Using Software Changes to Understand the Test Driven Development Process

Michael Hilton, Nicholas Nelson, Hugh McDonald, Sean McDonald, Ron Metoyer, Danny Dig
Oregon State University, USA
Email: {hiltonm, nelsonni, mcdonase, mcdonalh, metoyer, digd}@eecs.oregonstate.edu

***Abstract*—A bad software development process leads to wasted effort and inferior products. In order to improve a software process, it is important to first understand it. Our unique approach in this paper is to use code and test changes to understand conformance to a process. We analyze the meaning of these changes to obtain a deep, rich understanding about the process. In this paper we use Test Driven Development (TDD) as a case study to validate our approach.**

We designed a visualization to enable developers to better understand their TDD software process. We analyze our visualization by using the Cognitive Dimensions framework to discuss some findings and design adjustments. To enable this visualization, we developed a novel automatic inferencer that identifies the phases that make up the TDD process solely based on code and test changes. We evaluate our TDD inferencer by performing an empirical evaluation on a corpus of 2601 TDD sessions. Our inferencer achieves an accuracy of 87%.

I. Introduction

A bad software development process leads to wasted effort and inferior products. Unless we understand how developers are following a process, we cannot improve it. Previous research has shown that improving development processes has led to defect reductions of 85%, reduction of development cycle time by 50% [1], and productivity improvements of 35% year over year [2]. Unfortunately, many developers miss out on these potential gains.

In this paper we use Test Driven Development (TDD) as a case study on how software changes can illuminate the development process. TDD is a software development process which consists of writing unit tests before developing production code in small, rapid iterations [3], [4]. We chose this process because it is very controversial. While it is very popular in the agile community [5], [6], there are many who do not see the value in it [7], and some even go so far as to claim it is bad [8], [9]. One of the reasons for this controversy is a lack of consensus regarding the benefits of TDD [10]–[13]. Perhaps a better understanding of the process could help solve some of the disagreement around TDD.

The heart of TDD is the order of writing tests before code, and doing so in small iterations. In order to understand and improve how developers follow TDD, they need to first understand whether they follow TDD or whether they deviate from the process (e.g, by writing production code before writing failing tests). Even when they follow the process, developers need to be able to quantify how well or poorly they followed the TDD process. For example, they need to identify outliers in metrics such as the ratio of code/tests in a TDD cycle, or the number of lines of code/tests written in a cycle. Ultimately, it will be almost impossible for them to improve their process without this information.

Our unique approach in this paper is to use code and test changes to understand conformance to a process. We analyze the meaning of change to obtain a deep, rich understanding about the process. We convert syntactic changes into semantic changes that help us identify the phases of the process. For example, additions and removals of test methods and assertions in the test code tell us that the developer is in the test-writing phase of TDD, whereas changes in the production and test code that preserve the current test results tells us that the developer is in the refactoring phase of TDD.

With the information that we get from analyzing these code and test changes, we designed novel visualizations which enable developers to understand how they follow the TDD process. Our design was guided by the nested model of Munzner [14], a well established design process model that breaks design into 4 nested stages and establishes guidelines for evaluation/validation within each stage. Our visualizations show developers both the presence and absence of the TDD process during development. This provides high level TDD process conformance information. Also, our visualizations allow the developer to interactively drill down and gain a deeper understanding of their process. On the next level, the visualization shows how each cycle breaks down into the red, green, and refactor phases of TDD. This enables developers to quickly identify trends and check whether they are following TDD consistently.

Once a developer finds an outlier, she can drill down even further to see the relevant source code and test changes. This contextual information allows developers to better understand the part of their TDD process they are considering.

These different levels of information allow the developer to avoid irrelevant parts of their development history, while at the same time accessing detailed information on the outliers.

In order to enable these visualizations, we developed a TDD inference algorithm, which given a sequence of code edits and test runs it automatically detects code that is written following the TDD process. Moreover, the inferencer also associates specific code changes with specific parts of the TDD process. The inferencer is crucial for giving developers higher-level information that they need to improve their process.

One of the fundamental challenges is that during the TDD practice, not all code is developed according to the textbook definition of TDD. Even experienced TDD developers selectively apply TDD only during some of their code development, and only on some parts of their code. This introduces lots of noise for any automatic inferencer.

A fundamental challenge for any tool that is designed to improve a process is getting fine-grained data points at the intermediate steps, not only at the beginning and end of the process. In our case, mining open-source code repositories that might have been produced using TDD is insufficient because there is not enough intermediate information about the small TDD cycles. Previous research [15] shows that mining code snapshots stored when developers commit their changes is too coarse-grained and incomplete. An obvious solution is to instrument the development environment and record all micro edits and test runs. But this raises privacy and security concerns.

Another challenge is finding participants who are TDD practitioners. While we could bring participants into a lab and ask them to perform TDD programming tasks, this would lead to synthetic data which is not representative of how developers phase in and out of TDD.

To solve both challenges, in this paper we use a corpus of data from cyber-dojo [16], a website that allows developers to practice and improve their TDD by coding solutions to various programming problems. While this corpus is not production data, it is very large, with a total of 41766 fine-grained snapshots from 2601 programming sessions. Code is written from all over the world, which gives us a very rich, diverse corpus. There are over 30 different programming tasks, and there are many sessions for each task. However, this wide variety makes the task of developing an inferencing algorithm difficult, because we cannot target a single skill level or population segment. Thus, we believe this corpus is more representative than a controlled experiment.

In cyber-dojo all sessions are anonymous. Since we do not have access to the individual coders who produced the data in our corpus, we can not use them in an *empirical* evaluation of our visualization. Instead, we performed a formative *analytical* assessment of our design using the Cognitive Dimensions [17] framework. While this framework was designed to analyze visual programming languages, it has since been demonstrated to be applicable for any information artifact such as a visual data representation.

This paper makes the following contributions:

- **Process Understanding:** To the best of our knowledge, we are the first to propose a novel usage of software changes to infer process information. Instead of analyzing metrics taken at various points in time, we analyze deltas (i.e., the changes in code and tests) to understand conformance to a process.
- **TDD Visualization Design and Analysis:** To the best of our knowledge, we present the first visualization designed specifically for understanding the TDD process. Our novel visualizations show the presence or absence of TDD and allow progressive disclosure of TDD activities. We used the Cognitive Dimensions framework to perform an analysis that allows us to discuss some findings and design adjustments.
- **TDD Inference Algorithm:** As a case study of understanding a process, we focus on TDD. To the best of our knowledge, we present the first algorithm to infer the activities in the TDD process solely based on snapshots taken when tests are run.
- **Implementation and Empirical Evaluation:** We implemented and evaluated the accuracy of our inferencer using a corpus of 2601 TDD sessions from cyber-dojo. Our inferencer achieves an accuracy of 87%. This shows that our inferencer is effective.

