An Experimental Evaluation of ITL, TDD and BDD

Luis A. Cisneros, Marisa Maximiano, Catarina I. Reis
Computer Science and Communication Research Centre
Polytechnic of Leiria
Leiria, Portugal
email: 2160085@my.ipleiria.pt, {marisa.maximiano, catarina.reis}@ipleiria.pt

José Antonio Quiña Mera
Carrera de Ingeniería de Sistemas Computacionales
Universidad Técnica del Norte
Ibarra, Ecuador
email: aquina@utn.edu.ec

***Abstract*— Agile development embodies a distancing from traditional approaches, allowing an iterative development that easily adapts and proposes solutions to changing requirements of the clients. For this reason, the industry has recently adopted the use of its practices and techniques, e.g., Test-Driven Development (TDD), Behavior-Driven Development (BDD), amongst others. These techniques promise to improve the software quality and the productivity of the programmers; therefore, several experiments, especially regarding TDD, have been carried out within academia and in industry. These show variant results (some of them with positive effects and others not so much). The main goal of this work is to verify the impact made by the TDD and BDD techniques in software development by analyzing their main promises regarding quality and productivity. We aim to conduct the experience in academia, with a group of students from the Systems Engineering Degree of the Universidad Técnica del Norte, Ecuador. The students will receive training and appropriate education to improve knowledge about it, and we aspire to achieve interesting results concerning both quality and productivity. The challenge that it is also desirable, is to reproduce the experiment in industry or other adequate contexts.**

Keywords—Empirical research; ITL; TDD; BDD; Software Engineering; productivity; code quality; Incremental Test-Last; Test-Driven Development; Behavior-Driven Development.

I. INTRODUCTION

In software development, quality is probably the most important aspect [1]. The industry in this area is well aware of this because users prefer products that provide a satisfying and productive experience. However, these kind of products are difficult to build. To do this, teams make use of software development methodologies such as: traditional or agile that allow planning and controlling the process of creating a software [2]. Agile methods have been very popular in industry in contrast to traditional methods [1]. They use an iterative approach that responds quickly to the changing needs of the client [2][3]. They improve the quality and also increase the productivity of programmers [4]. A question arises: Do agile practices such as Test-Driven Development (TDD) or Behavior-Driven Development (BDD) help increase product quality and developer productivity? In this context, we intend to run a workshop and a controlled experiment that will answer that question.

The document is structured as follows: Section II introduces software testing and the techniques used in the experiment, Section III provides a summary of the related work, Section IV defines the goals, Section V contains the design of the study. Finally, the expected results and the conclusion and future work are presented in Section VI and VII, respectively.

II. BACKGROUND

Testing is one of the cornerstones of software development because it ensures the quality of the product [3]. In the traditional software development approach, Test-Last Development (TLD) is usually used. Tests are written at the final phase of the development cycle [4]. This means that the quality of the products is only determined in the final phase and, at that moment, making any change can present severe difficulties. On the contrary, in an agile approach that promotes the early development of tests; changes are welcomed and advancing with functional components and correcting defects is made earlier in the process [5].

A. Incremental Test-Last

Incremental software development is modeled around a gradual increase of feature additions to a system. This allows the programmer to take advantage of what was being learned during the development of the earlier ones and provide more user-visible functionality with each addition [6][7].

Incremental Test-Last (ITL) is a natural evolution of the TLD approach and became available upon the introduction of the Revised Waterfall model that enabled the Royce's iterative feedback [8]. Thus, it consists of the development of small portions of the production code, followed immediately by the performance tests of the corresponding unit [9]. The ITL flow is present in Figure 1.

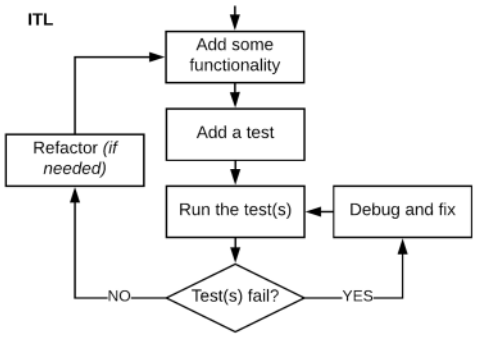

Figure 1. Incremental Test-Last flow.

