Understand TDD Using Software Changes

Michael Hilton, Nicholas Nelson, Hugh McDonald, Sean McDonald, Ron Metoyer, Danny Dig
Oregon State University
[hiltonm,nelsonni,mcdonalh,mcdonase,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 must be first understood. Our unique approach in this paper uses code and test changes to understand conformance to a process. As a case study, we use these changes to understand conformance to the Test Driven Development (TDD) process.

We designed and implemented TDDViz, a tool that supports developers in better understanding how they conform to TDD. TDDViz supports this understanding by providing novel visualizations of developers’ TDD process. We analyze these visualizations using the Cognitive Dimensions framework to discuss findings and design adjustments. To enable TDDViz’s visualizations, we developed a novel automatic inferencer that identifies the phases that make up the TDD process.

We evaluate TDDViz using two complementary methods: a controlled experiment with 35 participants to evaluate the visualization, and a case study with 2601 TDD Sessions to evaluate the inference algorithm. The controlled experiment shows that, in comparison to existing visualizations, participants performed significantly better when using TDDViz to answer questions about code. In addition, the case study shows that the inferencing algorithm in TDDViz infers TDD phases with an accuracy of 87%.

1. 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 shows that improving development processes leads to 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 key practice of eXtreme Programming and other agile software processes. In TDD a developer writes unit tests before before developing production code in small, rapid iterations [3, 4].

We chose TDD because while it is popular [5, 6], it is a skill that must be honed over time. The availability of a tool that assists in the practice of TDD will help developers improve over time. A better understanding of how a developer follows TDD could also help resolve some of the disagreement [7] and controversy [8, 9] around TDD.

To help developers achieve a better understanding of their process, we examined seminal research [14–17] that found questions software developers ask. From this research, we focused on three question areas. We felt that the answers to these could provide developers with a better understanding of their process. We choose three questions from the literature to focus on, and they spanned three areas: identification, comprehension, and comparability.

RQ1: Identification: “Can we detect testing strategies, such as test-driven development?” [15]
RQ2: Comprehension: “Why was this code changed or inserted?” [17]
RQ3: Comparability: “How much time went into testing vs. into development?” [16]

To answer these questions, we use code and test changes to understand conformance to a 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.

We present TDDViz, our tool which provides visualizations that support developers’ understanding of how they conform to the TDD process. Our visual design is meant to answer RQ1-RQ3 so that we ensure that our visualizations support developers in answering important questions about identification, comprehension, and comparability of code. Our design was guided by the nested model of Munzner [18], a well established design process model that breaks design into 4 nested stages and establishes guidelines for evaluation/validation within each stage.

The resulting visualization design supports developers in identifying the TDD components and overall process, drilling down to a more detailed view of their particular process, and comparing their process to proper TDD practices. This de-
sign is described in detail in Section 3.

These different levels of information allow the developer to avoid irrelevant parts of their development history, while simultaneously accessing detailed information to answer specific questions about their process.

In order to enable these visualizations, we designed a novel algorithm to infer TDD phases. Given a sequence of code edits and test runs, TDDViz uses this algorithm to automatically detect changes that follow 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 fundamental challenge for the inferencer is that during the TDD practice, not all code is developed according to the textbook definition of TDD. Even experienced TDD developers often selectively apply TDD during code development, and only on some parts of their code. This introduces lots of noise for any tool that checks conformance to processes. To ensure that our inference algorithm can correctly handle noisy data, we add a fourth Research Question.

RQ4: Accuracy: “Can an algorithm infer TDD phases accurately?”

In order to train and evaluate our inference algorithm we needed TDD process data. Previous research [19] shows that mining code snapshots stored when developers commit their changes is too coarse-grained and incomplete. One potential way to get finer grained data is to instrument the development environment and record all micro edits and test runs. But this raises privacy and security concerns.

Another potential way to get data 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 [20], 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 a very large, diverse corpus. It contains a total of 41766 fine-grained snapshots from 2601 programming sessions, each of which is an attempt to solve one of 30 different programming tasks. This wide variety makes the task of developing an inferring algorithm difficult, because we cannot target a single skill level or population segment of TDD practitioners. 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 used two other methods. First, we performed a formative analytical assessment of our design using the Cognitive Dimensions [21] 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 [22]. As our second method of evaluation, we performed a controlled experiment with 35 student participants already familiar with TDD. Our independent variable was using TDDViz or existing visualizations to answer questions about code.

