermore, it resonates with the assessment approach used in Purao, Storey, and Han (2003) that penalizes missing items or incorrect designs and rewards good extensions to the basic design. As indicated by the rubric in Appendix D, the solutions were evaluated on a scale of 0 to 125 and assessments were based on the correctness of object-oriented design, implementation of the user interface and appropriate methods, and conformance of the solution to stated requirements. Points were added for good design decisions and deducted for poor design choices, thus ensuring proper assessment of software quality. We trained two information systems doctoral students who were not related to the study as raters and provided them with the detailed rubric in order to facilitate consistency in evaluating the solutions. The scores assigned by the two graders were checked for internal consistency using the Pearson correlation coefficient, which was found to be 0.791.

We measured overall task satisfaction using a prevalidated instrument reported in Balijepally et al. (2009). Participants were asked to report their overall experience in performing the main programming task using a 7-point Likert scale with responses ranging from *very dissatisfied* to *very satisfied*, *very displeased* to *very pleased*, *very frustrated* to *very contented*, and *absolutely terrible* to *absolutely delighted*. The satisfaction instrument is presented in Appendix C5.

6 Analysis and Results

The comparison between programmers using the test-last method of software development and those using TDD was designed to reveal important and significant differences between the two methods. MANOVA and ANOVA were used for identifying these differences between the two groups.

6.1 Factor Analysis

Cronbach's alpha is an indicator of internal consistency and homogeneity of a measured variable (Kerlinger, 1986) and values over 0.7 are considered adequate for assuming reliability (Nunnally, 1978). The four items used to measure overall task satisfaction were checked for internal consistency, and Cronbach's alpha was found to be 0.956. Table 3 shows the mean values, standard deviations, and the correlation matrix for the items used for this perceptual

measure. Exploratory factor analysis was performed using principal component analysis. All four items were found to load onto a single factor. Table 4 shows the factor loadings found as a result of using principal component analysis, along with Eigen value and variance explained. Since high factor loadings were found, a composite score for overall task satisfaction was used. The item scores were summated and then averaged to compute the composite score, which was used in the subsequent analysis.

6.2 Assumption Check

Before proceeding with the statistical analysis, we performed checks for assumption violation. In ANOVA, three assumptions must be met in order to sustain statistical significance in substantiating hypothesized claims. These are constancy of error variance, independence of error terms, and normality of error terms (Kutner et al., 2005). The *F*-test is considered to be fairly robust against violations of equal error variance in a fixed ANOVA model if the factor-level sample sizes are approximately equal or not significantly different (Kutner et al, 2005). Since, in this study, sample sizes across the comparison were equal, departure from equal variance does not represent a threat to generalization. Upon checking for violations of normality, minor violations were found in some cases and transformations were applied as a remedy. Exponential transformation, with an exponent value of 2.5 alleviated the problem of normality violation. Upon examining the residual plots, no violation of the independence of error terms was found.

MANOVA provides a measure against inflated Type 1 errors; hence, testing for its significance before proceeding with ANOVA analyses is recommended (Hair et al., 2006). Once the significance of the MANOVA test is established, ANOVA tests subsequently follow to determine which of the dependent variables are significant. Therefore, we used MANOVA analysis using all dependent variables to compare the performance of the two groups. The results summarized in Table 5 show that the MANOVA model was significant. We then performed one-way ANOVA testing and present the results in Table 6.

Hypothesis 1 predicted that software quality scores would be higher for those who used TDD than for those who used the test-last method of software development. Based on the analysis presented in Table 6, the performance of programmers using TDD was found to be significantly higher than that of programmers using the test-last method of software development. On average, participants using TDD scored 93.73 while those using the test-last method scored 76.06 on the software quality measure. The ANOVA test resulted in an *F*-value of 13.55 with a *p*-value of 0.00 (significant at 0.01).

Hypothesis 2 predicted that participants using TDD would score higher in terms of task satisfaction compared to those using the test-last method. On average, software developers using TDD scored 5.64, whereas those using the test-last method scored 4.72. The difference between the two scores was found to be statistically significant and the ANOVA test resulted in an *F*-value of 7.85 with a *p*-value of 0.003 (significant at 0.01). The results of hypothesis testing are summarized in Table 7.

Table 3. Correlation Matrix: Satisfaction

	Mean	*SD*	Item 1	Item 2	Item 3	Item 4
Item 1	5.50	1.506	1.0			
Item 2	5.48	1.558	0.885	1.0		
Item 3	5.36	1.695	0.873	0.779	1.0	
Item 4	5.29	1.492	0.853	0.833	0.889	1.0

Table 4. Factor Loadings – Satisfaction

Questionnaire item	Factor loadings	Communality estimate
Item 1	0.958	0.917
Item 2	0.927	0.859
Item 3	0.939	0.882
Item 4	0.948	0.899
Eigen value	3.557	
Variance explained	88.91%	

Table 5. MANOVA Results

Statistical test	Value	*F* value	Degrees of freedom		Sig. *p*-value
			Between group	Within group	
Pillai's trace	0.197	3.829	5	78	0.004*
Wilk's lambda	0.803	3.829	5	78	0.004*
Hotelling-Lawley trace	0.245	3.829	5	78	0.004*
Roy's largest root	0.245	3.829	5	78	0.004*
Note: *significant at $p = 0.05$					

Table 6. ANOVA Results

Dependent measure	Test-driven development		Test-last method		*F* value	Sig. *p*-value
	Mean	*SD*	Mean	*SD*		
Software quality	93.73	17.40	76.06	25.78	13.55	0.000*
Overall task satisfaction	5.64	1.31	4.72	1.68	7.85	0.003*
Note: *significant at $p = 0.05$						