II. Overview of Test Driven Development

“The act of writing a unit test is more an act of design than of verification. It is also more an act of documentation than of verification. The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function” -Robert Martin [18], leading Agile Development author.

TDD is not a testing technique, it is a software design technique [19]. It is an iterative development process that consists of multiple cycles. A cycle consists of three phases.

- In the *red phase* the developer writes only enough unit test code so that it fails. A compilation error caused by missing production code is also considered a failure.
- In the *green phase* the developer writes enough production code to make the previously written test pass.
- In the (optional) *blue phase* the developer refactors the production code or the test code to clean up, remove duplication, or improve design.

In the ideal TDD environment, the cycles should be relatively small, and the developer transitions from phase to phase often [6], [18].

One of the foundational technologies that is needed in order to perform TDD is a unit test framework. This unit test framework should be able to execute all automated tests quickly and easily [20], [21]. When developing Java code, JUnit [22] is the leading framework. When performing TDD, it is very important to be able to quickly run these tests and know if they are passing or if one or more of them is failing. Knowing that all unit tests are passing is what gives TDD developers the confidence they need to continue. It also allows them to refactor frequently and confidently [6].

However, TDD is more then just a series of tools, it is a craft that must be honed over time. One technique that Agile developers use to develop technique and proficiency over time is code katas [23]. Kata [24] is a Japanese word meaning “form”, and in martial arts it describes a choreographed pattern of movements used to train yourself to the level of muscle memory. Dave Thomas [25] coined this term in the context of software development. He defines a code kata as a short exercise to help developers think about the issues behind programming.

Next we describe a sample kata called *Fizz Buzz*. This kata provides users with the following instructions:

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz". [16]

The user must implement all requirements in order to complete the kata, but the instructions do not provide explicit steps.

A sample TDD process to implement this kata would proceed as follows. In the first TDD cycle, the developer writes a test to assert that all numbers between 1–100 are printed. This test will fail because there is no production code yet. Then he writes the production code to make this test pass, and runs the test again, ensuring that it does pass.

In the second TDD cycle, the developer writes a failing test to assert that multiples of 3 print "Fizz" instead of the number. Then he writes the production code to make the test pass.

In the third TDD cycle, he writes a failing test to assert that multiples of 5 print "Buzz". Then he writes the production code.

In the fourth TDD cycle, he writes a failing test to assert that multiples of both 3 and 5 print "FizzBuzz". Then he writes the production code and runs all tests which should pass now. He realizes that he has duplicated code between the last cycle and the previous two. He refactors the code to remove duplication and reruns all the tests.

III. Visualization

In this section we present a visualization we designed to enable developers to better understand their TDD software process. Since color is the primary encoding that we use for our visualization, our figures are best viewed in color, and may be difficult to understand when printed in black and white.

Our design was guided by the nested model of Munzner [14], a well established design process model that breaks design into 4 nested stages and establishes guidelines for evaluation/validation within each stage. The four stages correspond to Domain Characterization, Data and Operation Abstraction Design, Encoding and Interaction Technique Design, and Algorithm Implementation. We carried out all four stages. For the purposes of this paper, we will discuss the Domain Characterization stage which requires a characterization of the target users of the visualization.

A. Target Audience for Visualization

The first stage of the model requires a characterization of the domain of study. In this case, the domain of interest is TDD and the specific target users are individuals interested in analyzing the TDD process. We focus on two particular users: coders and researchers.

The primary user is a coder who followed the TDD process and who is now interested in reviewing her TDD process to better understand it. We anticipate that this visualization could also be useful for a novice TDD coder to examine the TDD process of a more experienced coder in order to learn how to use TDD well. Researchers could also benefit from a tool that helped them review the TDD process over time to gain a much better understanding of how their subjects perform TDD.

To better understand the target user and the TDD analysis domain, we drew upon our own expert knowledge of the TDD process and observations of real developers. The outputs of this phase are generally presented as a set of questions that the target users may ask of the data at hand.

1) Coders:

- **What does my TDD workflow look like?** The developer may be interested in gaining a basic understanding of her process, which phase of the process she spends the most time in, where she wrote the most code, or what code corresponds to a specific segment of the TDD process.
- **How does my TDD workflow compare to others?** A novice practitioner may be interested in comparing his TDD workflow to that of an experienced developer to identify the differences and areas for improvement.

2) Researchers:

- **What does a real-world TDD workflow look like?** While there is a fairly simple set of rules that define TDD, identifying a rigorous definition can prove tricky [26]. A tool that maps a TDD session to a visual representation can help clarify what an actual TDD implementation looks like.
- **How does practical TDD differ from theoretical TDD?** Once researchers have a better understanding of actual TDD implementations, they can then begin to explore the differences between actual and theoretical TDD.

From these four primary questions, we defined a set of abstraction operations that the users would have to perform to answer them: *characterize* a TDD cycle, *compare* TDD cycles, *retrieve* values such as numbers of lines of code or time spent in a cycle, and *retrieve* code that corresponds to a particular cycle. We chose visual encodings and interactions for the representation based on these operations and data, which we then prototyped.

B. Visualization Elements

1) TDD Cycle Plot: We represent a single TDD cycle as shown in Figure 1. This representation was inspired by hive plots [27] and encodes the nominal cycle data with a positional and color encoding (red=test, green=code, blue=refactor). The position of the segment redundantly encodes the TDD cycle phase (e.g. the red phase is always top right, the green phase is always at the bottom, and the blue phase is always top left). The number of lines of code written in a phase or time spent in a phase are both quantitative values encoded in the length [28], [29] of the cycle plot segment. Taken together, a single cycle plot takes on a specific 'shape' based on the characteristics of the phases, effectively using a 'shape' encoding for different types of TDD cycles. We illustrate the shape patterns of various TDD cycles in the next section.