This paper makes the following contributions:

- **Process Conformance:** We propose a novel usage of software changes to infer conformance to a process. 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 TDD.

- **TDD Visualization Design and Analysis:** We present a visualization designed specifically for understanding conformance to TDD. Our visualizations show the presence or absence of TDD and allow progressive disclosure of TDD activities. We perform an analysis of the visualizations using the Cognitive Dimensions framework which allows us to discuss some findings and design adjustments.

- **TDD Phase Inference Algorithm:** 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 implement the visualization and inference algorithm in TDDViz, and empirically evaluate it using two complementary methods. First, we conduct a controlled experiment with 35 participants, in order to answer RQ1-3. We find the participants using TDDViz scored significantly better for two of the three questions, and scored marginally significant on the third. Second, we evaluate the accuracy of our inferencer using a corpus of 2601 TDD sessions from cyber-dojo, in order to answer RQ4. Our inferencer achieves an accuracy of 87%. This shows that TDDViz is effective.

2. OVERVIEW OF TDD

“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 [23], leading Agile Development author.

TDD is not a testing technique, it is a software design technique [24]. 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, 23].

One of the foundational technologies required to perform TDD is a unit test framework. This unit test framework should be able to execute all automated tests quickly and easily [25, 26]. When developing Java code, JUnit [27] 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 [28]. Kata [29] 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 [30] 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, one of the 30 programming tasks available on the cyber-dojo site. 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”. [20]

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 proceeds 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 no production code exists yet. Then she 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 she writes the production code to make the test pass.

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

In the fourth TDD cycle, she writes a failing test to assert that multiples of both 3 and 5 print “FizzBuzz”. Then she writes the production code to make this test pass, which she verifies by running all the tests. The developer realizes she duplicated code between the last cycle and the previous two. She then refactors the code to remove duplication and reruns all the tests.

3. VISUALIZATION

In this section we present the visualizations we designed to enable developers to better understand their conformance to TDD. 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 & white. Our design was guided by the nested model of Munzner [18], 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 and Task Abstraction stages.

3.1 Motivations

3.1.1 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.

3.1.2 Questions Software Developers Ask

The second component of domain characterization is identifying the domain specific questions of interest to the target users. In our case, we have identified three questions from prior literature that are particularly important for developers to better understand their development process (see RQ1-RQ3). From these three primary questions, we identified a set of three abstract tasks that the users would have to perform to answer RQ1-RQ3 respectively: 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 abstract operations, the data at hand, and best practices for representing such data and supporting these tasks. In the following sections, we describe our visualization design.

3.2 Visualization Elements

3.2.1 TDD Cycle Plot

We represent a TDD cycle using a single glyph as shown in Figure 1 [a]. This representation was inspired by hive plots [31] 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 time spent in a phase is a quantitative value encoded in the area [32,33] of the cycle plot segment (i.e., the larger the area, the more time spent in that phase during that cycle). Taken together, a single cycle plot forms a glyph or specific ‘shape’ based on the characteristics of the phases, effectively using a ‘shape’ encoding for different types of TDD cycles. This design supports both characterization of entire cycles as well as comparison of a developer’s time distribution in each phase of a cycle. We illustrate the shape patterns of various TDD cycles in the next section.

3.2.2 TDD Heartbeat

To support comparison of TDD cycles over time, we provide a small multiples view [34] that we call the TDD Heartbeat view. The TDD Heartbeat view consists of a series of
TDD cycle plots, one for every cycle of that session (See Figure 1, [b]) We call this the TDD heartbeat because this view gives an overall picture of the health of the TDD process as it evolves over time. This particular view particularly supports the abstract tasks of characterization and comparison.

In particular, the user can compare entire cycles over time to see how they evolve, and she can characterize how her process is improving or degrading.

For example, by looking at all the cycles that make up the TDD Heartbeat in Figure 1, the user sees that for every cycle in this data, the developer spent relatively more time writing production code than writing tests. They can also observe that the relationship between the time spent in each phase was fairly consistent.

3.2.3 Snapshot Timeline

The snapshot timeline provides more information about the TDD process, specifically showing all the snapshots in the current session. An example snapshot timeline is shown in Figure 1 [c]. The snapshot timeline consists of two parts, the snapshot classification bar (F. 1 [c][1]) on the top, and the snapshot event timeline on the bottom (F. 1 [c][2]). In the snapshot event timeline, 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, we represent this with an empty white rounded box. We encode time or order positionally, left to right, a typical and effective time encoding. The distance between each snapshot is evenly distributed, since the time in that phase is encoded in the TDD Cycle Plot.