Table 7. Results of Hypothesis Testing

Hypothesis	Finding
H1: While working on a programming task, programmers using TDD will produce software of higher quality than those using the test-last method of software development.	Supported ($p < 0.01$)
H2: While working on a programming task, overall task satisfaction of programmers using TDD is higher than the overall task satisfaction of those using the test-last method of software development.	Supported ($p < 0.01$)

7 Additional Data Analysis

Our study clearly demonstrates the efficacy of TDD in terms of software quality and task satisfaction. However, some questions still remain. For instance, prior studies have been inconclusive regarding the effect of TDD on productivity. The question that presents itself is whether higher software quality and task satisfaction come at the expense of productivity. In addition, given similarities between TDD and Kolb's experiential learning model (1976), it seems reasonable to expect that those engaged in TDD will experience greater learning outcomes. In this section, we inquire into these two questions:

1. *How does TDD influence productivity?*
2. *Does TDD facilitate learning?*

7.1 TDD and Productivity

7.1.1 As mentioned above, prior studies (see Table 1) have found that the productivity of TDD varies widely vis-à-vis the test-last method. There are several plausible reasons for this, including lack of control, small sample sizes, and other measurement issues. In our study, we measured the time taken to complete the main task as a surrogate for productivity but did not develop hypotheses related to productivity because we lacked theoretical justification to support such an argument.

7.1.2 We conducted ANCOVA (Neter et al., 1996) to compare the effects of the two groups (TDD and test-last) on quality using time as a covariate and found the two to be significantly different (*p*-value = 0.001). The result is presented in Figure 4, which shows software quality and time to completion for all observations with Qual_TDD and Qual_TLast,

indicating TDD and test-last experimental conditions, respectively. The trend lines for the two conditions are also plotted. The result clearly demonstrates that, for a given completion time (productivity level), TDD results in higher quality compared to the test-last development method. We further examined the data and found that among participants scoring more than 80 points (out of a maximum of 125) on software quality, those who used TDD far outnumbered those who used the test-last method (35, or 83.3%, vs. 20, or 47.6%). Thus, TDD appears to result in higher code quality without loss of productivity.

7.2 TDD and its Impact on Learning Outcomes

In this section, we present our analysis of the impact of TDD on learning outcomes. Following Gemino (1999), we assessed learning outcomes at three levels—verbatim recall, comprehension, and problem solving. Verbatim recall refers to the ability of the programmer to recall key words or key concepts learned while working on a programming task. Comprehension is the ability to understand key attributes—namely, classes, objects, methods, and their relationships. Finally, the capacity to apply the knowledge gained while working on a programming task to a new scenario is indicative of the problem solving ability of the programmer.

As discussed earlier, the cycle of code development used in TDD has similarities with Kolb's experiential learning model. This creates the potential for learning to occur when a developer engages in TDD. Thus, we hypothesize that TDD will result in higher levels of verbatim recall, comprehension, and problem solving ability.

HLa: While working on a programming task, programmers using TDD will demonstrate higher levels of verbatim recall than those using the test-last method of software development.

HLb: While working on a programming task, programmers using TDD will achieve higher levels of comprehension than those using the test-last method of software development.

HLc: While working on a programming task, programmers using TDD will acquire superior problem solving ability than those using the test-last method of software development.

Learning was measured through a questionnaire that participants filled out following completion of the main task. The items in the questionnaire were developed based on prior literature (Mayer, 1989; Gemino, 1999). The questionnaires for all the experimental conditions are given in Appendix C. Three types of tests were used to measure learning—a cloze test, comprehension test, and a problem solving test. The ability to recall verbatim was measured using the cloze test (Mayer, 1989). In our study, we operationalized this by providing subjects with the original problem description with several keywords missing (see Appendix C2) and asking the subjects to fill in the blanks based purely on memory. Following Gemino (1999), the comprehension test consisted of questions designed to evaluate the subject's understanding of the main programming task.