2) TDD Heartbeat: The primary way for users to understand their TDD process is the TDD Heartbeat view which consists of a series of TDD cycle plots, one for every cycle of that session (See Figure 1) We call this the TDD heartbeat because this view gives an overall picture of the *health* of the TDD process because it is designed to show the process as it evolves over time. This particular view is useful in answering all of the questions from above. By being able to see both information about individual cycles, as well as seeing how the cycles compare over time, the coder can gain a better understanding of their process. In order to better illustrate this, we present several katas from our data set, and some observations about how they represent the TDD Process.

Figure 1 shows a TDD process that fits the classic definition of TDD. It starts with a large cycle due to boilerplate code that needs to be written in order for the code to compile. This cycle is followed by a series of quick small cycles to complete the kata.

Figure 2 shows a kata that was coded still following TDD, but instead of a series of small rapid cycles, the developer used fewer cycles, but wrote more code in each cycle. While this is not as ideal as the process shown in Figure 1, it still is following TDD.

Figure 3 shows a kata that does not represent correct TDD. The third cycle only consists of a red phase, with no green phase, so this was not consistent TDD. Even for the cycles that do contain all the necessary phases, it is clear by the uneven sizes that there was not a consistent development process used to develop this kata.

Figure 4 shows the process of a TDD coder who spends a lot of time refactoring. By comparing this process against the more balanced approach in Figure 1, a coder might determine that she is refactoring much more than is typical in classic TDD. Our algorithm does not present this as being good or bad, it simply gives the coder the tools to begin to quantify and characterize her development process.

3) Snapshot Timeline: The snapshot timeline provides more information about the TDD process, specifically showing all the snapshots that are in the current session. An example snapshot timeline is shown in Figure 5. Each snapshot is represented with a rounded square. The color represents the outcome of the tests at that snapshot event. Red signifies the tests were run, but at least one test failed. If all the tests passed, then it is colored green. If the code and tests do not compile, then we represent this with an empty gray rounded box. We encode time or order positionally, left to right, a typical and effective time encoding. Above the snapshots are boxes that show which snapshots belong to which cycles and phases. The rounded gray boxes show the cycle boundaries, and inside each cycle the ribbon of red, green and blue signifies which snapshot events fall into which phases.

This view answers questions specifically dealing with how consistent the TDD process was followed, and what snapshots were written by coders using the TDD process, and which ones were not. Figure 6 shows an example where the TDD process was not followed consistently. The coder started with performing two cycles which do conform to TDD, but then she wrote some code that deviated from the TDD process. However, they managed to write another cycle before deviating from TDD for the rest of the session. This can be contrasted with Figure 5 which shows a consistent conformance to the TDD process.

This view also allows the user to interactively select snapshots that are used to populate the code edit area (described below). To select a series of snapshots, the user interactively drags and resizes the gray selection box.

This encoding supports the same questions as the cycle plot and heartbeat arrangement, however, it does so at a finer granularity, showing each test run individually.

4) TDD Code Edits: In order to allow the coder to understand the TDD process even better, we display the changes to the code between two snapshots in the code edits viewer. In order to understand the TDD process, a coder must be able to look at the code that was written and see how it evolved over time. By positioning the selection box on the timeline as described above, a user can view how all the code evolved over any arbitrary amount of time. The view consists of expandable and collapsable boxes for each file. Each box contains two code editors, one for the beginning of the selected region, and one for the end. The code is displayed in a visual diff tool [30] in order to succinctly show the differences.

Whenever the user selects a new selection area, these boxes dynamically repopulate their content with the correct diffs.

IV. Cognitive Dimension Analysis of Visualization

An important aspect of the nested model for visualization design is that each phase of the model requires some form of validation. The visual encodings and interaction techniques are often validated through a usability study after the artifact is completely built, however, the Cognitive Dimensions (CDs) Analysis of notations method can be used during design.

The Cognitive Dimensions analysis [31] is a broad brush usability analysis technique that was originally introduced for analyzing visual programming languages but have been more widely used to analyze the usability of *information artifacts* in general. The original framework consisted of 14 dimensions that represent design principles along which a design can be analyzed. Here, we discuss some findings and design adjustments from analysis of the most relevant dimensions for our design.

A. Consistency

In a consistent notation, the functionality of a visual element can be inferred based on what is known about the functionality of other elements in the interface. This dimension is particularly important in visualization design because the primary activity in the Encoding and Technique Design phase of the nested model is to choose appropriate visual encodings and interactions and these should be used consistently throughout the design. A visual encoding is a mapping of data to a visual property of a mark, where a mark is the visual element (e.g. circle, bar, dot, line, etc.). In our design, color is used to encode

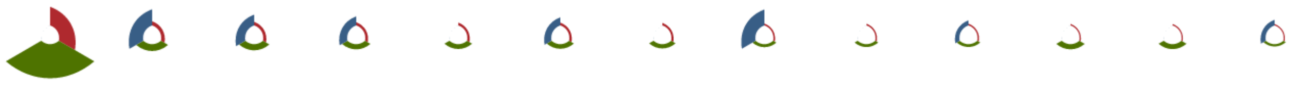
Fig. 1: Classical TDD Heartbeat. For each cycle plot, red is the top right third, green is the bottom third, and blue is the top left third.

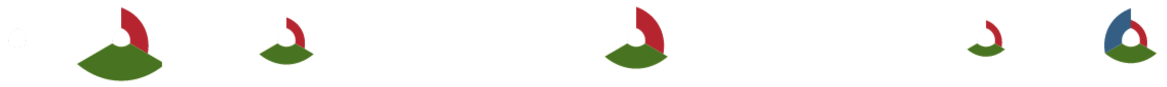
Fig. 2: TDD Heartbeat with less granularity

Fig. 3: Inconsistent TDD Heartbeat

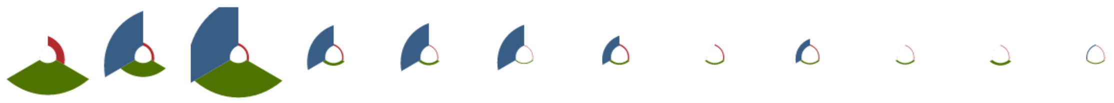
Fig. 4: TDD Heartbeat With Excesive Refactoring

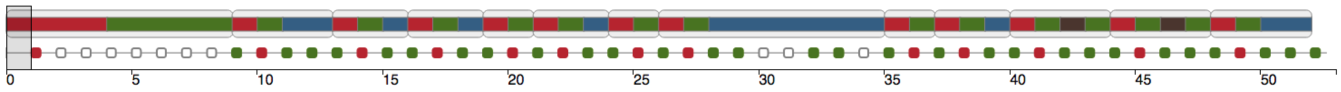

Fig. 5: Consistent TDD snapshot timeline. Since color is one of the primary encodings that we use, these figures will be difficult to read in greyscale.