The snapshot classification bar shows the cycle boundaries, and inside each cycle the ribbon of red, green and blue signifies which snapshot events fall into which phases. For example, in Figure 1, snapshots 17-19 are all part of the same green phase.

This view answers questions specifically dealing with how consistent coders followed the TDD process, what snapshots were written by coders using the TDD process, and which ones were not.

The snapshot timeline answers questions about identification. The timeline enables developers to identify which parts of the session conform to TDD and which do not. Figure 2 shows an example snapshot timeline where the TDD process was not followed consistently. In this specific example, the developer did not use TDD for their first 10 snapshots, and even after that, there was one very long green stretch that dominated the development session, from snapshot 16-25. This contrasts with Figure 1 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. In Figure 1, snapshots 5 and 6 are selected.

The snapshot timeline also answers questions dealing with comprehension. By seeing how TDDViz categorizes a snapshot, a user can determine why the selected changes were made. For example, Figure 1 shows a selected snapshot which represents the changes between snapshots numbers 5 and 6. Since the selected changes are part of a green phase (as noted by the green area in the Snapshot Classification Bar), a user can determine that these were production changes to make a failing test pass. This can be confirmed by observing the code edits.

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.

3.2.4 TDD Code Edits

Figure 1, [d] shows an example of a code edit, which displays the changes to the code between two snapshots. 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 two arbitrary snapshots. The code edit region contains an expandable and collapsable box for each file that was changed in the selected range of snapshots. Each box contains two code editors, one for the code at the selection’s starting snapshot, and one for the code at the ending snapshot. The code is displayed in a visual diff tool [35] in order to succinctly show the differences.

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

There are additional examples of our visualizations on our accompanying web page http://cope.eecs.oregonstate.edu/visualization.html.

4. 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 [21] is a broad brush usability analysis technique that was originally introduced for analyzing visual programming languages. Researchers have broadened its use 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.

4.1 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 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 earlier 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 inconsistency in that the snapshot timeline does not contain snapshots that are blue, to correspond to the refactoring phase. This is left for future work.

4.2 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 of the user 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 in the design in order to align the Heartbeat cycle plots with the corresponding snapshots in the timeline.

4.3 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.

4.4 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.

5. TDD PHASE INFERENCE

In order to be able to build the visualizations we have presented thus far, we needed to build a TDD phase inference algorithm which uses test and code changes to infer the TDD process. Instead of relying on static analysis tools, we present a novel approach where the algorithm analyzes the changes to the code. We designed our algorithm to take as input a series of snapshots. The algorithm then analyzes 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.

5.1 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 uses these snapshots to determine the developers changes to the program. It then uses these changes to infer the TDD process. In this paper, we use 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 [19] has shown that VCS commits are incomplete and imprecise when trying to study code changes.

5.2 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 than just the textual contents. To this end, our inference algorithm constructs 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 to the test code. It also calculates the number of methods and assert statements at each snapshot. For the purposes of the algorithm, we consider each assert to be an individual test. If a new assert is detected, we consider that to be a new test. All this information enables the algorithm to infer the phases of TDD. In our implementation of the algorithm in TDDViz, we use the Gumtree library [36] to create the ASTs.

5.3 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.

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)

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 occurs when no compilation errors exist and all the tests pass.

**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.

**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.

**White.** We define the white phase as follows:
- All remaining edits that do not conform to any of the TDD phases as described above.

### 5.4 Algorithm

We present the TDD phase inference algorithm using the state diagram in Figure 3. Our algorithm encodes a finite-state machine (FSM), where the state nodes are phases, and the transitions are guided by predicates on the current snapshot. The predicates take a snapshot and using the AST changes to the production and test code, as well as the result of the test runs, compute a boolean function. We compose several predicates to determine a transition to another state.

For example: in order to transition from green to blue, the following conditions must hold true. All the current unit tests must pass, and the developer may not add any new tests.

The transition requires passing tests, because if not, the developer either remains in the green phase or has deviated from TDD. No new tests are allowed because the addition of a new test, while a valid TDD practice, would signify that the developer has skipped the optional blue phase and moved directly to the Red phase.

There are a few special cases in our algorithm. The algorithm’s transition from Red to Blue is the case when a single snapshot comprised the entire Green phase, and therefore the algorithm has moved on to the blue phase. Another thing to note is that by definition, the brown phase only contains a single commit. Therefore, after the algorithm identifies a brown phase, it immediately moves back to the blue phase.

