IEEE Software
www.computer.org/software

Does Test-Driven Development Really Improve Software Design Quality?

David S. Janzen, California Polytechnic State University, San Luis Obispo
Hossein Saiedian, University of Kansas

Vol. 25, No. 2
March/April 2008

feature

software metrics

Does Test-Driven Development Really Improve Software Design Quality?

David S. Janzen, *California Polytechnic State University, San Luis Obispo*

Hossein Saiedian, *University of Kansas*

TDD is first and foremost a design practice. The question is, how good are the resulting designs? Empirical studies help clarify the practice and answer this question.

Software developers are known for adopting new technologies and practices on the basis of their novelty or anecdotal evidence of their promise. Who can blame them? With constant pressure to produce more with less, we often can't wait for evidence before jumping in. We become convinced that competition won't let us wait.

Advocates for test-driven development claim that TDD produces code that's simpler, more cohesive, and less coupled than code developed in a more traditional test-last way. Support for TDD is growing in many development contexts beyond its common association with Extreme Programming. Examples such as Robert C. Martin's bowling game demonstrate the clean and sometimes surprising designs that can emerge with TDD,[1] and the buzz has proven sufficient for many software developers to try it. Positive personal experiences have led many to add TDD to their list of "best practices," but for others, the jury is still out. And although the literature includes many publications that teach us how to do TDD, it includes less empirical evaluation of the results.

In 2004, we began a study to collect evidence that would substantiate or question the claims regarding TDD's influence on software.

TDD misconceptions

We looked for professional development teams who were using TDD and willing to participate in the study. We interviewed representatives from four reputable Fortune 500 companies who claimed to be using TDD. However, when we dug a little deeper, we discovered some unfortunate misconceptions:

- Misconception #1: TDD equals automated testing. Some developers we met placed a heavy emphasis on automated testing. Because TDD has helped propel automated testing to the forefront, many seem to think that TDD is only about writing automated tests.
- Misconception #2: TDD means write all tests first. Some developers thought that TDD involved writing the tests (all the tests) first, rather than using the short, rapid test-code iterations of TDD.

Unfortunately, these perspectives miss TDD's primary purpose, which is design. Granted, the tests are important, and automated test suites that can run at the click of a button are great. However,

0740-7459/08/$25.00 © 2008 IEEE

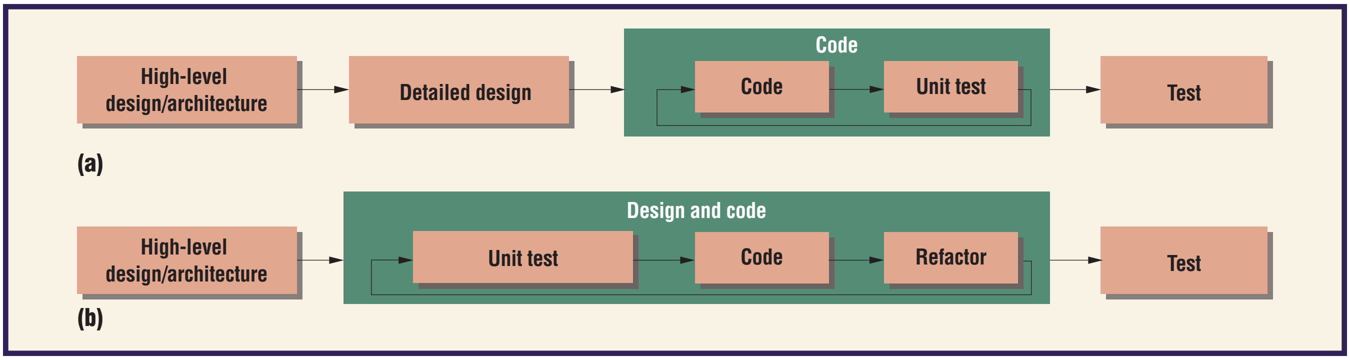


Figure 1. Development flow: (a) traditional test-last and (b) test-driven development/test-first flow.

from early on, TDD pioneers have been clear that TDD is about design, not the tests.[2]

Why the confusion regarding TDD? We propose two possible explanations.