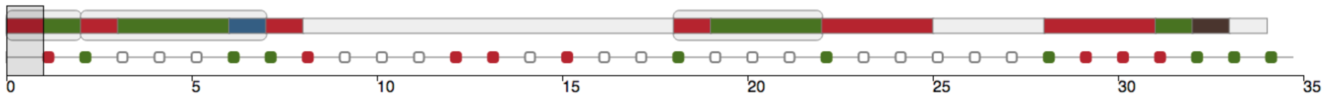

Fig. 6: Inconsistent TDD snapshot timeline

the *phase* of a TDD cycle (red = test, green = coding, blue = refactoring). These colors are used consistently in the cycle plots as well as in the snapshot timeline.

In an early iteration, however, we represented a snapshot that fails to compile as a yellow filled rounded square. Since yellow does not have a natural mapping in the TDD domain and it had not been used in any other part of the visualization, we replaced it with an unfilled circle to a) avoid confusion with existing colors and b) represent an incomplete (and thus not filled) test value.

There is still a minor inconstancy in that the snapshot timeline does not contains snapshots that are blue, to correspond to the refactoring phase. This is left for future work.

B. Juxtaposition and Hard Mental Operations

Juxtaposition refers to the ability of the user to compare different parts of the notation side by side simultaneously. We apply heavy use of juxtaposition in order to allow the user to simultaneously compare cycles from a session and to do so at multiple levels of granularity. However, during our CDs analysis, we realized that the cycle plots in the Heartbeat View were not aligned with the snapshot timeline plots and thus required a *Hard Mental Operation* in order to count and align the cycle plots to the timeline cycles. Hard Mental Operations is yet another dimension in which we investigate the required mental operations of the notation itself (not semantics) in an effort to identify and remove any difficult operations that are an artifact of the design. This alignment problem resulted in a modification to the design in order to align the Heartbeat cycle plots with the corresponding snapshots in the timeline.

C. Closeness of Mapping

This dimension refers to how closely the notation corresponds to the real world problem. In this case, there is a very tight correspondence. The software engineering community uses the colors red, green and blue to denote the TDD cycle phases and thus the choice of color encodings for the phases of a cycle was obvious here.

An early design iteration considered the use of simple bar charts to represent the cycle phases (one bar for each phases to indicate lines of code or time), however, in order to better communicate the circular aspect of a TDD cycle, we employed a hive-plot inspired design that implies a circular process.

D. Secondary Notation

This dimension refers to the ability of users to add extra marks in order to 'annotate' the notation. Our CDs analysis revealed that while we do not support any secondary notation, there are many opportunities to do so that would improve the usability of the tool. For example, users should be able to annotate any section of a cycle with text to describe an interesting finding. In addition, users should be able to 'label' interesting patterns or shapes. This is left as future work.

V. TDD Inference

In order to be able to build the visualizations we have presented thus far, we needed to build a TDD inference algorithm which uses a rich understanding of test and code changes, to infer the TDD process. Instead of relying on static analysis tools, we present a novel approach where we look at the changes to the code. This approach consists of an algorithm designed to take as input a series of snapshots. The algorithm then looks at the code changes between each snapshot and uses that information to determine if the code was developed using TDD. If the algorithm infers the TDD process, then it determines which parts of the TDD process those changes belong to.

A. Snapshots

We designed our algorithm to receive a series of snapshots as input. We define a *snapshot* as a copy of the code and tests at a given point in time.

In addition to the contents of code and tests, the snapshot contains the results of running the tests at that point in time. Our algorithm then uses these snapshots to determine what was changed. It then uses these changes to infer the TDD process. In this paper, we are using a corpus of data where a snapshot was taken every time the code was compiled and the tests were run. It is important that the snapshots have this level of detail, because if they do not, we do not get a clear picture of the development process. We cannot use Version Control System (VCS) commits because previous work [15] has shown that VCS commits are incomplete and imprecise when trying to study code changes.

B. Abstract Syntax Tree

Since our inference algorithm must operate on the data that the snapshots contain, it is important to have a deeper understanding of code then just the textual contents. Our inference algorithm gains this deeper understanding by constructing the Abstract Syntax Tree (AST) for each code and test snapshot in our data. This allows our inferencer to determine which edits belong to the production code and which edits belong the test code. It also calculates the number of methods and assert statements at each snapshot. This richer understanding of the code is necessary to be able to infer all the phases of TDD. We use the Gumtree library [32] to create the ASTs.

C. TDD Inference Definitions

We now provide specific definitions of each phase that the inferencer is looking for.

The following is a list of all the possible phases our inferencer identifies.

1) Red:

We define the red phase as follows:

- A red phase must not contain functional changes to the production code of the program. Only changes to the test code may occur.
- At the completion of the red phase, there must be at least one failing test, or a compilation error. (e.g., when a test calls a method that does not exist, resulting in a compilation error)

2) Green:

We define the green phase as follows:

- All code that is written during the green phase should be written with the purpose of making the failing test from the preceding red phase pass. While this should lead to all changes being exclusive to the production code, we do allow for minor changes to the test code as long as no new tests are being added. (e.g, adding an import statement in order to compile.)
- The end of the green phase is when no compilation errors exist and all the tests pass.

3) Blue (Refactor):

We define the blue phase as follows:

- Code written during the blue phase may involve edits to both the test and production code.
- Blue phases should start with and end with all tests passing.
- Blue phases must be preceded by a green phase.

4) Brown:

During the development of our inferencer, we found a specific case that doesn't fit into either red or green phases as defined above, but we believe still constitutes valid TDD. This is the case when a coder writes a new test during what would be the red phase, but either knowingly or unknowingly the production code will already make this test pass. We decided that we should consider this as a separate phase, and we call it the brown phase.

We define the brown phase as follows:

- The brown phase begins with all the tests passing.
- During the brown phase, the coder must only make edits to the test code, and those edits must include the addition of a new test.
- Brown phases should end with all tests passing.

5) White:

We define the white phase as follows:

- All remaining edits that do not conform to any of the TDD phases as described above.