B. Test-Driven Development

TDD, created by Kent Beck (inventor of Extreme Programming [10] and JUnit [11]) refers to a style of programming where three activities are closely intertwined: Coding, Testing (in the form of unit tests) and Design (in the form of refactoring) [12]. Its main idea is to

 ISBN: 978-1-61208-668-2

perform initial unit tests for the code that must be implemented [13], and then implement the actual feature.

The TDD process [4][5] is presented in Figure 2, and consists of the following steps: (1) select a user story, (2) write a test that fulfills a small task of the user story and that produces a failed test, (3) write the production code necessary to implement the feature, (4) execute the pre-existing tests again. Where if any test fails, the code is corrected and the test set is re-executed and finally (5) the production code and the tests are re-factored. This method produces some benefits that focus on the promise of increasing the quality of the software product and the productivity of programmers [13][14].

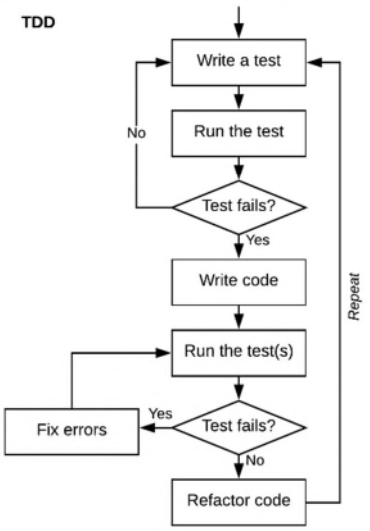

Figure 2. Test-Driven Development flow (*based on* [4]).

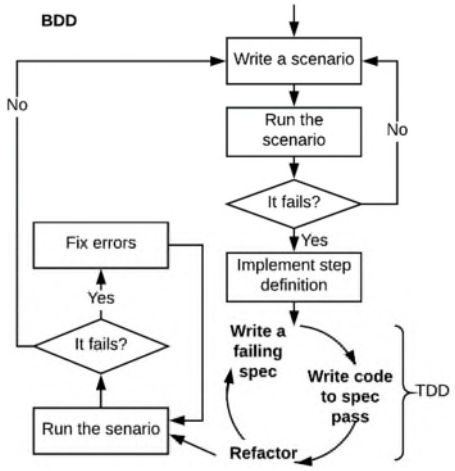

Figure 3. Behavior-Driven Development flow.

C. Behavior-Driven Development

BDD, initially proposed by Dan North [15], is a synthesis and refinement of software engineering practices that help teams generate and deliver higher quality software quickly [16][17]. It has core values that are guided by some agile practices and techniques, including, in particular: Test-Driven Development (TDD) and Domain Driven Design (DDD). Most importantly, BDD provides a common language based on simple structured sentences expressed in something extremely similar to spoken English (Gherkin) [18]. This aspect facilitates communication between project team members and business stakeholders [16]. Gherkin is used to write the acceptance tests as examples and descriptions of scenarios that anyone on the team can read [18].

The BDD process is similar to TDD (see Figure 3) and follows these steps: (1) write a scenario, (2) run the scenario that fails, (3) write the test that corresponds to the specifications of the scenario, (4) write the simplest code to pass the test and the scenario, and lastly, (5) refactor to eliminate duplication.

III. Related Work

Test-Driven Development has been exposed to several scientific experiments developed by researchers in order to validate the advantages offered by its use in software development. O. Dieste et al. [4] studied the produced effect by the technique on the developer's experience through analysis of external quality and productivity. By imparting theoretical and practical knowledge of ITL and TDD to a group of master's students and evaluating the application of techniques in the execution of programming exercises, the study showed that the effectiveness of TDD is lower than ITL. Although the differences are not significant, both productivity and quality improved in half of the cases. They deduced that the technique does not produce immediate benefits and that an intensive training for the subjects is of the utmost importance.