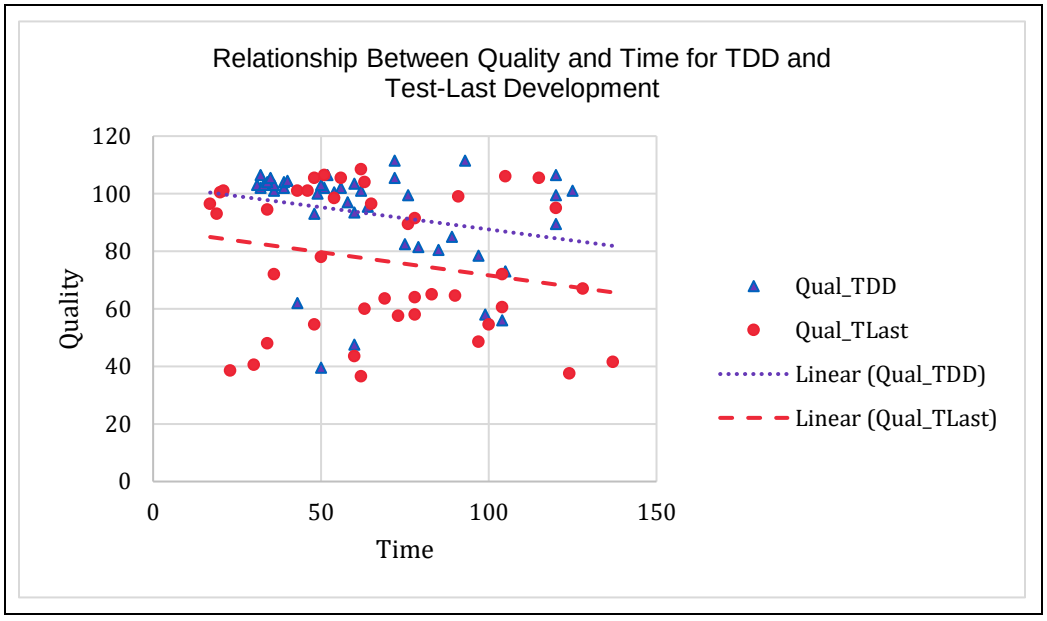

Figure 4. Relationship Between Software Quality and Time for Task Completion

Table 8. ANOVA Results: Learning

Dependent measure	Test-driven development		Test-last method		*F* value	Sig. *p*-value
	Mean	*SD*	Mean	*SD*		
Verbatim recall	7.71	1.70	7.26	1.87	1.342	0.125
Comprehension	6.60	1.49	6.02	1.44	3.175	0.038*
Problem solving	8.33	2.02	8.52	1.53	0.237	0.314
Note: significant at $p = 0.05$						

Comprehension was measured by items in the questionnaire that required the subject to identify objects, attributes of objects, and relationships among objects found in the main programming task (see Appendix C3). In order to assess the ability of subjects to apply their learning and comprehension to a new setting, we followed guidelines provided by Mayer (1989). Specifically, subjects were presented with a scenario that was different from the main programming task but offered opportunities to reuse lessons learned while performing the main task. Their responses to the questions (see Appendix C4) were used to evaluate their problem solving abilities when presented with an analogous situation.

Our approach to measuring learning outcomes is consistent with the extant literature (see, for example, Bostrom, Olfman, & Sein, 1990; Santhanam, Sasidharan, & Webster, 2008; Yi & Davis, 2003). Santhanam et al. (2008) assessed the effect of a self-regulated learning strategy using an experiment and measured learning outcomes using multiple-choice and fill-in-the-blank questions following the experiment. Bostrom et al. (1990) investigated training effectiveness with comprehension as a dependent variable. Comprehension was assessed using a multiple item quiz about the functions and features of the target software. Li, Santhanam, and Carswell (2009) assessed problem solving ability using questions about a new scenario.

The learning measures were included in the MANOVA reported in Table 5. We ran ANOVA to test the learning hypotheses and the results are shown in Table 8. On average, participants using TDD scored 7.71 on verbatim recall whereas those using the traditional method scored 7.26. The ANOVA test resulted in an *F*-value of 1.342 with a *p*-value of 0.125, which was not significant at 0.05. On average, participants using TDD scored 6.60 on the comprehension test, whereas those using the traditional method scored 6.02. The difference was statistically significant with an *F*-value of 3.175 and a *p*-value of 0.038 (significant at 0.05). Participants using the traditional method of software development scored 8.52 while those using TDD scored 8.33 on the problem solving test. Thus, HLa and HLc are not supported, but HLb is supported.

Our results show that TDD leads to a higher level of problem comprehension compared to the test-last development process. While our findings are interesting, learning is inherently a phenomenon that occurs over a long period of time and is hard to capture in a snapshot study. Developers learn over the entire length of time they spend working on projects when, for example, they solve problems that invariably occur in any software development endeavor. Thus, our findings must be interpreted appropriately and a future longitudinal study should be performed to reconfirm the impact of TDD on learning outcomes.

8 Discussion

Our study demonstrates that the TDD approach results in enhanced software quality when compared with the test-last approach and that this gain in software quality occurs without any loss of productivity. Furthermore, we found that subjects who used TDD were more satisfied than those who adopted the test-last approach. These findings are consistent with our hypotheses derived from a review of the extant literature. The superior software quality generated by TDD users may be attributed to the fact that it is not just a different testing practice but a design strategy that facilitates improvisation, because design ideas embodied in the test cases are continually refined as the code unfolds. While these findings are interesting, it must be kept in mind that the subjects in our experiment had limited programming experience, and, therefore, the results of our study are likely more applicable to entry-level developers.

As discussed above, TDD's cycle of software development is reminiscent of Kolb's experiential learning model. Given the similarities between TDD and Kolb's model, another plausible reason for the improved software quality of TDD users vis-à-vis subjects in the test-last condition is the greater opportunity for learning deriving from failed tests, adapting, and making necessary changes. The applicability of this finding to other domains such as product innovation is supported by a similar

observation by Eisenhardt and Tabrizi (1995) that "iterations and testing would rapidly build understanding and create multiple options" (p. 104); they showed that an experiential strategy using repeated iterations with frequent testing and improvisation leads to faster product innovation. We believe that demonstrating/affirming such empirical regularity across disciplines and/or multiple domains is an important step toward building robust theories.

The increased satisfaction of TDD subjects may be attributed to the continual attainment of milestones and incremental goals during the course of the development process. Furthermore, the upfront and iterative articulation of test cases and acceptance criteria clarifies and/or reaffirms subjects' understanding of the requirements before proceeding to write the code. Another plausible reason for the increased satisfaction of TDD subjects vis-à-vis test-last participants is the immediate feedback that the former receive regarding their actions. As discussed before, self-determination theory (Deci & Ryan, 1985) proposes that intrinsic motivation and its attendant benefits are likely to ensue when fundamental psychological needs such as autonomy, competence (influenced by immediate and frequent feedback), and relatedness are satisfied.