D. Algorithm

Our inferencer algorithm first examines a selection of one snapshot as a potential phase. If the algorithm determines that the current selection constitutes a phase, it will mark it as such and advances to the next snapshot. If the selection does not contain an entire phase, then the algorithm expands the selection to include the next snapshot and examines it again.

When the algorithm encounters a red-green or red-green-blue sequence of phases, it will mark them as a valid TDD cycle. If the algorithm determines that a potential cycle ends without the correct phases, it marks the potential cycle as an invalid TDD cycle. Our algorithm consists of a large, complex decision tree. Instead of trying to represent it here in text, we believe the reader would benefit more from walking through the following example.

Figure 7 shows the algorithm in action on a real instance of the "Game of Life" kata. Each rectangle consists of one snapshot. We have labeled each rectangle to indicate what type of edits took place in each snapshot. T indicates only test changes, while T+ indicates changes to the test code including the addition of a new test. P indicates production code edits.

We use the rows to simulate an animation as we describe how the algorithm walks over the phases.

The first snapshot that the algorithm encounters contains only edits to test code, which includes a new test being added. Since this phase meets all the requirements for a red phase as described above, the algorithm categorizes the first selection as a complete phase and marks that complete phase as a red phase. It then proceeds to create a new selection.

In the second row of Figure 7, we see that the second selection does not contain all the necessary requirements for a green phase, specifically that the tests that are failing as a part of the red phase do not pass. However, it also doesn't contain any changes that would preclude the current selection from being part of a larger selection that would be compliant with the requirements for green.

The algorithm continues expanding the selection to include snapshots while ensuring that the coder is not adding new tests. It takes two expansions before the selection contains all the necessary elements to categorize the frame as a green phase, as is shown in the third row.

In the fourth row, when evaluating the next phase, the inferencer is either looking for a blue (refactor) phase or a new red phase. In this case the selection contains edits to both the production and test code, and continues to pass all the tests, so the algorithm identifies it as a blue phase.

Once the algorithm has found a valid red phase, green phase, and blue phase in the correct order, it is able to identify those three phases as a cycle. In the fifth row, The algorithm marks this initial cycle as a valid TDD cycle and then continues.

The next selection is a complete red phase, so the inferencer marks it as such.

In the sixth row, after the red phase, the inferencer encounters a series of edits that do not meet the requirements for a complete green phase. There are tests added, changes to both the production code and the test code, and the production code does not make the tests pass. Since the coder did not follow the valid TDD process, the inferencer marks this is as a white phase. While the algorithm did find a correct red phase, the lack of a green phase prevents it from identifying those edits as valid TDD. In the seventh row, the next selection does contain the correct behavior, because the coder added a new test which fails, so the algorithm identifies a new red phase.

The coder then makes changes to the production code, which the algorithm categorizes as a correct green phase. Since the blue (refactor) phase is optional, the algorithm marks a valid red and green phase together as a valid TDD cycle. The last row of the example shows this.

VI. Empirical Evaluation of the Inferencer

We perform an empirical evaluation in order to measure the accuracy of our inferencer. This is crucial because the inferencer is the foundational component for the visualizations.

We begin by describing the origin of the corpus we are using for our evaluation. We then describe the corpus, and finally, present the results of our evaluation.

A. Corpus Origin

In this paper we use a corpus of katas that comes from cyber-dojo, a site that allows developers to practice and improve TDD by coding solutions to various katas. While this it is not industrial production data, it is a large very and diverse corpus. It is very important for us to use non-contrived data that was collected in the wild. We believe that it is very important to use real data, as opposed to synthetic data that was generated with the express purpose of being classified as TDD. While the coding dojo concept is tightly tied to agile development, and specifically TDD, there is nothing about the cyber-dojo site that enforces TDD. Any code and any tests can be written in any order, so our inferencer can make no assumptions about the existence of the TDD process in the data.

When developers arrive at the cyber-dojo site, they first choose a language and a kata. cyber-dojo offers 33 different language and test framework combinations (e.g., Java/JUnit, Java/Cucumber, Python/unittest). It provides 34 different katas that users can choose to practice. They are small algorithms for computing scoring of games (e.g., bowling), text manipulation (e.g., word wrapping), and various mathematical problems (e.g., leap year calculator). Each kata in cyber-dojo can be implemented in any of the 33 language/testing combinations.

Once a developer selects the kata and language, cyber-dojo assigns an anonymous identifier for that session.

Now we describe the process of doing a kata in cyber-dojo. Each session begins with a failing test. The developer can edit the production code or tests, or can run the tests. Whenever the developer runs the tests, cyber-dojo presents three possible outcomes: test failure, test pass, or compilation error. At the same time, cyber-dojo automatically commits the current source code to a local git repository (this happens silently in the background, without any user interaction). Due to this, developers are able to step back and forth in time and review their progress through the kata. This also enables us to review how developers perform TDD at fine-grained intervals: not only can we see the final result, but also we can study how the kata evolved over time.

cyber-dojo collects all the data anonymously. Thus, cyber-dojo is a "safe-to-fail" environment [33], where developers focus on improving, and need not be concerned about the judgment of others.

B. Demographics

cyber-dojo is a publicly available website. Since all the sessions are anonymous, we do not know precisely the demographics of developers that have performed katas in our corpus. As we researched the demographics, we learned that one regular group of users of cyber-dojo is Code Craftsman Saturdays [34]. They are an organization in Michigan which hosts 1-day events where programmers come together and do kata-based exercises on cyber-dojo. We sent a survey to everyone who had participated in a Code Craftsman Saturday during the last year. Among the 92 people who received the survey, 26 responded in the two days the survey was active.

