On the Influence of Test-Driven Development on Software Design

Agenda

- Introduction
- Overview of paper
- Experimental design
- Results
- Related and future work
- Conclusion
- References
What is Test-Driven Development (TDD)?

- A development technique where you must first write a test that fails before you write new functional code
  1. Write just enough code to fail
  2. Run the test to ensure the new test does in fact fail
  3. Update functional code to make it pass the new test
  4. Run test again
     a) If fail, go back to step 3
     b) If pass; repeat for next function

- Requires programmer to create automated unit tests that define code requirements before writing the code itself
Comparison of TDD against Non-TDD

Source: On the Effectiveness of the Test-First Approach to Programming (Erdogmus 2005)
TDD and Agile

- Agile methods reject comprehensive design phase preceding programming in favor of a small architectural sketch followed quickly by programming
  - Software design and architecture allowed to ‘emerge’
- TDD is seen as essential strategy in such a process
  - “forces” the programmer to carefully consider design decisions such as software interface and behavior before committing to code
  - Design can be cleaner and clearer than is often achieved by other methods [Beck]
- “Test-first code tends to be more cohesive and less coupled than code in which testing isn’t a part of the intimate coding cycle” (Beck, 2003)
TDD and Software Quality

- Research have mostly concentrated on external quality rather than internal quality
  - External quality characteristics are those parts of a product that face its users. E.g. defects
  - Internal quality characteristics are those that do not. E.g. coupling, cohesion, code complexity
- George and Williams have presented several studies that conclude that TDD improves external quality of software in industry (George, 2003)
- Muller has presented studies on the effects of TDD on internal quality by using open-source projects as data source, concluding that TDD has no effect on internal quality (Müller, 2006).
Overview of paper

- An empirical study on:
  - the effects of TDD on programmer productivity and internal software design quality
  - examines effects of test coverage on software quality when using TDD
  - Surveys programmer perceptions regarding TDD
- Based on findings from a controlled experiment conducted in an academic setting
Experimental design - Objectives

- To examine relationship of TDD with software quality and programmer productivity
- To demonstrate feasibility of using TDD with traditional development methods (i.e. not Agile)
Experimental design - Participants

- Conducted with upper-level undergraduate students in a software engineering course
- Junior and senior level students who had completed at least two previous programming courses
- All students were taught:
  - simplified Unified Process,
  - automated testing using JUnit framework
  - how to write software in Test-First and Test-Last manner
- Students were divided into 3 project groups:
  - Two groups to use a Test-First approach, one group to use Test-Last.
  - Teams were self-selected but each team must have at least one member with previous Java experience.
Experimental design – Task

- **Task:**
  - Design and build an HTML pretty print system
    - Take an HTML file as input and transform file into human readable format
  - Project divided into two iterations
    - Iteration 1: text-base UI and partial set of features
    - Iteration 2: GUI and additional features
## Experimental Design: Hypotheses

<table>
<thead>
<tr>
<th>Name</th>
<th>Null Hypothesis</th>
<th>Alternative Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>$\text{Prod}<em>{TF} = \text{Prod}</em>{TL}$</td>
<td>$\text{Prod}<em>{TF} &gt; \text{Prod}</em>{TL}$</td>
</tr>
<tr>
<td>T1</td>
<td>$#\text{Tests}<em>{TF} = #\text{Tests}</em>{TL}$</td>
<td>$#\text{Tests}<em>{TF} &gt; #\text{Tests}</em>{TL}$</td>
</tr>
<tr>
<td>T2</td>
<td>$#\text{TestCov}<em>{TF} = #\text{TestCov}</em>{TL}$</td>
<td>$#\text{TestCov}<em>{TF} &gt; #\text{TestCov}</em>{TL}$</td>
</tr>
<tr>
<td>Q1</td>
<td>$\text{IntQlty}<em>{TF} = \text{IntQlty}</em>{TL}$</td>
<td>$\text{IntQlty}<em>{TF} &gt; \text{IntQlty}</em>{TL}$</td>
</tr>
<tr>
<td>Q2</td>
<td>$\text{IntQlty</td>
<td>Tested}_{TF} = \text{IntQlty</td>
</tr>
<tr>
<td>O1</td>
<td>$\text{Op}<em>{TF} = \text{Op}</em>{TL}$</td>
<td>$\text{Op}<em>{TF} &gt; \text{Op}</em>{TL}$</td>
</tr>
<tr>
<td>O2</td>
<td>$\text{Op</td>
<td>TF}_{TF} = \text{Op</td>
</tr>
</tbody>
</table>

*Table 1. Formalized Hypotheses*
Of the three groups:

- **Test-First:**
  - Only one of the two Test-First teams actually complied to TDD approach.
  - Completed a GUI and implemented about twice as many features as the other teams.
  - Similar number of defects.