First, we can blame it on the name, which includes the word "test" but not the word "design." But, alas, "test-driven development" seems to be here to stay. We're unlikely to revert to earlier, more accurately descriptive names such as "test-driven design."

A second source of confusion is the difference between internal and external quality. Several early studies focused on TDD's effects on defects (external quality) and productivity.[3] Many results were promising although somewhat mixed. Boby George and Laurie Williams reported fewer defects but lower productivity.[4] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano reported minimal external quality differences but improved productivity.[5] Adam Geras, Michael Smith, and James Miller reported no changes in productivity, but more frequent unplanned test failures.[6] The emphasis on external quality is valid and beneficial, but it can also miss TDD's primary focus on design.

Matthias Müller addressed internal quality in a recent case study comparing five open-source and student TDD projects with three open-source non-TDD projects.[7] (The study incorrectly identified one TDD project, JUnit, as being non-TDD, and it didn't confirm whether two projects, Ant and log4j, were TDD or non-TDD.) Although Müller focused on a new metric to gauge testability, he indicated that software developed with TDD had lower coupling, smaller classes, and higher testability, but less cohesiveness.

Despite the misconceptions about TDD, some of the traditional test-last development teams we interviewed reported positive experiences with automated testing, resulting in quality and productivity improvements. Other test-last teams reported frustrations and eventual abandonment of the approach. We believed that focusing on internal qualities, such as simplicity, size, coupling, and cohesion, would emphasize TDD's design aspects and help clarify how to use it.

TDD in a traditional development process

We wanted to examine TDD independent of other process practices, but we had to select a methodology to minimize independent variables. We chose to study TDD in the context of a somewhat traditional development process based on the Unified Process.[8] The projects in this research were relatively short (typically three to four months). We believe the process we used could be repeated as iterations in a larger evolutionary process model, but we didn't study this.

Figure 1a illustrates a traditional test-last flow of development. This process involves significant effort in specifying the system architecture and design before any significant software development. Such an approach does not preclude some programming to explore a prototype or prove a concept, but it assumes that no significant production software is constructed without a detailed design. Unit testing occurs after a unit is coded. We asked test-last programmers in the study to use an iterative approach in which the time from unit construction to unit testing was very short (seconds or minutes rather than weeks or months).

Figure 1b illustrates the test-first development flow. In this approach, the project identifies some high-level architecture early, but that design doesn't proceed to a detailed level. Instead, the test-first process of writing unit tests and constructing the units in short, rapid iterations allows the design to emerge and evolve.

Neither of these flows makes any assumptions about other process practices.

Study design and execution

We designed our study to compare the test-first TDD approach with a comparable but reversed test-last approach. In particular, programmers in both

Table 1
Study profile

		Test-First				Test-Last			
Study*	Experiment Type	Classes	LOC	Teams†/ experience*	Technologies/ real world?	Classes	LOC	Teams†/ experience	Technologies/ real world?
INT-TF	Quasi-controlled	28	842	A/>5 years	J2EE, Spring/ real world	18	1,562	A/>5 years	J2EE/real world
ITL-TF	Quasi-controlled	28	842	A/>5 years	J2EE, Spring/ real world	21	811	AB/>5 years	J2EE/real world
ITF-TL	Quasi-controlled	69	1,559	ABC/>5 years	J2EE, Spring, Struts/real world	57	2,071	BC/>5 years	J2EE, Spring, Struts/real world
ICS	Case study	126	2,750	ABC/>5 years	J2EE, Spring, Struts/real world	831	49,330	ABCDE‡/ >5 years	J2EE, Spring, Struts/real world
GSE	Quasi-controlled	19	1,301	Two teams of 3 participants/ 0–5 years	Java/ academic	4	867	One team of 3 participants/ >5 years	Java/academic
USE	Quasi-controlled	28	1,053	One team of 3 participants/ novice	Java/academic	17	1,254	Two teams of 3 and 4 participants/ novice	Java/academic
Unique totals		173	5,104	12 participants	N/A	852	51,451	15 participants	N/A

* *INT-TF (industry no-tests followed by test-first), ITL-TF (industry test-last followed by test-first), ITF-TL (industry test-first followed by test-last), ICS (industry case study), GSE (graduate software engineering), USE (undergraduate software engineering).*

† *A, B, C, D, and E identify five developers to show overlap between teams.*

‡ *One of the early test-last projects had additional developers.*

the test-first and test-last groups wrote automated unit tests and production code in short, rapid iterations. We conducted pre-experiment surveys to ensure no significant differences existed between the test-first and test-last groups in terms of programming experience, age, and acceptance of TDD. The only difference was whether they wrote the tests before or after writing the code under test.