We also explored the impact of TDD on learning outcomes measured at three levels: verbatim recall, comprehension, and problem solving ability. Our findings are intriguing: Subjects using TDD demonstrated higher levels of problem comprehension compared to those using the traditional approach to software development, but no statistically significant difference was found between the performance of the two groups on verbatim recall and problem solving ability. A plausible reason for the lack of superior performance of the TDD group in verbatim recall may be attributed to the sophistication of the contemporary IDE (integrated development environment) Eclipse that we used in our experiment. Research has shown that a tool or model that helps manage factual data about a problem domain disincentivizes remembering facts about the problem, thus leading to lower ability to recall facts from memory (Mayer, 1989). The lack of performance difference regarding problem solving ability may be attributed to a limitation of our experimental design. Specifically, the transfer of problem solving skills from one task to another is best assessed by judging performance of the subjects in a follow-up task. However, we could not use a follow-up task in our study because of the time limit imposed by our experimental setup. Instead, we measured transfer of problem solving skills through a questionnaire. We believe that this approach of measuring skill transfer may have contributed to the confounding result. This issue could be further explored in a future study by using an appropriate research design.

9 Implications for Practice and Research

This study makes significant contributions to the practice of software development. Our research demonstrates that TDD not only enables developers to produce code of a higher quality but also helps them achieve higher task satisfaction. Higher quality code translates to fewer defects, less rework, and increased satisfaction of end users with the resulting information system. Organizations invest great monetary resources in software development, maintenance, and evolution. Minimizing defects and reducing maintenance related to rework can yield significant savings. Our study validates that TDD results in higher levels of satisfaction with the overall software development experience, as compared to the test-last method. In an industry where developers are under considerable stress deriving from changes and innovations in methods (Chilton, Hardgrave, & Armstrong, 2010), TDD appears to be an innovation that actually enhances task satisfaction. Higher satisfaction among developers can lead to higher morale and reduced turnover. Given these benefits, the widespread adoption of TDD for software development may be a fruitful strategy for organizations.

From a research perspective, our study makes a significant contribution to the information systems development literature. IS practitioners often lead the field in developing new techniques and methodologies based on their experience. Academics play a critical role in assessing the efficacy of such new practices through rigorous research. While there have been several studies that have assessed the efficacy of TDD, the results are inconclusive. Our research uses a rigorously designed and executed laboratory experiment to shed light on this phenomenon and to create a benchmark to investigate TDD and its variations. Another contribution of our study is the understanding of the impact of TDD on task satisfaction. Developer satisfaction based on software development tasks is an important but underexplored area of research that holds significant promise to identify avenues for enhancing job satisfaction, reducing burnout, and improving employee retention—all critical for improving the working condition and emotional well-being of the software development community. Finally, the investigation of learning as an outcome of a software development process is an important but unexplored area of research. Enhancing learning at the individual level can be beneficial in the long run in terms of improved quality of work leading to fewer defects. Though tentative, our preliminary findings on the impact of TDD on learning outcomes can serve as a catalyst to

spawn more research efforts focusing on improving the understanding of how learning occurs at the individual and group levels in software development.

10 Limitations and Future Research

The use of student subjects in our study raises some concern about the external validity and generalizability of the results. While students may not be adequate proxies for experienced software developers, they are good surrogates for entry-level developers (Balijepally et al., 2009), and student subjects have been widely used in experimental research involving software development (see, for example, Burton-Jones & Meso, 2006; Khatri et al., 2006; and Balijepally et al., 2009). Indeed, it has been argued that the similarities between students and practitioners engaged in processes consistent with organizational phenomena outweigh the differences between them (Locke, 1986). Nevertheless, we acknowledge this as a limitation of our study, and future studies should replicate this research using practitioners as subjects.

In a similar vein, the limited programming experience of our subjects may also affect the generalizability of our findings. We recommend that future empirical studies use experienced developers as subjects to confirm the validity of our results. Furthermore, our assessment of learning outcomes, though consistent with the extant literature, may be considered somewhat tentative. It could be argued that learning is best assessed through longitudinal studies. Thus, this remains an open research question for further validation using alternate theoretical framing and research designs. We used a single task to evaluate our research model. It may be worthwhile to study the efficacy of TDD under conditions of varying task complexity. Introducing different levels of task complexity may help tease out differences in the way that subjects learn. It may also be useful to examine the interplay between the dynamics of task complexity and the learning styles of individuals. Our study demonstrates that TDD leads to a higher quality of software and increased satisfaction among developers in terms of the coding process. Future studies should examine the possible process variables that may account for these relationships.

11 Conclusion

TDD offers a novel approach to software development. Research on the efficacy of TDD has been found to be inconclusive; some studies show a gain in performance whereas others find no change or even a decrease in performance vis-à-vis the traditional approach to software development. We conducted a laboratory experiment to compare the efficacy of TDD with that of the test-last approach and found that TDD not only leads to the development of higher-quality software but also results in greater levels of satisfaction with the development task among software developers. Through a post hoc assessment, we studied the impact of TDD on productivity and learning. The findings of this study have important implications for practice and research, and the influence of software development processes on learning is an unexplored area that holds significant promise given the current emphasis on creating learning organizations.

Acknowledgments

The authors thank Dr. Mark Eakin of the University of Texas at Arlington for his help with the analysis of the productivity data reported in Figure 4.

References

Ambler, S., & Lines, M. (2012). *Disciplined agile delivery: A practitioner's guide to agile software delivery in the enterprise*. IBM Press.