### 6. EVALUATION

To evaluate the usefulness of TDDViz, we answer the following research questions:

**RQ1. Identification:** Can programmers using TDDViz identify whether the code was developed in conformance with TDD?

**RQ2. Comprehension:** Can programmers using TDDViz identify the reason why code was changed or inserted?

**RQ3. Comparability:** Can programmers using TDDViz determine how much time went into testing vs. development of production code?

**RQ4. Accuracy:** Can an algorithm infer TDD phases accurately?

In order to answer these research questions, we used two complementary empirical methods. We answer the first three questions with a controlled experiment with 35 participants, and the last question with a case study of 2601 TDD sessions. The experiment allows us to quantify the effectiveness of the visualization as used by programmers, while the case study gives more confidence that the proposed algorithm can handle a wide variety of TDD instances.

#### 6.1 Controlled Experiment

**Participants.** We asked all 35 students in a 3rd-year undergrad Software Engineering class at Oregon State University whether they want to volunteer in a TDD experiment. We neither rewarded nor penalized students for participation in the experiment. We chose these students as subjects because of their familiarity with TDD. The students were introduced to TDD in the second week of the term and continued using TDD and Extreme Programming to develop the code for their class project. We performed the experiment in the last week (week 10) of the term.

**Treatment.** Our study consisted of two treatments. For the experimental treatment, we asked the participants to answer questions dealing with identification, comprehension, and comparability (RQ1–RQ3) by examining several coding sessions from cyber-dojo presented with our visualization. For the control treatment, we used the same questions applied to the same real-life code examples, but the code was visualized using the visualization [37] that is available on the cyber-dojo site. This visualization shows both the code, and the test results at each snapshot, but it does not present any information regarding the phases of TDD. We used this visualization for our control treatment because it is specifically designed to view the data in our corpus. Also, it is the only available visualization other than our own which shows both the code and the history of the test runs.

**Experimental procedure.** In order to isolate the effect of our visualization, both treatments had the same introduction, except for when describing the parts of the visualizations which are different across treatments. Both treatments received the exact same questions on the same real-life data in the same order. The only independent variable was which visualization was presented to each treatment. We randomly assigned the students into two groups, one group with 17 participants and the other group with 18 participants. We then flipped a coin to determine which group received which treatment. We gave both treatments back to back on the same day.
6.2 Controlled Experiment Results

Table 1 tabulates the results for the three questions. We will now explain each result in more detail.

**RQ1: Identification.** When we asked the participants to identify TDD, we found that significantly more participants correctly identified TDD and non-TDD sessions using TDDViz than when using the default cyber-dojo visualization, as Table 1 shows (Fisher’s Exact Test: p < 0.0005). This shows that our visualization does indeed aid in identifying TDD.

**RQ2: Comprehension.** When we asked participants why a specific code change had been made. Specifically we asked them to identify if the given code was changed or inserted to make a test pass, make a test fail, or to refactor. We found that significantly more participants correctly identified why the code was changed when using TDDViz than when using the default cyber-dojo visualization (see Table 1: Comprehension, Fisher’s Exact Test: p < 0.0013).

**RQ3: Comparability.** When we asked our participants to compare the amount of time that went into writing tests vs. the time that went into writing code, participants using TDDViz were able to outperform those using cyber-dojo but only by a small margin. The difference was only just approaching significance (Fisher’s Exact Test: p < 0.0578). Additionally, as Table 1: Comparability shows, there were slightly more incorrect answers than correct answers for the experimental group. To answer this question, users had to mentally quantify whether the chart contained more red than green overall. In the future we plan on improving the visualization by providing a representation that provides a clear, numerical answer to this question.

6.3 Case Study

We now answer our fourth research question, which measures the accuracy of the TDD phase inference part of TDDViz, using a corpus of 2601 TDD sessions.

**Corpus Origin.** 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 is not industrial production data, it is a large and diverse corpus. 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. This provides a much more realistic environment for evaluating our algorithm. 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 inferences can make no assumptions about the existence of TDD 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), various mathematical problems (e.g., leap year calculator), etc. 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.

**The Process of doing a Kata in cyber-dojo.** Each TDD session begins with a generic failing test that is provided by cyber-dojo. The developer can edit the production code or tests, or they can run the tests. Whenever the developer runs the tests, cyber-dojo presents three possible outcomes: test failure, test pass, or compilation error. After the tests are run, 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 can 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 we can also study how the kata evolved over time, by looking at these commits.