The research directed by Munir et al. [1] was developed in the industry with professional Java developers with previous knowledge of software testing. It aimed to visualize the impact produced by TDD on the quality of internal code, the quality of external code, and productivity, when compared to TLD (Test-Last Development). For this purpose, a programming exercise consisting of 7 user stories was executed. This allowed the participants to put into practice the aforementioned techniques. The results of the analysis by the number of approved test cases: McCabe's Cyclomatic complexity, branch coverage, the number of lines of code per person per hour, and the number of user stories implemented per person per hour. The tests showed slightly significant improvements in favor of TDD, especially in reducing the number of defects. In terms of productivity, the tests suggest that subjects who used TDD achieved an average productivity slightly lower than TLD. This indicates that the adoption of TDD requires compliance with the guidelines of all aspects of software development and adequate training to improve the skill set of the tests.

There is also a recent study designed by Fucci et al. [19], where TDD was compared to ITL through a controlled experiment with professionals within software companies (two in Europe and one in Asia). To achieve a more exact qualification of the effect produced by the techniques within quality and productivity four characteristics were formulated: sequencing, granularity, uniformity, and refactoring effort. The resolution of programming exercises of different levels of difficulty revealed that the improvements found in TDD were associated with granularity and uniformity. The remaining characteristics did not have a relevant influence on the experiment. Thus, the benefits of TDD are due to encouraging stable and precise steps that improve the

 ISBN: 978-1-61208-668-2

focus and flow of software development which in turn promise to improve quality and productivity.

Regarding Behavior-Driven Development (BDD), no experiments have been found that evaluate the benefits proposed by the technique, but being based on a set of practices including Test-Driven Development, we hope that it improves or, at least, maintains benefits granted by TDD. In addition, some investigations were found [20][21][22] in which BDD is put into practice in the development of computer solutions while obtaining good results.

IV. STUDY DESIGN

The goal of this empirical experiment is to analyze the impact on software quality and developer productivity produced by applying test-based techniques in software development. The project goal will be achieved through four related steps:

- Step 1: Teach a group of computer systems engineering students about Software Testing, JUnit, Incremental Test-Last, Test-Driven Development, and Behavior-Driven Development.
- Step 2: Provide a workshop about software development and testing techniques with the execution of ***code katas*** (programming exercise).
- Step 3: Provide a challenge to the students so that they can apply the techniques in an autonomous way.
- Step 4: Analyze and evaluate the results obtained by the challenge in order to show the incidence in the quality of the developed software and the productivity with the mentioned techniques.

A. Research questions

The experiment is focused on the following research questions with regard to three outcomes: external software quality (fulfillment of stakeholder requirements), internal software quality (the way that the system has been constructed) and developer productivity. External quality is based on functional correctness, and specifically, average percentage correctness [4][19]. Internal code quality deals with the code quality in-terms of code complexity, branch coverage, coupling and cohesion between objects [1]. Developer productivity is based on speed of production, or amount of functionality delivered per effort unit [4][19].

- *RQ1: Does TDD and BDD improve external code quality compared to ITL?*
- *RQ2: Does TDD and BDD improve internal code quality compared to ITL?*
- *RQ3: Does TDD and BDD improve productivity compared to ITL?*
- *RQ4: Does BDD improve external code quality compared to TDD?*
- *RQ5: Does BDD improve internal code quality compared to TDD?*
- *RQ6: Does BDD improve productivity compared to TDD?*

B. Experimental description

The experiment will be done with students (approximately 20) from the Systems Engineering Degree of the Universidad Técnica del Norte (Ibarra - Ecuador). It will have an approximate duration of 30 hours and will consist of three phases: knowledge, training, and experimentation.

In the initial phase (**knowledge phase**), information will be given such as: Introduction to agile development, Testing, JUnit, Incremental Test-Last, Test-Driven Development, Behavior-Driven Development, and Cucumber [18]. In addition, at the end of the explanation of each of the techniques, a ***simple calculator*** will be created that allows the calculations of adding and dividing. The application will not have graphical interface so that the students can focus on the understanding of the execution of the technique.

In the **training phase**, jointly done with the students, two code katas will be developed: Rock Paper Scissors (RPS) and Roman Numerals (RM). They will be developed using the techniques chosen for the experiment. RPS is a traditional game involving two players making pre-defined hand gestures while playing against each other, with the winner being decided based on the rules [23]. RM is about the conversion of Arabic numbers into their Roman numeral equivalents, and vice versa [24].