We selected a development group in one company to conduct three quasi-controlled experiments and one case study. (We call the studies quasi-controlled because the teams weren't randomly assigned.) We selected this group because of their willingness to participate in the study, to share project data, and to use TDD as an integral part of design. Developers voluntarily participated as part of their regular full-time work for the company, which assigned all projects and used the results in production.

In addition, we conducted two quasi-controlled experiments in undergraduate and graduate software engineering courses at the University of Kansas during the summer and fall of 2005. Nineteen students worked in teams of three or four programmers each. Both courses involved the same semester-long project.

Table 1 summarizes the studies. The three industry quasi-controlled experiments involved five similar but distinct projects completed by overlapping groups of five developers. The "Teams" columns in table 1 identify these developers with the letters A through E and indicate how the teams overlap on projects. All industry developers had computing degrees and a minimum of six years' professional development experience. The projects were all Web applications completed in Java, developed as part of the team's normal work domain, and completed in three to 12 months each.

Companies are rarely willing to commit two teams to develop the same system just to see which approach works better. So, to make things fair, we interleaved the approaches and mixed up their order in completing individual projects. The first quasi-experiment involved a test-last project with no automated tests, followed by a second phase of the same project completed with a test-first approach. The test-first project used the Spring framework. We labeled this comparison INT-TF