Cyber-dojo collects all the data anonymously. Thus, cyber-dojo is a “safe-to-fail” environment [38], where developers focus on improving, and need not be concerned about the judgment of others.

**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 Craftsmen Saturdays [39]. 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 Saturdays event and we received 39 responses. Our respondents were from 16 different states around the US. The group is almost evenly split between experienced and beginners. All of our respondents are developers, half of them are men, and half of them are women.
man Saturday during the last year. Among the 92 people who received the demographic survey, 26 responded in the two days the survey was active.

Table 2 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.

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

<table>
<thead>
<tr>
<th></th>
<th>1-2</th>
<th>2-5</th>
<th>5-10</th>
<th>&gt;10</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than</td>
<td>4%</td>
<td>16%</td>
<td>24%</td>
<td>12%</td>
</tr>
<tr>
<td>1 year</td>
<td>11%</td>
<td>15%</td>
<td>19%</td>
<td>19%</td>
</tr>
</tbody>
</table>

Q2: How would you describe yourself?

<table>
<thead>
<tr>
<th></th>
<th>Frequency</th>
<th>Sometimes</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>student</td>
<td>34%</td>
<td>42%</td>
<td>23%</td>
</tr>
<tr>
<td>programmer</td>
<td>61%</td>
<td>15%</td>
<td></td>
</tr>
<tr>
<td>analyst</td>
<td>15%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>designer</td>
<td>19%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>other</td>
<td>19%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Q3: Do you practice TDD at work?

<table>
<thead>
<tr>
<th></th>
<th>Frequently</th>
<th>Sometimes</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>44%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brown</td>
<td>19%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Green</td>
<td>34%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blue</td>
<td>15%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Petent</td>
<td>11%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Code Craftsman Survey shows demographics of its users.

Evaluation Corpus. To build our corpus we used all the Java/JUnit sessions as our algorithm only parses Java at the moment. This gives us a corpus of 2601 total Java/Junit sessions. We use this corpus to evaluate our inferencer.

Corpus Preparation. We developed a Ruby on Rails application that allowed us to work with this corpus in an efficient manner. The raw data that we used to build the corpus consists of a repository and session data. The git repository 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 data contains meta-data files that track events 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.

The Gold Standard. In order to evaluate our phase inferencer, we created a Gold Standard. The first two authors manually labeled 2489 snapshots from 2601 sessions with the TDD phase to which they belong. In order to not bias the selection process, we randomly selected the sessions for our Gold Standard. We graded our inferencer by comparing its results against the 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 Gold Standard on 52 sessions (32% of the sessions).

Once we were convinced that we had reached a consensus among the raters, we divided the rest of the Gold Standard sessions up and rated them individually. We labeled all snapshots in a total of 160 sessions in our Gold Standard out of a corpus of 2601 sessions, which is 6% of the data.

We classified every snapshot in the snapshot event timeline, by manually marking the snapshot classification bar with the correct TDD phase.

- **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
- **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
- **White**: This is all other code which is written in a way that deviates from TDD

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

Inference Evaluation. After we manually labeled each snapshot, we ran our inference algorithm against the sessions that compose the Gold Standard. We then compared the results of the algorithm at each snapshot and compared it against the labels that were assigned by hand.

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 True Positive. If the inferencer considers a snapshot to be in a different category than in the Gold Standard, we consider this a False Positive. A False Negative 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:

\[
\text{precision} = \frac{|\text{TruePositive}|}{|\text{TruePositive}| + |\text{FalsePositive}|}
\]
\[
\text{recall} = \frac{|\text{TruePositive}|}{|\text{TruePositive}| + |\text{FalseNegative}|}
\]

We calculate accuracy using the traditional harmonic mean of precision and recall.

6.4 Case Study Results

**Precision.** The Gold Standard contained 2489 snapshots. Of those, 2028 were correctly identified by the inferencer. This led 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 edge cases that our algorithm cannot recognize in its current form. Many of these cases the authors disagreed on how to classify them.

**Recall.** Our Gold Standard contains 1517 snapshots that belong to one of the TDD phases (i.e., non-white phases). Of those, our inferencer correctly classified 1440, leading to a recall of 95%. Of the remaining 5% missed cases, most of them arise because of how the test code evolved from the initial default test case. This is an issue that can be solved in our future work.
RQ4: 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.