In the third phase (**experimental phase**), students will develop two code katas through the techniques learned: FizzBuzz variant (FB) and String Calculator (SC). FB is a counting and number replacement game, where: any number that is divisible by 3 is replaced with the word 'fizz', any number divisible by 5 is replaced with the word 'buzz', any prime number is replaced with the word 'whiz', any number simultaneously divisible by 3 and 5 is replaced with 'fizz buzz', any prime number divisible by 3 is replaced with 'fizz whiz', and any prime number divisible by 5 is replaced with 'buzz whiz' [21]. SC is about building a *string calculator* with a simple add method [26]. It receives a string with some numbers separated by one or multiple delimiters and returns the sum of all the numbers. An estimation of four hours duration was made to ensure the total resolution of each exercise.

It is important to emphasize that each code *kata* was evaluated with the ***function point metric*** that provides a measure to the difficulty of the exercise. The purpose is to solve exercises of similar difficulty both in the training phase as well as in the practice phase. This metric allows the evaluation of the functionality of a software at any stage of its life cycle [27][28].

C. Design and threats

The order of the interventions used in an experiment can affect the behavior of the subjects or elicit a false response due to fatigue, carry-over, resolution order, or outside factors [4][29]. To counteract this, a counterbalanced design could be applied (see Figure 4), which reduces the impact of the order of interventions or other factors adversely influencing the results [29][30]. This process is called “Latin Square”.

Our experiment will have three interventions (ITL - A, TDD - B and BDD – C). We will divide the subjects into 6 groups and choose the interventions’ order according to the following: ABC, ACB, BAC, BCA, CAB and CBA.

 ISBN: 978-1-61208-668-2

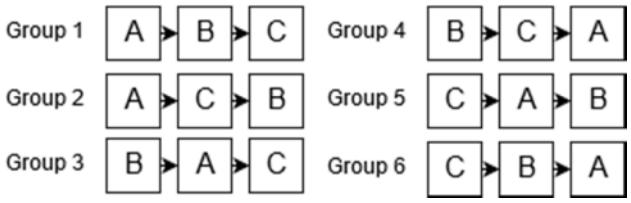

Figure 4. Counterbalanced design for three conditions.

The experimental sessions will be applied contiguously in time, so the main obstacle is fatigue. To counteract this threat, we will provide an adequate time period for the execution of each exercise and grant breaks within the resolution of each technique.

D. Factors and metrics

The experiment will be based upon two factors. The development approach level [4]: ITL, TDD, or BDD, will be used as the main factor. The tasks [4] corresponding to the development of code katas (FB and SC) will be used as the secondary factor. The effectiveness of the development approach will be studied under the perspective of the experiments [1][4][19].

The external quality metric (QLTY) represents the degree of agreement of the system with the functional requirements [4][19]. The formula for calculating QLTY is defined as:

$$QLTY = \frac{\sum_{=1}^{\#TUS} QLTY_i}{\#TUS} \; x \; 100 \quad (1)$$

where $QLTY_i$ is the quality of the user history *i*th implemented by the subject. $QLTY_i$ is defined as:

$$QLTY_i = \frac{\#ASSERT_i(PASS)}{\#ASSERT_i(ALL)} \quad (2)$$

In turn, the number of user stories addressed (#TUS) is defined, such as:

$$\#TUS = \sum_{i=0}^{n} \begin{cases} 1 & \#ASSERT_i(PASS) > 0 \\ 0 & otherwise \end{cases} \quad (3)$$

where *n* is the number of user stories that make up the task. In both cases, it represents the number of passing JUnit assert statements in the set of tests associated with the *i*th user history. Consequently, a user history is considered addressed if it passes at least one of its JUnit assert statements. For example, supposing that a person assesses two user stories (#TUS = 2), this means that there are two user stories for which at least one assert statement passes in the test suite. Let us assume that the acceptance tests of the first analyzed user story contains twelve assertions, out of which six are passing. The acceptance tests of the second user story contain nine assertions, of which three are passing. The quality value of the first assessed user story (QLTY1) is 0.50, while the second user story has a quality value of 0.33 (QLTY2). Therefore, the QLTY measure for the subject is 41.5 percent, i.e., (QLTY = (0.50 + 0.33) / 2 * 100).