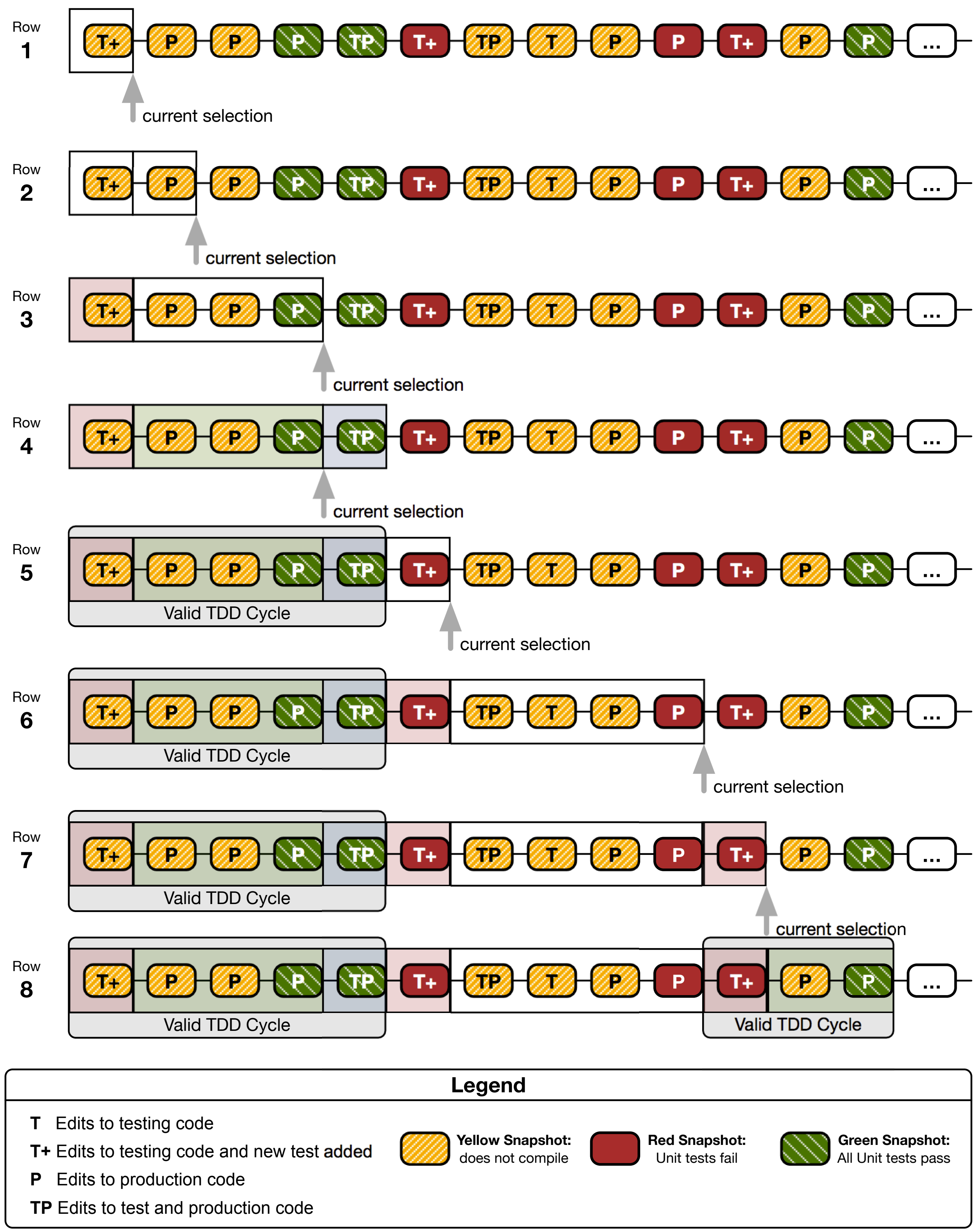

Fig. 7: Example of TDD inference algorithm in action.
Each row represents an interesting point as the algorithm advances.

Table I presents the results of the survey. Over half of them have over 5 years of experience as a professional software developer. Also, 76% of them use TDD at work. While we cannot claim that this is representative of the entire population, at least one group of users represents a very competent audience.

TABLE I: Code Craftsman Survey

Q1: How many years of professional experience as Software Developer?

less then 1 year	1–2 years	2–5 years	5–10 years	>10 years
4%	16%	24%	12%	44 %

Q2: How would you describe yourself?

student	programmer	analyst	designer	other
11%	61%	15%	19%	19%

Q3: Do you practice TDD at work?

Frequently	sometimes	never
34%	42%	23%

C. Evaluation Corpus

To build our corpus we used all the Java/JUnit sessions as as our algorithm only parses Java at the moment.. This gives us a corpus of 2601 total Java/Junit sessions.

We are using this corpus to evaluate our inferencer as all the sessions were attempted by people who had no knowledge of our work.

1) Corpus Preperation: We developed a ruby on rails application that allowed us to be able to work with this corpus in an efficient manner. The raw data that we used to build the corpus consisted of two parts. Each session has a git repository that contains commits of the code each time the coder pressed the "Test" button. This provides a fine-grained series of snapshots that allow us to evaluate the process used to develop the code. The session also contains meta-data files that track things such as when the session occurred, and what was the result of each compile and test run. cyber-dojo uses a traffic light abstraction to represent the results of a compile/run event to the user, where red means the code compiled but did not pass all the tests, green signifies that the code compiled, ran and passed all the tests, and yellow lets the coder know that there was a compilation error.

2) The Gold Standard: In order to evaluate our inferencer, we created a Gold Standard by manually labeling snapshots as to which TDD phase they belonged to, if any. We then grade our inferencer by comparing it's results against the Gold Standard.

In order to not bias the selection process we randomly selected the sessions for our Gold Standard. To ensure that we were labeling consistently, we first verified that we had reached an inter-rater agreement of at least 85% between both of the authors that labeled the corpus. Once we were convinced that we had agreed a consensus among the raters, we divided the rest of the Gold Standard sessions up and rated them individually. We labeled a total of 160 sessions in our Gold Standard out of a corpus of 2601, which is 6%.

We labeled each snapshot with one of the following labels.

- *Red*: This category indicates that the coder was writing test code in an attempt to make a failing test
- *Green*: This category is when the coder is writing code in an attempt to make a failing test pass
- *Blue*: This is when the coder has gotten the tests to pass, and is refactoring the code
- *White*: This is when the code is written in a way that deviates from TDD
- *Brown*: This is a special case, when the coder writes a new test, expecting it to fail, but it passes on the first try, without altering the existing production code

This is the Gold Standard that we use to evaluate our inferencer.

3) Inference Evaluation: In the previous section, we describe how we manually label each snapshot with the part of TDD that it corresponds to. We then ran our inference algorithm against the sessions that compose the Gold Standard. We then compare the results of the algorithm at each snapshot and compare it against the labels that were assigned by hand. We next describe how we use this comparison to calculate precision and recall.