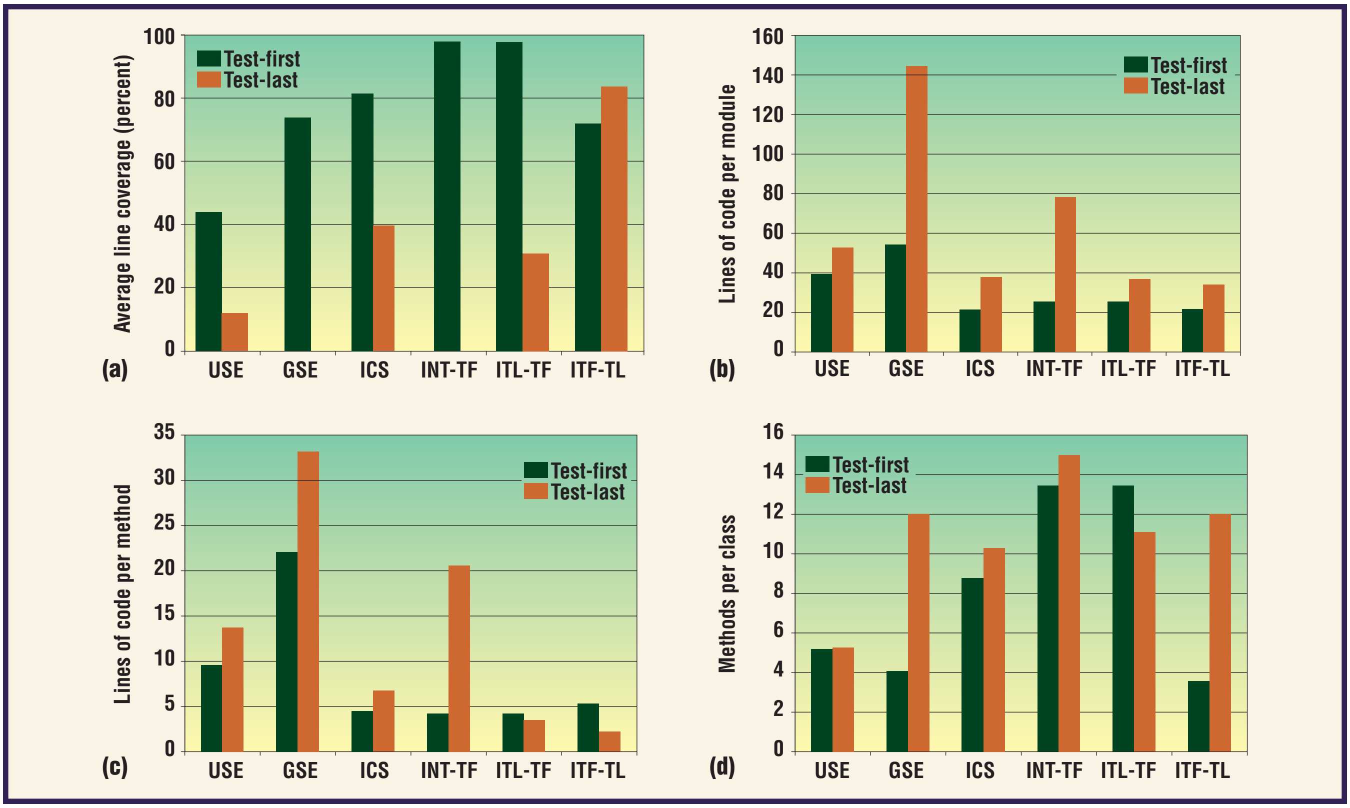


Figure 2. Code size metrics: (a) average line coverage of automated tests, (b) lines of code per module (class), (c) lines of code per method, and (d) methods per class.

for "industry no-tests followed by test-first." The second quasi-experiment involved a test-last project followed by a test-first project. Again, the test-first application used the Spring framework; we labeled this comparison ITL-TF for "industry test-last followed by test-first." The third quasi-experiment involved a test-first project followed by a test-last project. Both projects used the Struts and Spring frameworks along with object-relational mapping patterns and extensive mock objects in testing; we labeled this comparison ITF-TL for "industry test-first followed by test-last."

The case study, labeled ICS, examined 15 software projects completed in one development group over five years. The 15 projects included the five test-first and test-last projects from the industry quasi-experiments. The group had completed the remaining 10 projects prior to the quasi-experiment projects. We interviewed the developers from these 10 projects and determined that all 10 used a test-last approach. All 15 case study projects were completed in three to 12 months with less than 10,000 lines of code by development teams of three or fewer primary developers. Six projects were completed with no automated unit tests; six projects, with automated tests in a test-last manner; and three projects, with automated tests in a test-first manner. All projects used Java to develop Web applications in a single domain.

We labeled the academic studies GSE for "graduate software engineering" and USE for "undergraduate software engineering." We divided the student programmers into test-first and test-last groups and gave them the same set of programming requirements for the semester-long project—specifically, to design and build an HTML pretty-print system. The system was to take an HTML file as input and transform the file into a more human-readable format by performing operations such as deleting redundant tags and adding appropriate indentation.

Students self-selected their teammates, and we compared the results from pre-experiment surveys to ensure that no statistically significant differences existed between the teams in preparation or bias. In particular, we established Java experience as a blocking variable to ensure that each team had a minimum and balanced skill set. In every case, the teams were fairly balanced and didn't change during the study. All but one student in the GSE study had at least one year of professional development experience. Students in the USE study were all juniors or seniors.

We developed TDD and automated testing training materials and delivered them in conjunction with each study. We gave the training to the industry participants in summer 2004. The test-first and test-last projects began in fall 2004 and ran through spring 2006. Although the developers might have experienced some learning curve with TDD during the first few months, we believe the project durations and total time elapsed established sufficient experience in the test-first projects.

The software engineering courses involved relatively short training sessions (about two hours) dedicated to automated unit testing and TDD topics. Some students noted challenges with applying TDD at first. We observed undergraduate students in a lab setting and provided additional informal instruction as needed to keep them writing automated tests. The industry training consisted of a full-day course on automated unit testing and TDD. We carefully presented the materials for both test-first and test-last approaches to avoid introducing any approach bias.

Analyzing the studies

We used several popular software metrics to evaluate the artifacts from the study. Although experts differ regarding the most appropriate metrics, particularly in areas such as coupling[9] and cohesion,[10] we selected a representative set that are widely calculated and reported.