- **Test-Last:**
  - The other Test-First team actually wrote automated tests AFTER writing functional code.

- **No-Tests:**
  - Remaining team (originally Test-Last) did not write any automated tests.
Data Analysis - Productivity

![Table 2. Effort in minutes](image)

- Test-First spent least amount of development effort yet delivered the most functionality

**Definitions:**

- *Total effort*: time spent on all activities, including meetings & research
- *Dev effort*: analysis, design, code, test, fix, review
Data Analysis – Code Size

<table>
<thead>
<tr>
<th>Team</th>
<th>Classes</th>
<th>LOC</th>
<th>Test LOC</th>
<th>LOC/method</th>
<th>LOC/feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test-First</td>
<td>13</td>
<td>1053</td>
<td>168</td>
<td>12.10</td>
<td>87.75</td>
</tr>
<tr>
<td>Test-First(no GUI)</td>
<td>11</td>
<td>670</td>
<td>168</td>
<td>11.75</td>
<td>55.83</td>
</tr>
<tr>
<td>No-Tests</td>
<td>7</td>
<td>995</td>
<td>0</td>
<td>27.64</td>
<td>199.00</td>
</tr>
<tr>
<td>Test-Last</td>
<td>4</td>
<td>259</td>
<td>38</td>
<td>7.40</td>
<td>43.17</td>
</tr>
</tbody>
</table>

Table 3. Code Size Metrics

- Test-First implemented more code (including test and GUI) than other two
- Test-First and Test-Last have a reasonable average method size and LOC/feature
- No-Tests wrote long methods and excessive amount of code for the provided functionality
Data Analysis – Test Density

<table>
<thead>
<tr>
<th>Team</th>
<th>Assertions/SLOC</th>
<th>Test Coverage (lines)</th>
<th>Test Coverage (branches)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test-First</td>
<td>0.077</td>
<td>19.00%</td>
<td>39.00%</td>
</tr>
<tr>
<td>Test-First (less GUI)</td>
<td>0.086</td>
<td>31.00%</td>
<td>43.00%</td>
</tr>
<tr>
<td>No-Tests</td>
<td>0.000</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Test-Last</td>
<td>0.045</td>
<td>29.00%</td>
<td>23.00%</td>
</tr>
</tbody>
</table>

*Table 4. Test Density and Coverage Metrics*

- Calculated using STREW Eclipse plug-in
- Test-First wrote twice as many assertions/SLOC as Test-Last
- Statement coverage between Test-First and Test-Last not significantly different, but
- Branch coverage of Test-First is 86% higher than Test-Last
Data Analysis – Internal Quality

Table 5. Internal Quality Metrics with Warnings

<table>
<thead>
<tr>
<th>Team</th>
<th>Nested Block Depth</th>
<th>Cyclomatic Complexity</th>
<th>Parameters</th>
<th>CBO</th>
<th>IF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td>Test-First</td>
<td>2.02</td>
<td>6</td>
<td>2.33</td>
<td>13</td>
<td>0.62</td>
</tr>
<tr>
<td>Test-First (no GUI)</td>
<td>1.85</td>
<td>6</td>
<td>2.59</td>
<td>13</td>
<td>0.89</td>
</tr>
<tr>
<td>No-Tests</td>
<td>3.00</td>
<td>6</td>
<td>6.53</td>
<td>27</td>
<td>1.08</td>
</tr>
<tr>
<td>Test-Last</td>
<td>1.20</td>
<td>3</td>
<td>1.46</td>
<td>4</td>
<td>0.57</td>
</tr>
</tbody>
</table>

Definitions:

- **Nested Block Depth** = the depth of nested blocks of code
- **Cyclomatic Complexity** = number of flows through an individual method
- **Parameters** = total number of parameters in selected scope
- **CBO** = Coupling Between Objects = the number of connections between objects
- **IF** = Information Flow = measures the interactions between subroutines of the system
Internal Quality (cont.)