The productivity metric (PROD) represents the work done by the subjects with the required quality and within the specified time [19]. Its formula is defined as:

$$PROD = \frac{OUTPUT}{TIME} \quad (4)$$

OUTPUT symbolizes the percentage of passing JUnit assert statements found in the set of tests for a task.

$$OUTPUT = \frac{\#ASSERT(PASS)}{\#ASSERT(ALL)} \; x \; 100 \quad (5)$$

TIME (minutes) is an estimate of the amount of work used in the resolution of a task and is based on the time records (milliseconds) collected by the IDE.

$$TIME = \frac{t_{close} - t_{open}}{6000} \quad (6)$$

For example, a person implements a task with a total of 50 assert statements in a test suite. After running the acceptance test suite against the person's solution, 40 assert statements are passing. Then OUTPUT = (40 / 50) x 100 = 80%. Suppose that the solution was delivered in one and a half hours (i.e., TIME = 90 minutes). The person's PROD is therefore 0.89 (80/90), denoting an assertion passing rate of 0.89 percent per minute.

Regarding the *internal quality* analysis, the metric used in the experiment by Munir et al. [1], McCabe's cyclomatic complexity metric, provides a quantitative measurement of the logical complexity of a software; that is, it indicates how a program can be difficult to test and maintain [1][31]. Furthermore, the Source Code Analyzer PMD will be applied to find common programming flaws like: unused variables, empty catch blocks, unnecessary object creation, and so forth [32].

E. Development Environment Operationalization

The development environment that the participants will use includes: Java 8 using the IDE: IntelliJ IDEA with the 4 additional plugins of Cucumber, Activity Tracker, Metrics Reloaded, and QAPlug. The Cucumber plugin will allow the implementation of the BDD technique in the resolution of the exercises. The Activity Tracker plugin is intended to track and record the activity of the IDE user, such as the time spent on tasks. McCabe's cyclomatic complexity metric will be applied with the use of Metrics Reloaded plugin. In addition, QAPlug plugin implements PMD module to manage code quality.

V. Expected Results

We expect that the descriptive statistics analysis of the information compiled from the code katas implementation by ITL, TDD and BDD responds positively to questions RQ1, RQ2, RQ4 and RQ5. Meaning that the exercises developed through TDD and BDD should present improvement of internal and external quality. A slight decrease of the productivity is expected due to the fact that both TDD and BDD present more steps in its process (RQ3 and RQ6).

VI. Conclusion

The experiments that analyze TDD against other techniques mention that the benefits are not very evident and emphasize training as one of the relevant facts for obtaining such results. Therefore, this work focuses on increasing training and performing exercises at the same level of difficulty with the intention of maximizing understanding of the implementation of the techniques used and obtaining better results. This work is now complete after the initial application of the study, held between May and June 2018. We are currently gathering all the information and conducting the statistical analysis

 ISBN: 978-1-61208-668-2

that will be of great benefit if the research applied in other environments such as in industry and other countries.

ACKNOWLEDGMENTS

This work is financed by national funds through the FCT - Foundation for Science and Technology, I.P., under project UID / CEC / 04524/2016.

REFERENCES

[1] H. Munir, K. Wnuk, K. Petersen, and M. Moayyed, "An experimental evaluation of test driven development vs. test-last development with industry professionals," *Proc. 18th Int. Conf. Eval. Assess. Softw. Eng. - EASE '14*, pp. 1–10, 2014.

[2] J. Shore and S. Warden, *The Art of Agile Development*. O'Reilly Media, Inc, 2008.

[3] T. D. Hellmann, A. Sharma, J. Ferreira, and F. Maurer, "Agile testing: Past, present, and future - Charting a systematic map of testing in agile software development," *Proc. - 2012 Agil. Conf. Agil. 2012*, pp. 55–63, 2012.

[4] O. Dieste, E. R. Fonseca, G. Raura, and P. Rodríguez, "Efectividad del Test-Driven Development: Un Experimento Replicado," *Rev. Latinoam. Ing. Softw.*, vol. 3, no. 3, p. 141, 2015.

[5] D. Janzen and H. Saiedian, "Test-driven development: Concepts, taxonomy, and future direction," *Computer (Long. Beach. Calif).*, vol. 38, no. 9, pp. 43–50, 2005.