We began our analysis by considering whether the programmers in our studies actually wrote automated unit tests. We informally monitored developers during the studies through brief interviews and observed code samples. The post-experiment survey asked developers to anonymously report whether they used the prescribed approach. In all the studies but one, programmers reported using the approach they were instructed to use (test-first or test-last). The one exception was a team in the undergraduate software engineering course. Despite being instructed to use a test-first approach, the team reported using a test-last approach, so we reclassified them into the test-last control group.

Figure 2a reports each study's average line coverage. This measure indicates the percentage of lines of code that the automated test suites execute. Not surprisingly, line coverage is rather low in the student studies and some test-last teams failed to write any automated tests. In their post-survey comments, several student test-last team members reported running out of time to write tests. In contrast, professional test-last developers in the ICS, ITL-TF, and ITF-TL studies reported more faithful adherence to the rapid-cycle "code-test-refactor" practice.

In every study but the last one, the test-first programmers wrote tests that covered a higher percentage of code. The test-last control group in the INT-TF study performed only manual testing, so the group had no line coverage. In the case study, we omitted test-last projects with no automated tests from the line-coverage percentage calculation to avoid unfairly penalizing the test-last measures. In all the studies, we found additional testing metrics such as branch coverage and number of assertions to be generally consistent with the line-coverage results.

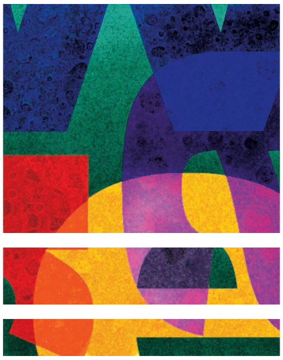
In the final study, when the same professional developers completed a test-last project after having completed a test-first project earlier, they increased their average line coverage. Average branch test coverage (Boolean expressions in control structures) was actually a bit lower at 74 percent for test-last while the test-first project achieved 94 percent. We observed a similar phenomenon in a separate study with beginning programmers.[11] In that study, student programmers who used the test-first approach first wrote more tests than their test-last counterparts. However, on the subsequent project, when students were asked to use the opposite approach, the test-last programmers (those who used test-first on the first project) again wrote more tests. Could it be that the test-first approach has some sort of a residual effect on a programmer's disposition to write more tests? If so, we wonder whether this effect would diminish over time.

Impact on code size

The simplest software metric is size. Figure 2b reports lines of code per module (generally a class). In all studies, test-first programmers wrote smaller modules than their test-last counterparts. The case study was the only study with enough classes to analyze the data statistically. A two-sample, two-tailed, unequal variance t-test indicated that the difference in ICS lines of code per module was statistically significant with $p < 0.05$. Unless stated otherwise, we use this same test and criteria when claiming statistical significance.

Similarly, test-first programmers tended to write smaller methods on average. Figure 2c reveals that test-first programmers' average method size in lines of code was below the test-last averages in all but the last two industry studies (ITL-TF and ITF-TL). The use of simple one-line accessor methods affects these differences. The ITF-TL study had the most striking difference with nearly 40 percent of the methods in the test-last project being simple one-line accessors. In contrast, only 11 percent of the test-first methods were simple accessors. Inlining the one-line accessor methods strengthens the claim that test-first programmers write smaller methods on average.