4) Accuracy: We calculate the accuracy of our inferencer by using the traditional F-measure. To compute this, we must first compute precision and recall. If the inferencer identifies a snapshot to have the same category that it has in the Gold Standard, we consider this a $TruePositive$. If the inferencer considers a snapshot to be in a different category than in the Gold Standard, we consider this case to be a $FalsePositive$. A $FalseNegative$ is where a snapshot that should have been classified as one of the TDD phases was classified by the inferencer as white (non-TDD).

Once we calculated these for each session in the Gold Standard, we calculate precision and recall using the following formulas.

$$precision = \frac{|TruePositive|}{|TruePositive| + |FalsePositive|}$$

$$recall = \frac{|TruePositive|}{|TruePositive| + |FalseNegative|}$$

Once we can calculated both precision and recall, we calculate accuracy using the following formula.

$$accuracy = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

D. Results

We will now present the results of our empirical evaluation.

1) Precision: The Gold Standard contains 2489 snapshots. Of those, 2028 were correctly identified by the inferencer. This lead to a precision of 81%. The diversity of our corpus leads to a wide variety of TDD implementations, and there are quite a few edge cases. While our algorithm handles many of them, there are still a few edges cases that our algorithm cannot recognize.

2) Recall: There were a total of 1517 snapshots in our Gold Standard that belonged to one of the TDD phases (i.e., non-white phases). Of those, our inferencer only failed to identify 77 of them, leading to a recall of 95%.

3) Accuracy: We calculate the accuracy using the F-measure. This gives us an accuracy of 87%. This shows that our inferencer is accurate and effective.

E. Threats to Validity

Construct Validity: Are we asking the correct questions? Because the inference algorithm is the foundation that everything else is built on, it is important that it be accurate. In order to determine accuracy we are using the widely used information retrieval metrics of precision and recall. This gives us confidence that we are correctly describing the accuracy of our algorithm.

Internal Validity: How did we mitigate bias during manual inspection? To mitigate any kind of bias during the selection of our Gold Standard, we chose them completely at random. This allows us to feel confident that we are not biasing the results via our Gold Standard selection process. When it comes to the content of the katas themselves, they were written by coders who had no knowledge of us or our experiment. This gives us confidence to say that we could not have biased them in any way.

External Validity: Do our results generalize? Our results are specific to the TDD development process. We believe that this work could be adapted for other processes, but we leave that as future work. While we acknowledge that our corpus is not industrial production data, it is a very large and diverse corpus, which was generated by a very wide and diverse set of developers. This diversity results in many different approaches to the TDD process, which provide depth to our study. While we feel this data represents many different TDD process approaches, we do not claim to have fixed all the implementation details that could be a challenge when implementing TDD for industrial production code.

VII. Related Work

We classify the related into two lines of work.

Understanding TDD. Researchers have attempted different approaches to evaluate developer compliance with Test-Driven Development (TDD) practices.

Several projects [35]–[37] developed plugins for the Eclipse IDE in order to record code events. These implementations have all used low-level developer activity to identify the TDD process based on custom-defined rule sets. In order to evaluate their plugins, these projects utilized sample data from a group of 6 experienced developers, 34 fourth-year undergraduate students majoring in computer science, and 1 senior developer that "was asked to develop a web-based system for academic institution" [35, p. 201].

We evaluated our implementation against 160 individual TDD sessions in our Gold Standard [16], sampled from a diverse corpus of 2601 TDD sessions. Our implementation does not require integration with IDEs, or that developers even use an IDE. This allows our algorithm to be both IDE-independent and platform-agnostic. Using these snapshots, our algorithm is able to infer the TDD process without requiring the entire stream of IDE events.

Several projects [38]–[42] build on top of HackyStat [43], a framework for data collection and analysis. Hackystat collects "low-level and voluminous" data, which it sends to a web service for lexical parsing, event stream grouping, and development process analysis. Evaluations for these projects was conducted on a total of 16 undergraduate students and 28 graduate students in four introductory software engineering courses, 20 industry developers (with only 4 developers installing and activating the appropriate plugins), a 1200 source line High-Performance Computing application, and "various programming tasks from an experienced programmer who was learning TDD" [41, p. 8].

Our approach reduces privacy concerns by looking at snapshots, without tracking all low-level developer actions in an IDE. This reduces the privacy concerns about intrusion and data collection. Using AST analysis, the inferencer infers the TDD process without the entire stream of low-level actions.

Understanding Code Changes. While our use of code changes to study software development processes is new, there are many other uses of software changes to understand artifacts. Purushothaman and Perry [44] look at code changes and find that the size of the change impacts the probability that a given change will introduce a defect. Fluri and Gall [45] were able to qualify change couplings by considering software changes. German [46] performed an empirical evaluation on code changes, and created a visualization to show which developer caused which changes. Panetella et al. [47] compared code changes with developer collaborations such as mailing lists and issue trackers. Giger et al. [48] predict code changes using data mining models. However, none of these works were using software changes to identify information about the process used to develop the code.

VIII. Conclusions

Without understanding there can be no improvement. In this paper we presented visualizations that enable developers to better understand the development process. We particularly focused on a process, TDD, which is often misunderstood. We were surprised that there are no visualizations yet to help TDD developers improve. We hope that our visualizations which show conformance to TDD, but also its absence and outliers, lead to a better understanding of TDD.

In order to design these visualizations, we developed an inferencer that infers the TDD process with a novel use of code changes. We evaluated our inferencer and showed that it is accurate and effective, with 81% precision and 95% recall.

In the future, we will explore deeper into our extensive corpus to discover whether following TDD leads to better quality of code. We will also investigate the outliers and we plan to offer novel visualizations that compare a developer's TDD behavior with that of a population of TDD developers.

We hope that others will pursue research on understanding how the current development processes are performed. This deeper understanding will help the community identify characteristics of good processes and ways to improve them.

References

[1] K. D. Prenger, "Costs and benefits of software process improvement," DTIC Document, Tech. Rep., 1997.

[2] D. Paulish and A. Carleton, "Case studies of software-process-improvement measurement," *Computer*, vol. 27, no. 9, pp. 50–57, 09 1994.

[3] D. Janzen and H. Saiedian, "Test-driven development: Concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, Sep. 2005.

[4] R. C. Martin, "The three laws of TDD." [Online]. Available: http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

[5] S. Nerur, R. Mahapatra, and G. Mangalaraj, "Challenges of migrating to agile methodologies," *Commun. ACM*, vol. 48, no. 5, pp. 72–78, May 2005.

[6] K. Beck, *Test Driven Development: By Example*, 1st ed. Boston: Addison-Wesley Professional, Nov. 2002.