Aniche, M., & Gerosa, M. A. (2015). Does test-driven development improve class design? A qualitative study on developers' perceptions. *Journal of Brazilian Computer Society*, *21*(15), 1-11.

Argyris, C., & Schon, D. A. (1978). *Organizational learning: A theory of action perspective*. Addison-Wesley, MA.

Armstrong D. J., Brooks N. G., & Riemenschneider, C. K. (2015). Exhaustion from information system career experience: Implications for turn-away intention. *MIS Quarterly*, *39*(3), 713-727.

Avgar A., Tambe P., & Hitt, L. M. (2018). Built to learn: how work practices affect employee learning during healthcare information technology implementation. *MIS Quarterly*, *42*(2), 645-659.

Balijepally V., Mahapatra R., Nerur S., & Price, K. (2009). Are two heads better than one for software development? The productivity paradox of pair programming. *MIS Quarterly*, *33*(1), 91-118.

Beck, K. (1999). *Extreme Programming explained: Embrace change*. Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile Software Development*. agilemanifesto.org

Beck, K. (2002). *Test-driven development: by example*. Addison-Wesley.

Bhat, T., & Nagappan, N. (2006). *Evaluating the efficacy of test-driven development: industrial case studies*. Proceedings of International Symposium on Empirical Software Engineering.

Bostrom, R., Olfman, L., & Sein, M. (1990). The importance of learning style in end-user training, *MIS Quarterly*, *14*(1), 101-119.

Buchan, J., Li, L., & MacDonell, S. G. (2011). Causal factors, benefits and challenges of test-driven development: Practitioners perceptions. *Proceedings of the 18th Asia-Pacific Software Engineering Conference*.

Burton-Jones, A., & Meso, P. N. (2006). Conceptualizing systems for understanding: an empirical test of decomposition principles in object-oriented analysis. *Information Systems Research*, *17*(1) 38-60.

Cameron, B. H., & Purao, S. (2010). Enterprise integration: An experiential learning model. *Information Systems Education Journal*, *8*(38) 1-13.

Canfora, A. (2006). *Evaluating advantages of TDD: A controlled experiment with professionals*, Proceedings of the International Symposium on Empirical Software Engineering.

Chilton, M. A., Hardgrave, B. C., & Armstrong, D. J. (2010). Performance and strain levels of IT workers engaged in rapidly changing environments: A person-job fit perspective. *Database for Advances in Information Systems*, *41*(1) 8-35.

Cockburn, A., & Highsmith, J. (2001). Agile software development: The people factor. *Computer*, *34*(11), 131-133.

Cohen, J. (1992). A power primer. *Psychological Bulletin*, *112*(1), 155-159.

Crispin, L. (2006). Driving software quality: how test-driven development impacts software quality. *IEEE Software*, *23*(6), 70-71.

Deci, E. L., & Ryan, R. M. (2000). The "what" and "why" of goal pursuits: Human needs and the self-determination of behavior. *Psychological Inquiry*, *11*, 227-268.

Deci, E. L., & Ryan, R. M. (1985). *Intrinsic motivation and self-determination in human behavior*. Plenum.

Dustin, E. (2002). *Effective software testing: 50 ways to improve your software testing*. Addison-Wesley.

Edwards, S. (2004). Using software testing to move students from trial-and-error to reflection-in-action. *ACM SIGSCE Bulletin*, 26-30.

Eisenhardt, K. M., & Tabrizi, B. N. (1995). Accelerating adaptive processes: Product innovation in the global computer industry. *Administrative Science Quarterly, 40*(1), 84-110.

Erdogmus, H., Morisio, M., & Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, *31*(3) 226-237.

Fucci, D., Erdogmus, H., Turhan, B., Oivio, M., Juristo, N. (2017). A dissection of the test-driven development process: Does it really

matter to test-first or test-last? *IEEE Transactions on Software Engineering*, *43*(7), 597-614.

Gemino, A. C. (1999). *Empirical comparison of systems analysis modeling techniques* (Unpublished doctoral dissertaion, University of British Columbia, Canada)

George, B., & Williams, L. (2004). A structured experiment of test-driven development, *Information & Software Technology*, *46*(5), 337-342.

Gilb T. (1989). *Principles of software engineering management*. Addison Wesley/Longman.

Hair, J. H., Black, W. C., Babin, B. J., Anderson, R. E., & Tatham, R. L. (2006). *Multivariate data analysis*. Prentice Hall / Pearson Education

Ilardi, B. C., Leone, D., Kasser, T., & Ryan, R. M. (1993) Employee and supervisor ratings of motivation: Main effects and discrepancies associated with job satisfaction and adjustment in a factory setting. *Journal of Applied Psychology*, *23*(21), 1789-1805.

Janzen, D. S., & Saiedian, H. (2006). On the influence of test-driven development on software design. *Proceedings of 19th Conference on Software Engineering Education and T*raining.

Kerlinger, F. N. (1986), *Foundations of behavioral research*. Holt, Rinehart, & Winston.

Khatri, V., Vessey, I., Ramesh, V., Clay, P., Park, S.-J. (2006). Understanding conceptual schemas: exploring the role of application and is domain knowledge. *Information Systems Research*, *17*(1), 81-99.

Kolb, D. A. (1976). Management and the Learning Process. *California Management Review*, *18*(3), 21-31.

Kutner, M. H., Nachtsheim, C. J., Neter, J., & Li, W. (2005). *Applied linear statistical models* (5th ed.). McGraw Hill.

Larman C., & Basili, V. (2003). Iterative and incremental development: A brief history. *Computer*, *36*(6), 47-56.