Finally, figure 2d indicates that the test-first



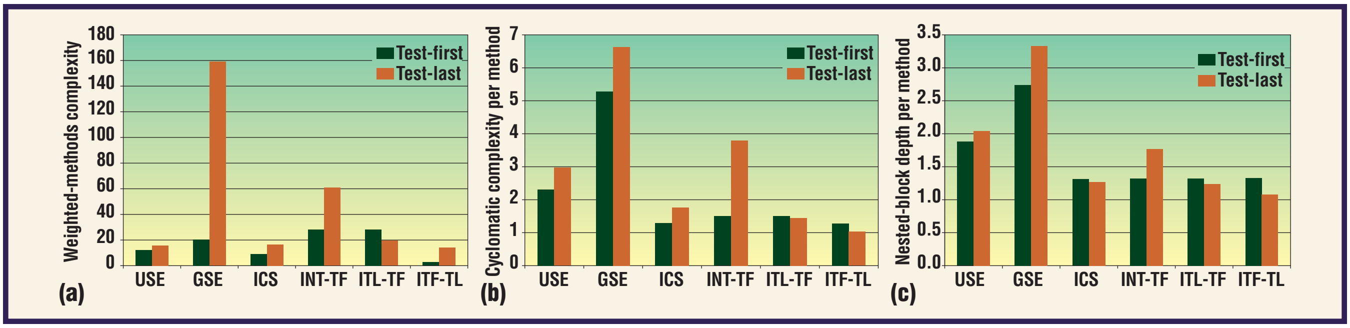


Figure 3. Complexity metrics: (a) weighted-methods complexity, (b) cyclomatic complexity per method, and (c) nested block depth per method.

programmers wrote fewer methods per class in all but the ITL-TF study (the difference was very slight in the USE study).

In summary, the data shows a possible tendency for test-first programmers to write smaller, simpler classes and methods.

Impact on complexity

Size is one measure of complexity: smaller classes and methods are generally simpler and easier to understand. Other common complexity measures include counting the number of independent paths through code (cyclomatic complexity) and measuring the degree of nesting (nested block depth). More branches, paths, and nesting make code more complex and therefore more difficult to understand, test, and maintain.

We report three metrics to compare the complexity differences between the test-first and test-last projects. *Weighted-methods complexity* measures the sum of cyclomatic complexities for all methods in a class. In figure 3a, we see that test-first programmers consistently produced classes with lower complexity in terms of the number of branches and the number of methods. The ICS and ITF-TL differences were statistically significant. The consistently simpler classes by test-first programmers isn't surprising considering the earlier report of fewer methods per class.

The remaining two metrics, *cyclomatic complexity* per method and *nested block depth* (NBD) per method, measure whether individual methods are more or less complex. Figures 3b compares cyclomatic complexity per method, and figure 3c compares NBD per method. The method-level differences are less consistent than those at the class-level. Cyclomatic complexity per method was lower in the test-first projects in four of the six studies. The difference was statistically significant in ICS and INT-TF. In the two studies where the test-last methods were less complex, the difference was small and the method complexity was low for both test-first and test-last methods. The difference in the ITF-TL study was statistically significant, but we question the difference, given the earlier discussion on accessor methods in this study.

NBD comparisons were similar. The test-first projects had lower NBD in three studies. In the remaining three studies, the test-last projects had lower NBD, but the values are low and the differences are small.

We think the complexity metrics point to a tendency of test-first programmers to write simpler classes and sometimes simpler methods.

Impact on coupling

The tendency of test-first programmers to implement solutions with more and smaller classes and methods might generate more connections between classes. Figure 4a shows the *coupling between objects* (CBO), which measures the number of connections between objects. Half the studies had a lower CBO in the test-first projects, and half were lower in the test-last projects. The average CBO values were acceptable in all the studies; none of the differences were statistically significant. The maximum CBO for any class was acceptably low (12 or fewer) for all the projects except two test-last ICS projects (CBO of 28 and 49) and the two projects in the ITF-TL study. Interestingly, the test-first project in the ITF-TL study had a class with a CBO of 26 and the test-last project had a class with a CBO of 16, both of which might be considered unacceptably high.

Figure 4b reports differences in another coupling measure: fan-out per class. Fan-out refers to the number of classes used by a class. Not surprisingly, the results are similar to those for CBO. The differences are small—not statistically significant—and the values are acceptable.

Two additional metrics seem informative when considering coupling: the average number of method parameters (PAR) and the information flow (IF = fan-in^2 * fan-out^2), where fan-in refers to the number of classes using a particular class. In all but the GSE study, PAR was higher in the test-first projects. This difference was statistically significant in all the industry studies. In all but the ITL-TF study, IF was higher in the test-first projects.