[6] C. Larman and V. R. Basili, "Iterative and incremental developments. a brief history," *Computer (Long. Beach. Calif).*, vol. 36, no. 6, pp. 47–56, 2003.

[7] I. Sommerville, *Software Engineering*. Pearson, 2016.

[8] R. Martin, "Iterative and incremental development (iid)," *C++ Rep.*, vol. 11, no. 2, pp. 26–29, 1999.

[9] D. S. Janzen, "On the Influence of Test-Driven Development on Software Design," pp. 0–7, 2006.

[10] K. Beck and M. Fowler, *Planning Extreme Programming*. Addison-Wesley, 2000.

[11] K. Beck, *JUnit pocket guide*. O'Reilly Media, 2004.

[12] Agile Alliance, "What is Test Driven Development (TDD)?" [Online]. Available: https://www.agilealliance.org/glossary/tdd/. [Accessed: 11-Nov-2017].

[13] R. Martinez, "TDD, una metodología para gobernarlos a todos," 2017. [Online]. Available: https://www.paradigmadigital.com/techbiz/tdd-una-metodologia-gobernarlos-todos/. [Accessed: 11-Nov-2017].

[14] K. Beck, *Test-driven development: by example*. Addison-Wesley, 2003.

[15] D. North, "Introducing BDD | Dan North & Associates," 2006. [Online]. Available: https://dannorth.net/introducing-bdd/. [Accessed: 07-Dec-2017].

[16] J. F. Smart, *BDD In Action: Behavior Driven Development for the Whole Software Lifecycle*. Manning, 2014.

[17] Agile Alliance, "BDD: Learn about Behavior Driven Development." [Online]. Available: https://www.agilealliance.org/glossary/bdd/. [Accessed: 16-Nov-2017].

[18] M. Wynne and A. Hellesoy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*, Pragmatic. 2012.

[19] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?," *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 597–614, 2017.

[20] M. Rahman and J. Gao, "A reusable automated acceptance testing architecture for microservices in behavior-driven development," *Proc. - 9th IEEE Int. Symp. Serv. Syst. Eng. IEEE SOSE 2015*, vol. 30, pp. 321–325, 2015.

[21] R. A. De Carvalho, F. Luiz, D. Carvalho, R. S. Manhães, and G. L. De Oliveira, "Implementing Behavior Driven Development in an Open Source ERP," pp. 242–249, 2013.

[22] P. L. De Souza, A. F. Do Prado, W. L. De Souza, S. M. Dos Santos Forghieri Pereira, and L. F. Pires, "Combining behaviour-driven development with scrum for software development in the education domain," *ICEIS 2017 - Proc. 19th Int. Conf. Enterp. Inf. Syst.*, vol. 2, no. Iceis, pp. 449–458, 2017.

[23] Agile Katas, "Rock Paper Scissors Kata." [Online]. Available: http://agilekatas.co.uk/katas/RockPaperScissors-Kata. [Accessed: 16-Feb-2018].

[24] Agile Katas, "Roman Numerals Kata." [Online]. Available: http://agilekatas.co.uk/katas/RomanNumerals-Kata. [Accessed: 16-Feb-2018].

[25] M. Whelan, "FizzBuzzWhiz Kata." [Online]. Available: https://github.com/mwhelan/Katas/tree/master/Katas.FizzBuzzWhiz. [Accessed: 16-Feb-2018].

[26] R. Osherove, "TDD Kata 1 - String Calculator." [Online]. Available: http://osherove.com/tdd-kata-1/. [Accessed: 16-Feb-2018].

[27] Ifpug, "Function Point Counting Practices Manual," *Group*, vol. on06/23/. 2010.

[28] F. Sánchez, "Medida del tamaño funcional de aplicaciones software," *Univ. Castilla-La Mancha*, 1999.

[29] N. J. Salkind, *Encyclopedia of Research Design*. SAGE Publications, 2010.

[30] D. J. Saville and G. R. Wood, *Statistical Methods: The Geometric Approach*. Springer New York, 1991.

[31] T. J. McCabe, "A Complexity Measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.

[32] PMD Open Source Project, "PMD Source Code Analyzer." [Online]. Available: https://pmd.github.io/. [Accessed: 19-Feb-2018].

 ISBN: 978-1-61208-668-2