TDDViz:
Using Software Changes to Understand Conformance to Test Driven Development

Michael Hilton, Nicholas Nelson, Hugh McDonald, Sean McDonald, Ron Metoyer, and 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 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. To enable TDDViz's visualizations, 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 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 evolution. In addition, the case study shows that the inferencing algorithm in TDDViz infers TDD phases with an accuracy (F-measure) of 87%.

Key words: Test Driven Development, Software Visualization, Development Process

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.

In this paper we use Test Driven Development (TDD) as a case study on how software changes can illuminate the development process. To help developers achieve a better understanding of their process, we examined seminal research [3–5] 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: *"Can we detect strategies, such as test-driven development?" (Identification)* [3]
RQ2: *"Why was this code changed or inserted?" (Comprehension)* [5]
RQ3: *"How much time went into testing vs. into development?" (Comparability)* [4]

To answer these questions, we use code and test changes to understand conformance to a process. In this paper, 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 RQ1-3 so that we ensure that our visualizations support developers in answering important questions about *identification*, *comprehension*, and *comparability* of code.

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 proceses. To ensure that our inference algorithm can correctly handle noisy data, we add a fourth Research Question.
RQ4: *"Can an algorithm infer TDD phases accurately?" (Accuracy)*

To answer this question, in this paper we use a corpus of data from cyber-dojo[1] , a website that allows developers to practice and improve their TDD by coding solutions to various programming problems. Each time a user run tests, the code is committed to a git repository. Each of these commits becomes a fine-grained commit. Our corpus 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.

To evaluate TDDViz, 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 the TDD Process.

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

[1] www.cyberdojo.org

show the presence or absence of TDD and allow progressive disclosure of TDD activities.
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**. 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%. Together, both of these show that TDDViz is effective.

2 Visualization

2.1 Visualization Elements

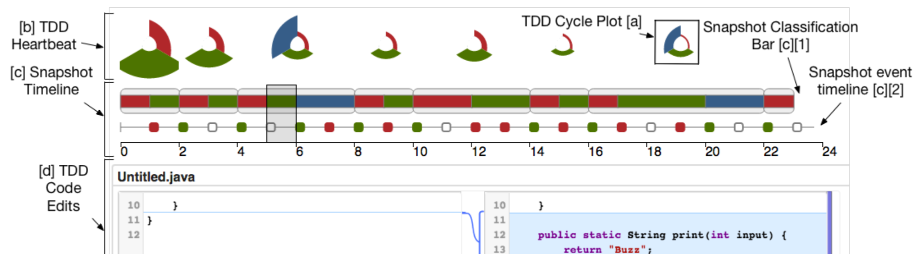

Fig. 1: Interactive visualization of a TDD session. The user can choose any arbitrary selection they wish. This example shows a session that conforms to TDD. Sizes in the TDD Heartbeat plot represent time spent in each phase. The different parts of the visualization have been labeled for clarity. [a] a TDD Cycle plot, [b] TDD Heartbeat, [c] Snapshot Timeline, [d] TDD Code Edits

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 [7] 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 [8, 9] of the cycle plot segment (i.e., the larger the area, the more time spent in that phase during that cycle). All subplots are on the same fixed scale. 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.

TDD Heartbeat To support comparison of TDD cycles over time, we provide a small multiples view [10] 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 kata, 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.

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. 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 17-20 are all part of the same green phase. Snapshots 17-19 the developer is trying to get to a green, but they are not successful in making the tests pass until snapshot 20.

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.

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 catagorizes a snapshot, a user can determine why 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, but, it does so at a finer granularity, showing each individual test run.

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.

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`.

3 TDD Phase Inferencer

In order to build the visualizations we have presented thus far, we needed 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.

3.1 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.

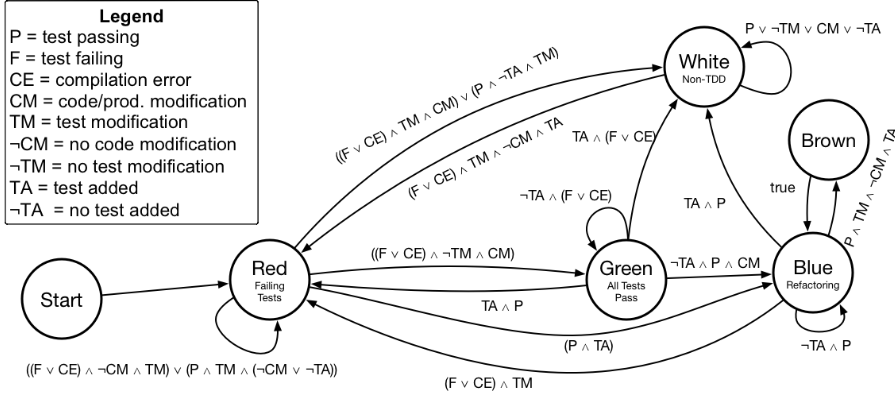

Fig. 2: Pseudo-code of the TDD Phase Inference Algorithm.

3.2 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 code with asserts to be test codes, and code with no asserts to be production code. We consider each assert to be an individual test, even if it is in a method with other asserts. 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 [11] to create the ASTs.

3.3 Algorithm

We present the TDD phase inference algorithm using the state digram in Figure 2. 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.
We define each of the states as follows:
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 and it passes on the first try, without altering the existing production code. It could be they

were expecting it to fail, or perhaps they just wanted to provide extra tests for extra security.

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.

4 Evaluation

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

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

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

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

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

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.

4.1 Controlled Experiment

Participants. Our participants were 35 students in a 3rd-year undergrad Software Engineering class who were in week 10 of a course in which they had used TDD for their class project.