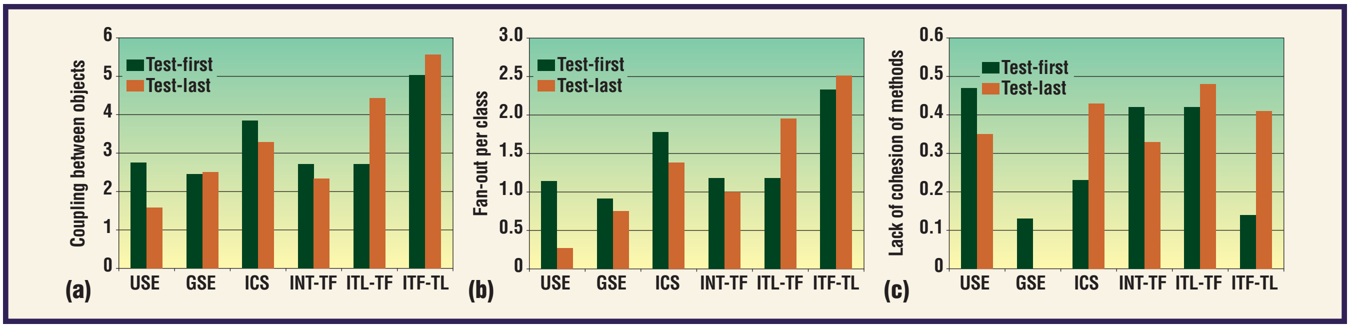


Figure 4. Coupling and cohesion between objects per project: (a) coupling between objects per project, (b) fan-out per class, and (c) lack of cohesion of methods.

The PAR and IF measures indicate a high volume of interaction and data passing between units in the test-first projects. This could reflect the increased testing discussed earlier. Test-first developers often report writing more parameters to make a method easier to configure and test. The higher IF values in the test-first projects might indicate high reuse (fan-in).

We were curious about whether the possible increased coupling was good or bad. Coupling can be bad when it's rigid and changes in one module cause changes in another module. However, some coupling can be good, particularly when it's configurable or uses abstract connections such as interfaces or abstract classes. Such code can be highly flexible and thus more maintainable and reusable.

Many test-first programmers make heavy use of interfaces and abstract classes to simplify testing. For instance, the dependency-injection pattern[12] is popular among TDD developers, and it's central to frameworks such as Spring,[13] which several projects in our study used. To check this out, we looked at several abstraction metrics, including Robert Martin's abstractness measure (RMA),[1] number of interfaces implemented (NII), number of interfaces (NOI), and number of abstract classes (NOA) in all the projects. Our evaluation of these measures didn't give a conclusive answer to whether the test-first approach produces more abstract designs. However in most of the studies, the test-first approach resulted in more abstract projects in terms of RMA, NOI, and NII.

The coupling analysis doesn't reveal clear answers. It appears that test-first programmers might actually tend to write more highly coupled smaller units. However, possible increases in abstractness might indicate that the higher coupling is a good kind of coupling, resulting in more flexible software. The coupling question needs more work.

Impact on cohesion

Cohesion is difficult to measure. The most common metrics look at the sharing (or use) of attributes among methods. We elected to use Brian Henderson-Sellers' definition of *lack of cohesion of methods*, LCOM5,[14] because it normalizes cohesion values between zero and one. In addition, several popular tools calculate LCOM5.

Figure 4c reports LCOM5 measures for the studies. LCOM is an inverse metric, so lower values indicate better cohesion. The chart indicates that cohesion was better in the test-first projects in half the studies (ICS, ITL-TF, and ITF-TL) and worse in the other half. The difference was statistically significant in only two studies (ICS and ITF-TL).

One known problem with most cohesion metrics is their failure to account for accessor methods.[10] Most cohesion metrics, including LCOM5, penalize classes that use accessor methods. The use of accessor methods is common in Java software, and all the study projects involved Java.

To gauge the impact of this concern, we calculated the percentage of accessor to total methods in all but the ICS studies. The test-first projects had an average of 10 percent more accessors in all but the ITF-TL study. It seems plausible that correcting for the accessor problem would bring the test-first cohesion metrics in line with the test-last measures. We were nevertheless unable to substantiate claims that TDD improves cohesion.