6.5 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. Additionally, to avoid a single author biasing the results, we verified that there was at least an 85% inter-rater agreement between the authors who labeled the Gold Standard, using standard techniques [40].

When it comes to the content of the katatas 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 large and diverse corpus, which was generated by a 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.

Our controlled experiment was conducted with students, so there is a possibility that our results could not generalize for all TDD developers. However, these were all students in at least their third year of university, and we gave them the study the very last day of a term during which TDD was one of the core subjects. During the term they were required to use TDD during their development of the class project. So while they are not professional developers, we know that they at least know the concepts of TDD, and have put it into practice in at least one project.

7. RELATED WORK

We classify the related work into two lines of work: (i) understanding TDD, and (ii) mining fine-grained changes.

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

Multiple projects [41, 42] detect the absence of TDD activities and give warnings when a developer deviates from TDD. These implementations have used low-level developer activity inside of the IDE based on custom-defined rule sets to identify when a developer spends too much time writing code without tests.

In contrast, TDDViz provides detailed analysis of the TDD phases, infers the presence or absence of TDD not based on time intervals between test runs, but on code and test changes. Thus, it is much more precise.

TDD Dashboard [49] is a service offered by Industrial Logic, to visualize several metrics about the TDD process: the number of test runs per day, number of refactorings per day, number of TDD cycles per day. The data is consolidated over an entire team. TDD Dashboard is based on recording test and refactoring events from the IDE. However, unlike the TDD Dashboard, TDDViz does not simply tally the low level TDD events or TDD cycles. Instead, TDDViz infers and visualizes the phases the compose each cycle, thus enabling developers to answer questions on identification, comprehension, and comparability.

Several projects [43–47] infer TDD phases from low-level IDE edits. They all build on top of HackyStat [48], 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” [46, p. 8].

In contrast to these approaches, TDDViz reduces privacy concerns by analyzing snapshots, without tracking all low-level developer actions in an IDE. This reduces the privacy concerns about intrusion and data collection. Using AST analysis, TDDViz infers the TDD process without the entire stream of low-level actions. In addition, TDDViz 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. Moreover, our evaluation corpus is order of magnitudes larger. Lastly, these approaches do not provide any TDD visualizations.

Mining Fine-Grained Changes. Several researchers [19, 50–56] have presented applications of mining fine-grained code changes. Negara et al. [19] found that mining fine-grained code changes is necessary because mining version control snapshots is often incomplete and imprecise, missing up to 30% of the changes that happened in the code. Robbes and Lanza [51] found that the time between developer initiated snapshots often is on the order of several hours or days, thus VCS snapshots are too coarse-grained. Such studies motivate several researchers to treat fine-grained changes as first-class citizens. For example, Negara et al. [50] mined fine-grained code changes to detect frequent change patterns. Robbes and Lanza propose a tool, SpyWare [52] which captures data from an IDE directly as it happens, and it has several novel applications for a change-centric IDE. Hattori and Lanza [57] developed a tool called Syde which mines fine-grained source code changes to refine code ownership. Ebnaat [53] proposed that fine-grained changes could be used to make Feature-oriented Programming more expressive. Yoon et al. [54] developed a visualization for fine-grained code change history in the IDE. This visualization shows changes made in the IDE, but does not infer any additional information. Gu et al. [55] developed a fine-grained code change recorder called IDE++ which displays a timeline visualization of the user’s interactions with the IDE.

These approaches differ from ours in that we are using...
8. 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 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 inference that infers the TDD process with a novel use of code changes. We implemented the visualizations and the inferencer in a tool, TDDViz. We evaluated TDDViz using two complementary methods. We evaluated the visualization using 35 participants. We found that participants that used our visualization had significantly more correct answers when answering questions on identification, comprehension, and comparability of code. We evaluated the TDD phase inferencer and showed that it is accurate and effective, with 81% precision and 95% recall.

In the future, we plan to explore deeper into our extensive corpus to determine 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.

9. REFERENCES


[38] Jon Jagger. private communication, August 2014.


Figure 1: Conforming to TDD. Sizes in the TDD Heartbeat plot represent time spent in each phase. The different parts of the visualization have been labeled for clarity: [a] a TDD Cycle plot, [b] TDD Heartbeat, [c] Snapshot Timeline, [d] TDD Code Edits.

Figure 2: Non-conforming to TDD. Sizes in the TDD Heartbeat plot represent time spent in each phase.

Figure 3: Pseudo-code of the TDD Phase Inference Algorithm.