[7] D. H. Hansson, "Tdd is dead. long live testing." 2015. [Online]. Available: http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

[8] "TDD leads to an architectural meltdown around iteration three otaku, cedric's blog." [Online]. Available: http://beust.com/weblog/2008/03/03/tdd-leads-to-an-architectural-meltdown-around-iteration-three/

[9] "why tdd is bad - kushal's java blog | software engineering blog." [Online]. Available: http://sanjaal.com/java/tag/why-tdd-is-bad/

[10] H. Munir, K. Wnuk, K. Petersen, and M. Moayyed, "An experimental evaluation of test driven development vs. test-last development with industry professionals," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '14. New York, NY, USA: ACM, 2014, pp. 50:1–50:10.

[11] M. Muller and O. Hagner, "Experiment about test-first programming," *Software, IEE Proceedings -*, vol. 149, no. 5, pp. 131–136, Oct. 2002.

[12] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: Industrial case studies," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ser. ISESE '06. New York, NY, USA: ACM, 2006, pp. 356–363.

[13] H. Erdogmus, G. Melnik, and R. Jeffries, *Test-Driven Development.*, 2010.

[14] T. Munzner, "A nested model for visualization design and validation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 921–928, 11 2009.

[15] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *ECOOP 2012 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, J. Noble, Ed. Springer Berlin Heidelberg, 2012, no. 7313, pp. 79–103.

[16] J. Jagger, "cyber-dojo: the place to practice programming," 2014, [Online; accessed 21-August-2014]. [Online]. Available: http://cyber-dojo.org/

[17] T. R. Green, "Cognitive dimensions of notations," *People and computers V*, pp. 443–460, 1989.

[18] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C#*, 1st ed. Upper Saddle River, NJ: Prentice Hall, Jul. 2006.

[19] K. Beck, "Aim, fire [test-first coding]," *IEEE Software*, vol. 18, no. 5, pp. 87–89, Sep. 2001.

[20] "MSDN TDD," 2015, [Online; accessed 22-Jan-2015]. [Online]. Available: https://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx

[21] J. L. Tim Ottinger, "Unit tests are first," 2015, [Online; accessed 22-Jan-2015]. [Online]. Available: https://pragprog.com/magazines/2012-01/unit-tests-are-first

[22] "JUnit," 2015. [Online]. Available: http://junit.org/

[23] R. B. d. Luz, A. G. S. S. Neto, and R. V. Noronha, "Teaching TDD, the coding dojo style," in *Proceedings of the 2013 IEEE 13th International Conference on Advanced Learning Technologies*, ser. ICALT '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 371–375.

[24] D. Thomas, "Kata, Kumite, Koan, and Dreyfus," 2013, [Online: accessed 31-August-2014]. [Online]. Available: http://codekata.com/kata/kata-kumite-koan-and-dreyfus/

[25] ——, "CodeKata," 2014, [Online; accessed 22-Jan-2015]. [Online]. Available: http://codekata.com/

[26] P. Johnson and H. Kou, "Automated recognition of test-driven development with zorro," in *Agile Conference (AGILE), 2007*, Aug. 2007, pp. 15–25.

[27] M. Krzywinski, I. Birol, S. J. Jones, and M. A. Marra, "Hive plotsrational approach to visualizing networks," *Briefings in Bioinformatics*, p. bbr069, Dec. 2011.

[28] J. Mackinlay, "Automating the design of graphical presentations of relational information," *ACM Transactions on Graphics (TOG)*, vol. 5, no. 2, p. 141, 1986.

[29] T. Munzner, *Visualization Analysis and Design*. CRC Press, 2014.

[30] "Mergely," 2015, [Online; accessed 22-Jan-2015]. [Online]. Available: http://www.mergely.com/

[31] T. R. Green, "Cognitive dimensions of notations," *People and computers V*, pp. 443–460, 1989.

[32] "GumTree," 2015, [Online; accessed 22-Jan-2015]. [Online]. Available: http://www.mergely.com/

[33] J. Jagger, private communication, Aug. 2014.

[34] "Code craftsman saturdays," 2015, [Online; accessed 22-Jan-2015]. [Online]. Available: http://codecraftsmansaturdays.blogspot.com/

[35] L. Madeyski and Ł. Szała, "The impact of test-driven development on software development productivityan empirical study," in *Software Process Improvement*. Springer, 2007, pp. 200–211.

[36] O. Mishali, Y. Dubinsky, and S. Katz, "The TDD-guide training and guidance tool for test-driven development," in *Agile Processes in Software Engineering and Extreme Programming*. Springer, 2008, pp. 63–72.

[37] C. Wege, "Automated support for process assessment in Test-Driven Development," Dissertation, Universitt Tbingen, 2004.

[38] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita, "Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH," in *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE, 2004, pp. 136–144.

[39] P. M. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita, "Improving software development management through software project telemetry," *IEEE software*, vol. 22, no. 4, pp. 76–85, 2005.

[40] P. M. Johnson and M. G. Paulding, "Understanding HPC development through automated process and product measurement with hackystat," in *Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)*. Citeseer, 2005, p. 62.

[41] Y. Wang and H. Erdogmus, "The role of process measurement in test-driven development," in *4th Conference on Extreme Programming and Agile Methods*, 2004.

[42] H. Kou, P. M. Johnson, and H. Erdogmus, "Operational definition and automated inference of test-driven development with zorro," *Automated Software Engineering*, vol. 17, no. 1, pp. 57–85, 11 2009.

[43] P. M. Johnson, "Project hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis," *Department of Information and Computer Sciences, University of Hawaii*, vol. 22, 2001.

[44] R. Purushothaman and D. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 511–526, Jun. 2005.

[45] B. Fluri and H. Gall, "Classifying change types for qualifying change couplings," in *14th IEEE International Conference on Program Comprehension, 2006. ICPC 2006*, 2006, pp. 35–45.

[46] D. M. German, "An empirical study of fine-grained software modifications," *Empirical Software Engineering*, vol. 11, no. 3, pp. 369–393, May 2006.

[47] S. Panichella, G. Bavota, M. Di Penta, G. Canfora, and G. Antoniol, "How developers' collaborations identified from different sources tell us about code changes," in *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2014, pp. 251–260.

[48] E. Giger, M. Pinzger, and H. Gall, "Can we predict types of code changes? an empirical analysis," Jun. 2012, pp. 217–226.