Threats to validity

Like most empirical studies, the validity of our results is subject to several threats. In particular, the results are based on a small number of developers. Team selection wasn't randomized, and participants knew that we were comparing TDD and non-TDD approaches, leading to a possible Hawthorne effect. Furthermore, in the industry experiments, it was nearly impossible to control all variables except the use or non-use of TDD, while keeping the projects real and in-domain.

We made every effort to ensure that the TDD and non-TDD teams were applying the approach assigned to them. We interviewed developers during the projects and at their conclusion. We observed the undergraduate academic teams in a lab setting and examined multiple in-process code samples to see that automated unit tests were written in step with production code. Still, developers could have misapplied the TDD and non-TDD approaches at some points. We look forward to additional studies in varied domains that will increase the results' validity and broaden their applicability.

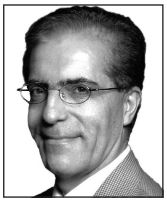
By focusing on how TDD influences design characteristics, we hope to raise awareness of TDD as a design approach and assist others in decisions on whether and how to adopt TDD. Our results indicate that test-first programmers are more likely to write software in more and smaller units that are less complex and more highly tested. We weren't able to confirm claims that TDD improves cohesion while lowering coupling, but we anticipate ways to clarify the questions these design characteristics raised. In particular, we're working to eliminate the confounding factor of accessor usage in the cohesion metrics.

References

1. R.C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Pearson Education, 2003.
2. K. Beck, "Aim, Fire," *IEEE Software*, Sept./Oct. 2001, pp. 87–89.
3. D. Janzen and H. Saiedian, "Test-Driven Development: Concepts, Taxonomy, and Future Direction," *Computer*, Sept. 2005, pp. 43–50.
4. B. George and L. Williams, "A Structured Experiment of Test-Driven Development," *Information and Software Technology*, vol. 46, no. 5, 2004, pp. 337–342.
5. H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of the Test-First Approach to Programming," *IEEE Trans. Software Eng.*, vol. 31, no. 3, 2005, pp. 226–237.
6. A. Geras, M. Smith, and J. Miller, "A Prototype Empirical Evaluation of Test Driven Development," *Proc. 10th Int'l Symp. Software Metrics* (Metrics 04), IEEE CS Press, 2004, pp. 405–416.
7. M.M. Müller, "The Effect of Test-Driven Development on Program Code," *Proc. Int'l Conf. Extreme Programming and Agile Processes in Software Eng.* (XP 06), Springer, 2006, pp. 94–103.
8. P. Kruchten, *The Rational Unified Process: An Introduction*, 3rd ed., Addison-Wesley, 2003.
9. L.C. Briand, J.W. Daly, and K. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Trans. Software Eng.*, vol. 25, no. 1, 1999, pp. 91–121.
10. L. Briand, J. Daly, and J. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.*, vol. 3, no. 1, 1998, pp. 65–117.
11. D. Janzen and H. Saiedian, "A Leveled Examination of Test-Driven Development Acceptance," *Proc. 29th Int'l Conf. Software Eng.* (ICSE 07), IEEE CS Press, 2007, pp. 719–722.
12. M. Fowler, "Inversion of Control Containers and the Dependency Injection Patterns," 2004; www.martinfowler.com/articles/injection.html.
13. R. Johnson et al., *Java Development with the Spring Framework*, Wrox, 2005.
14. B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall, 1996.

About the Authors

David Janzen is an assistant professor of computer science at California Polytechnic State University, San Luis Obispo, and president of Simex, a software consulting and training company. His teaching and research interests include agile methodologies and practices, empirical software engineering, software architecture, and software metrics. He received his PhD in computer science from the University of Kansas and is a member of the IEEE Computer Society and the ACM. Contact him at California Polytechnic State University, Computer Science Department, San Luis Obispo, California, 93407, djanzen@calpoly.edu or david@simexusa.com.



Hossein Saiedian is a professor of software engineering in the Department of Electrical Engineering and Computer Science at the University of Kansas and a member of the university's Information and Telecommunication Technology Center. His research interests are in software engineering—particularly, technical and managerial models for quality software development. He received his PhD in computer science from Kansas State University. He's a senior member of the IEEE. Contact him at EECS, University of Kansas, Lawrence, KS 66049, saiedian@eecs.ku.edu.