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[2] 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.
Tasks. The experiment consisted of three tasks. To evaluate *identification*, we asked "Is this entire session conforming to TDD?" To evaluate *comprehension* we asked "Why was this code changed?" To evaluate *comparability* we asked "Was more time spent writing tests or production code?" For each task we asked the same question on four different instances of TDD sessions. The students we accustomed to using clickers to answer questions, and so for each task they answered questions using questions with their clickers. Each question was a multiple choice question.
Measures. The independent variable in this study was the visualization used to answer the questions. The dependent measure was the answers to the questions. For each of the three tasks we showed the subjects four different instances and evaluated the total correct responses against the total incorrect responses. We then looked at each question and compared the control treatment versus the experimental treatment. We used Fisher's Exact Test to determine significance because we had non-parametric data.

4.2 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-

[2] http://cyber-dojo.org/dashboard/show/C5AE528CB0

	Identification		Comprehension		Comparability	
Treatment	Correct	Not Correct	Correct	Not Correct	Correct	Not Correct
Control ($n = 18$)	37	34	24	48	23	49
Experimental ($n = 17$)	55	13	42	26	33	35

Table 1: Results from Controlled Experiment.

TDD sessions using TDDViz than when using the default cyber-dojo visualization, as Table 1 shows (Fisher's Exact Test: p<.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, 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<.0013). They were able to identify if the given code was changed or inserted to make a test pass, make a test fail or to refactor.

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, the experimental participants were able to outperform the control group 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 then 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.

4.3 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.
Evaluation Corpus. To build our corpus we used *all* the Java/JUnit sessions as our evaluation framework currently only supports Java. Adding other languages would be straightforward, but is left as future work. 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.

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 things such as when the session occurred, and what was the result of each compile and test run.

The Gold Standard. In order to evaluate our phase inferencer, we created a Gold Standard. The first two authors manually labeled 2489 snapshots with the TDD phase to which they belong.

We then graded our inferencer by comparing its 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 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 a total of 2489 snapshots in our Gold Standard out of a corpus of 41766 snapshots in the corpus, which is 6% of the data. We labeled each snapshot as previously defined in Section 3.3.

Inference Evaluation. After we manually labeled each snapshot, we 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.

Accuracy. We calculate the accuracy of our inferencer by using the traditional F-measure, which considers both precision and recall. We compute precision and recall by first identifying $True$ and $False\ Positives$. 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 the Gold Standard, we consider this case to be 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 standard formulas. Next we calculate accuracy using the traditional harmonic mean of precision and recall.

4.4 Case Study Results

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 edge cases that our algorithm cannot recognize in its current incarnation. These are cases that are hard even for human experts to agree upon.

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 difficulty identifying the template code the katas start with. This is an issue that can be easily 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.

5 Related Work and Conclusions

Related Work Multiple projects [12, 13] detect the absence of TDD activities and give warnings when a developer deviates from TDD by identifying when a developer spends too much time writing 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.

Several projects [14, 15] infer TDD phases from low-level IDE edits. They all build on top of HackyStat [16], 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. In contrast to these approaches, by using AST analysis, TDDViz infers the TDD process without the entire stream of low-level actions.

TDD Dashboard[3] is a service offered by Industrial Logic, to visualize the TDD process. It is based on recording test and refactoring events in a IDE, but does not infer and visualize the phases of each cycle, thus enabling developers to answer questions on identification, comprehension, and comparability.

Conclusions Without understanding there can be no improvement. In this paper we presented visualizations that enable developers to better understand the development process. To design these visualizations, we developed an inferencer 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.

Acknowledgements We thank Mihai Codoban, Kendall Bailey and anonymous reviewers for their feedback on earlier versions of this paper. We thank Jon Jagger for making the TDD data available. We also thank Lutz Prechelt for his helpful and very thorough review. This work was partially funded through the NSF CCF-1439957 grant.

[3] https://ecoach.industriallogic.com/ dashboard?team=il

References

1. Karen D Prenger. Costs and benefits of software process improvement. Technical report, DTIC Document, 1997.
2. D.J. Paulish and A.D. Carleton. Case studies of software-process-improvement measurement. *Computer*, 27(9):50–57, 09 1994.
3. Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.
4. Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *ICSE 2014*, June 2014.
5. Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *PLATEAU '10*, pages 8:1–8:6, 2010.
6. T. Munzner. A nested model for visualization design and validation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):921–928, 11 2009.
7. Martin Krzywinski, Inanc Birol, Steven JM Jones, and Marco A. Marra. Hive plots, rational approach to visualizing networks. *Briefings in Bioinformatics*, page bbr069, December 2011.
8. J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics (TOG)*, 5(2):141, 1986.
9. Tamara Munzner. *Visualization Analysis and Design.* CRC Press, 2014.
10. Edward R Tufte and PR Graves-Morris. *The visual display of quantitative information*, volume 2. Graphics press, 1983.
11. Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.
12. Oren Mishali, Yael Dubinsky, and Shmuel Katz. The TDD-guide training and guidance tool for test-driven development. In *Agile Processes in Software Engineering and Extreme Programming*, pages 63–72. Springer, 2008.
13. Christian Wege. *Automated support for process assessment in Test-Driven Development.* Dissertation, Universitat Tubingen, 2004.
14. Yihong Wang and Hakan Erdogmus. The role of process measurement in test-driven development. In *XP/Agile Universe '04*, 2004.
15. Hongbing Kou, Philip M. Johnson, and Hakan Erdogmus. Operational definition and automated inference of test-driven development with zorro. *Automated Software Engineering*, 17(1):57–85, 11 2009.
16. Philip 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*, 22.