Locke, E. A. (1986). *Generalizing from laboratory to field settings*. Lexington Books

Locke, E. A., & Latham, G. P. (1990). Work motivation and satisfaction: Light at the end of the tunnel. *American Psychological Society*, *1*(4), 240-246.

Li, P., Santhanam, R., & Carswell, C. M. (2009). Effects of animations in learning: A cognitive fit perspective. *Decision Sciences Journal of Innovative Education*, *7*(2), 377-410

Lui, K. M., & Chan, C. C. (2004). *Test driven development and software process improvement in China: Extreme programming and agile processes in software engineering* (pp. 219-222). Springer

Madeyski, L. (2010). The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Information and Software Technology*, *52*(2), 169-184.

Madeyski, L. (2005). Preliminary analysis of the effects of pair programming and test-driven development on the external code quality. In K. Zelinski & T. Szmuc (Eds.), *Software engineering: Evolution and emerging technologies* (1st ed., pp. 113-123). IOS Press

Mangalaraj, G., Nerur, S., Mahapatra, R. K., & Price, K. H. (2014). Distributed cognition in software development: an experimental investigation of the role of design patterns and collaboration. *MIS Quarterly*, *38*(1), 249-274.

Mayer, R. E. (1989). Models for understanding. *Review of Educational Research*, *59*(1), 43-64.

Melnik, G., & Maurer, F. (2006). Comparative analysis of job satisfaction in agile and non-agile software development teams., In P. Abrahamsson, M. Marchesi, & G. Succi (Eds.), *Extreme Programming and Agile Processes in Software Engineering* (pp. 32-42). Springer.

Moore, J. E. (2000). One road to exhaustion: An examination of work exhaustion in technology professionals. *MIS Quarterly*, *24*(1), 141-168.

Morris, M. G., & Venkatesh, V. (2010). Job characteristics and job satisfaction: Understanding the role of enterprise resource planning system implementation. *MIS Quarterly*, *34*(1), 143-161.

Muller, M. M., & Hagner, O. (2002). Experiment about test-first programming. *IEEE Proceedings: Software, 149*(5), 131-136.

Neter, J., Kutner, M. H., Nachtsheim, C. J., & Wasserman, W. (1996). *Applied Linear Statistical Models* (4th ed.), WCB McGraw-Hill.

Nerur S., Mahapatra, R. K., & Mangalaraj, G. (2005). Challenges in migrating to the agile methodologies. *Communications of the ACM*, *48*(5), 73-78.

Nunnally, J. C. (1978). *Psychometric theory*. McGraw-Hill

Pancur, M., & Ciglaric, M. (2011). Impact of test-driven development on productivity, code, and tests: A controlled experiment. *Information and Software Technology*, *53*(6), 557-573.

Porter, A. A., Siy, H. P., Toman, C. A., & Votta, L. G. (1997). An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Transactions on Software Engineering*, *23*(6), 329-346.

Purao, S., Storey, V., & Han, T. (2003). Improving analysis pattern reuse in conceptual design: augmenting automated processes with supervised learning. *Information Systems Research*, *14*(3) 269-290.

Pressman, R. S. (2005). *Software engineering: A practitioner's approach* (6th ed.). McGraw-Hill.

Rafique, Y., & Misic, V. B. (2013). The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, *39*(6), 835-856.

Schon, D. A. (1983). *The reflective practitioner: How professionals think in action*. Basic Books.

Shalloway, A., Beaver, G., & Trott, J. R. (2009). *Lean-agile software development: achieving enterprise agility*. Addison-Wesley.

Santhanam, R., Sasidharan, S., & Webster, J. (2008). Using self-regulatory learning to enhance e-learning-based information technology training. *Information Systems Research*, *19*(1) 26-47.

Singh, P. V., Tan, Y., & Youn, N. (2011). A hidden Markov model of developer learning dynamics in open source software projects. *Information Systems Research*, *22*(4) 790-807.

Sonnentag, S., Broadbeck, T. H., & Stolte, W. (1994). Stressor-burnout relationship in software development teams. *Journal of Occupational and Organizational Psychology*, *67*, 327-341.

Tripp, J. F., & Riemenschneider, C. K. (2014). *Towards an understanding of job satisfaction on agile teams: Agile development as work redesign*. Proceedings of the 47th Hawaii International Conference on System Sciences.

Tripp, J. F., Riemenschneider, C. K., & Thatcher, J. (2016). Job Satisfaction in agile development teams: Agile development as work redesign. *Journal of the Association for Information Systems*, *17*(4) 267-307.

Wastell, D. (1999). Learning dysfunctions in information systems development: Overcoming the social defenses with traditional objects. *MIS Quarterly*, *23*(4), 581-600.

Westlund, S. G., & Hannon, J. C. (2008). Retaining talent: Assessing job satisfaction facets most significantly related to software developer turnover intentions. *Journal of Information Technology Management*, *19*(4), 1-15.

Wilkerson, J., Nunamaker, J. F., & Mercer, R. (2012). Comparing the defect reduction benefits of code inspection and test-driven development. *IEEE Transactions on Software Engineering*, *38*(3), 547-560.

Yi, M. Y., & Davis, F. D. (2003). Developing and validating an observational learning model of computer software training and skill acquisition, *Information Systems Research*, *14*(2), 146-169.

Appendix A: Warm-Up Task

For participants using the traditional method of software development:

A movie rental business owner has hired you as a software consultant and wants you to develop an application for him. The application should allow a way to create a list of movies. It should also allow for the addition of movies to the list. The order of the movie list is not important. The application should display the total number of movies listed at a time. You should get the output displayed on the monitor (command prompt).

A sample output would be:

No. of movies currently available: 5

Note: For participants using TDD: The same task was given with the following instruction inserted before the task description:

"You have to use TDD and write relevant unit test cases in developing the following application."

Appendix B: Main Task

For participants using the traditional method of software development:

The owner of a bookstore wants to keep records of the books in stock on the computer. The owner wants the application that would enable him to identify the books that are available in the store. You are required to develop an application that can be used to keep records of the books in stock.

There can be many different ways of identifying a book. The most direct way to identify a book is by its name. However, it might lead to a situation where two books may have the same name. Therefore, a book should also be described by a unique identifier number. The unique identifier number for the book should be an assigned integer.

The bookstore owner also wants the names of the author(s) to be available along with the name and unique identifier number of a book. A book could be written by one or more than one author. The name of an author consists of the first name and the last name. Since two authors may have the same name, an author should also be identified by a unique identifier number in addition to his or her name. The application should be so developed that it contains details about the authors; it should have the functionality to add author(s) to an existing book record.

In your application, you should have appropriate methods that will enable the user to get names and unique identifier numbers of books as well as the names and unique identifier numbers of the corresponding authors of these books. A book may have one or more than one author. The application should accommodate any number of authors for a book.

Your application should be able to display on console (at the command prompt) the information about the books and authors, a sample output of which is as shown below.

Book ID: 1234986

Book Name: Gravitational Relativity

Author1 ID: 653

Author1 Name: Issac Newton

Author2 ID: 474

Author2 Name: Albert Einstein

Note: For participants using TDD: The same task was given with the following instruction inserted before the task description:

“You have to use TDD and write relevant unit test cases in developing the following application.”

Appendix C: Questionnaire

11.1 Appendix C1: Demographic Questions

For individual participants using the traditional method of software development:

1. Please circle your gender:

Male

Female

2. Please indicate your age on your last birthday _____

3. Highest educational level (including currently pursuing degree):

a) High school b) Technical school or community college

c) Undergraduate degree d) Graduate degree

e) Doctoral Degree f) Other: _____

4. Indicate number of years of your programming experience in any programming language?

a) 0-1 b) 1-2 c) 2-3

c) 3-4 d) 4-5 e) more than 5

5. Indicate number of years of your programming experience in object-oriented languages?

a) 0-1 b) 1-2 c) 2-3

c) 3-4 d) 4-5 e) more than 5

6. What would you consider to be your level of experience in object-oriented programming?

a) No experience b) Novice

c) Intermediate d) Expert

7. What object-orient programming languages are you familiar with?

a) C++ b) C# c) Java

d) Small Talk e) Objective-C f) Eiffel

g) Python h) VB.NET i) Other _____

8. How comfortable are you with the IDE “Eclipse”?

a) Very comfortable b) Comfortable

c) Not much comfortable d) Not at all comfortable

For individual participants using TDD only:

9. What would you consider to be your level of experience in TDD?

a) No experience b) Novice

c) Intermediate d) Expert

11.2 Appendix C2: Verbatim Recall

Section A

Please fill in the blanks based on the description given in the main task:

The owner of a bookstore wants to keep records of the books in stock. The owner wants an application that would enable him to _____ the books that are available in the store by their _____. Additionally, a book should be described by a/an _____that should be an assigned _____. A book could be written by one or more than one author. The _____ of an author consists of the _____ and _____ names. But, that might lead to a

situation where two authors may have _____ names. So the author should also be identified by a / an _____ as well. The application should be so developed that it contains details about the authors, it should have the functionality to _____ the author or authors to the existing book records.

11.3 Appendix C3: Comprehension

Section B

Answer the following questions based on the description given in the main task.

1. Which fields (variables and references) are used in book class?

a. Book name

b. Book ID

c. Publisher

d. Both a and b

2. A Book object is identified by:

a. Book Name

b. Unique identifier number

c. Both a and b

d. Either a or b.

3. How many authors can be added to a book?

a. One

b. Two

c. As many as needed

4. Can Book object be added to an author?

a. Yes.

b. No.

c. Insufficient Information

5. An author is identified by:

a. Author name

b. Author ID number

c. Both a and b

d. Either a or b

6. Can we list all the books written by an author without going through the entire collection of books?

a. Yes

b. No

c. Insufficient information

7. Two authors who have the same name may be identified by:

a. First, middle, last name together

b. Unique identifier number

c. A randomly generated numeric value

8. When checking for the availability of a specific book, it is best to search by:

a. Name

b. Unique identifier number

c. Publisher

d. All of the above

9. If you want to store the publisher information in your application, which is a more appropriate place to store the information?

a. Book class

b. Author class

c. Publisher class

10. The application that you developed for the scenario is similar to which of the following:

a. Customers opening an account in bank

b. Students registering for classes in student information system

c. Customer receiving invoices

11.4 Appendix C4: Problem Solving

Section C

Please read the following scenario and answer the questions that follow:

A major international conference is to be organized in six months. The organizers of this conference have announced a call for papers. Many researchers are expected to submit their papers for publication in the conference journal. You are required to develop an application that can be used to keep records of the papers that are submitted to the conference. The organizers want to easily identify the submitted papers. The submitted paper can be identified by its title, but since two papers could have the same title, you should also identify the submitted paper using a unique identifier number. Your application should use an integer value for the unique identifier number.

Since the organizers wish to maintain the standard of the papers that are published in their conference journal, quality of the submitted work needs to be judged. For this purpose, the organizers have requested researchers to serve as reviewers. However, those who choose to volunteer as reviewers will not be allowed to submit their own papers. The submitted papers will be reviewed by the reviewers before being accepted for publication in the conference journal.

For the review process, the organizers should be able to assign each paper to the reviewers. Hence, a paper should have details about the reviewers. The papers could be reviewed by one or more reviewers and the conference organizers should be able to add the name or names of the reviewer or reviewers to a submitted paper. The name of a reviewer consists of first and last names. There could be a scenario of two reviewers with the same name, so in addition to the name, the reviewer should also be identified by a unique identifier number. The application should have the functionality to add the reviewer or reviewers to the existing records of the submitted papers.

1. As compared to the main task, a paper is analogous to:

a. Book

b. Author

c. Publisher

2. As compared to the main task, the organizer is analogous to:

a. Author

b. Owner

c. Publisher

3. As compared to the main task, the reviewer is analogous to:

a. Author

b. Publisher

c. Owner

4. You will resolve the issue of two reviewers with the same name by:

a. first, middle, and last name together

b. a randomly generated numeric value

c. Unique identifier number

5. Will ArrayList of authors will be similar to ArrayList of:

a. Papers

b. Reviewers

c. Organizers

11.5 Appendix C5: Task Satisfaction

Section D:

Please answer the following questions based on your experiences:

How do you feel about your overall experience of working on the programming task today?

Very Dissatisfied	1	2	3	4	5	6	7	Very Satisfied
Very Displeased	1	2	3	4	5	6	7	Very Pleased
Very Frustrated	1	2	3	4	5	6	7	Very Contented
Absolutely Terrible	1	2	3	4	5	6	7	Absolutely Delighted

Appendix D: Software Quality Rubric

S. No.	Description	Points
1.	*Book class evaluation:*	
a.	Variable declaration: name should be string, ID number should be string or int, ArrayLists used Deduct points for wrong variable types, syntax, use of arrays instead of ArrayLists	6 (4)
b.	Constructor: using a proper constructor (not default) Deduct points for return type, incorrect parameters, wrong assignment	8 (6)
c.	Gettor methods: for book name, book number Deduct points for return type, incorrect parameters, wrong assignment	6 (3)
d.	addAuthor method (author object should be passed as a parameter) Deduct points for wrong parameter, invalid return type, lack of functionality, syntax errors.	10 (10)
e.	getAuthor method (ArrayList of Authors should be iterated through and author object read one by one) Deduct points for wrong implementation of loop, invalid return type, lack of functionality, syntax errors.	15 (12)
2.	*Author class evaluation:*	
a.	Variable declaration: First and last name should be String, author ID number can be String or int Deduct points for each wrong variable type	6 (3)
b.	Constructor: using a proper constructor (not default) Deduct points for return type, incorrect parameters, wrong assignment	10 (6)
c.	Gettor methods: for author name, author number Deduct points for return type, incorrect parameters, wrong assignment	6 (3)
3.	*Display class with main method*	
a.	Creating book objects passing correct parameters (at least one Book object)	5
b.	Creating author objects passing correct parameters (at least two Author objects)	5
c.	Adding at least two author objects to book objects by calling addAuthor method	6
d.	Getting information from ArrayList() of Authors using getAuthorList() method	6
e.	Creating display at command prompt	5
f.	If program compiles correctly displaying required information without any error	5
4.	*Going beyond requirements*	
a.	Maintainability considerations, appropriate indentation, comments, etc.	5
b.	Using settor methods	5
c.	Creation of user interface using JOptions pane	6
d.	Creating additional class and/or methods to provide enhanced functionality	8
	Total max points possible	125

About the Authors

Vikram S. Bhadauria is an assistant professor of MIS at Texas A&M University in Texarkana, Texas. He received his PhD in information systems from the University of Texas at Arlington. His current research includes software development methodologies, environmental sustainability, security, blockchain, IoT, and self-driving technology adoption. His research publications appear in *Journal of Database Management, Computers in Human Behavior, Industrial Management & Data Systems, International Journal of Productivity and Quality Management, Management Research Review, Supply Chain Management: An International Journal*, and other journals. He has also presented papers at several international conferences.

RadhaKanta Mahapatra is a professor of information systems and chair of the ISOM Department at the University of Texas at Arlington. He holds a bachelor's degree in electrical engineering from the National Institute of Technology, Rourkela, India, a PGDM (MBA) from the Indian Institute of Management, Ahmedabad, India, and a PhD in information systems from Texas A&M University. His research interests include healthcare information systems, IT4D, agile software development and project management, data warehousing and business intelligence, and data quality. His research publications have appeared in *MIS Quarterly, Communications of the ACM, Decision Support Systems, Information & Management, European Journal of Information Systems*, and other journals. He is passionate about improving the conditions of marginalized populations around the world through the use of information technologies.

Sridhar Nerur is the Goolsby-Virginia and Paul Dorman Endowed Chair in Leadership and professor of information systems at the University of Texas at Arlington (UTA). He is also the chair of the Graduate Studies Committee on Business Analytics at UTA. His research has been published in premier journals/magazines such as *MIS Quarterly, Strategic Management Journal, Communications of the ACM, European Journal of Information Systems, Information & Management, IEEE Software,* and *Journal of International Business Studies.* He has served on the editorial boards of the *European Journal of Information Systems and the Journal of Association for Information Systems*. His research and teaching interests include social networks, machine learning/AI/deep learning, text analytics, neuroeconomics, and agile software development.