- Gathered twenty-five structural and object-oriented metrics using freely available tools.
- No-Tests and Test-Last wrote code that is procedural in nature
  - No-Tests had classes with verb names and many long, complex loops.
  - Test-Last wrote all functionality in three classes besides `main()`, and `main()` contained primary program logic.
- Test-Last produced better “scores” than Test-First!
  - Test-Last used java.util.regex library to achieve more functionality with less code.
- GUI implementation negatively affects measurement as GUI code was not covered by any automated unit tests.
Data Analysis – Test-First Micro-evaluation

Table 6. Metrics on Tested and Untested Code of Test-First Project

<table>
<thead>
<tr>
<th>Code</th>
<th>WMC</th>
<th>CBO</th>
<th>NBD</th>
<th>Complexity</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>max</td>
<td>avg</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td>Tested</td>
<td>7.80</td>
<td>21</td>
<td>2.2</td>
<td>3</td>
<td>1.77</td>
</tr>
<tr>
<td>Untested</td>
<td>13.55</td>
<td>53</td>
<td>4.5</td>
<td>20</td>
<td>2.53</td>
</tr>
</tbody>
</table>

Definitions:

- **WMC** = *Weighted Methods Complexity* = sum of cyclomatic complexities for all methods in a class
- **CBO** = *Coupling Between Objects* = the number of connections between objects
- **NBD** = *Nested Block Depth* = the depth of nested blocks of code
- **Complexity** = number of flows through an individual method
- **Parameters** = total number of parameters in selected scope
Test-First Micro-evaluation (cont.)

- Measures quality of code covered by automated unit tests compared to code not covered by any tests
- Tested methods 43% lower in average complexity
- Tested classes had 104% lower coupling measures
- Lower quality of untested code might be due to either:
  1. Absence of tests which leads to poor quality, OR
  2. Indicates programmer spent less effort on untested segments of code
- Lack of test coverage might be indicator of internal quality issues
Data Analysis – Programmer Perception

<table>
<thead>
<tr>
<th>Team</th>
<th>Test-First</th>
<th>Test-Last</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pre</td>
<td>Post</td>
</tr>
<tr>
<td>Test-First</td>
<td>3.67</td>
<td>4</td>
</tr>
<tr>
<td>No-Tests</td>
<td>1.5</td>
<td>2</td>
</tr>
<tr>
<td>Test-Last</td>
<td>2.33</td>
<td>3.25</td>
</tr>
</tbody>
</table>

*Table 7. Programmer Perception of Test-First and Test-Last (0 to 4 scale)*

- Questionnaire administered pre and post-experiment
- All three teams perceive Test-First more positively after the experiment, and inversely perceived Test-Last more negatively
  - 89% thought Test-First produced simpler design
  - 70% thought Test-First produced code with fewer defects
  - 75% thought Test-First was best approach
Data Analysis – Programmer Perception (cont.)

- **Test-First team:**
  - 100% of team preferred to use Test-First over Test-Last in future projects
  - Reported higher confidence in ability to make future changes to their software

- **Test-Last team:** 50:50 split among *Test-Last* team

- **No-Test team:** 100% of team preferred to use Test-Last again
  - More comfortable with approach that they already know
Threats to Validity (Caveats)

- Experiment was conducted with only 10 students
- A single programmer on each of the three teams implemented majority of the core functionality
- Differences in quality and productivity could be attributed to individual skill levels of only three programmers
Conclusion

- Test-First approach may have positive correlation with programmer productivity
- Test-First was not shown to produce better quality software than Test-Last
- Internal quality issues may arise when Test-First process breaks down and tests are not written
- Programmers perceive TDD more positively after exposure to it, and more likely to adopt TDD after trying it
Related Work

- For comparison studies of TDD in industry:
  - George and Williams presented a study of 24 XP programmers, and reported quality improvement but with decline in productivity (George, 2004).
  - Nagappan et al presented a similar study conducted at Microsoft and IBM, concluding that TDD led to improved quality but increase in development time (Nagappan, 2008).

- For studies on adoption of TDD in industry:
  - Maximiliean and Williams present an experience paper on TDD adoption at IBM (Maximiliean, 2003).
  - Kaufman and Janzen presented an experience paper on TDD adoption at Sun Microsystems (Kaufman 2003).
  - Rendell proposed a pragmatic approach to TDD adoption based on his experience at T-Mobile (Rendell, 2008).
For comparison studies of TDD in academia:

- Edwards presented a comparison study of external quality improvement with 59 undergraduate students and concludes that software produced by students using TDD had less defects. (Edwards, 2002)
- Müller and Hagner conducted a similar study with CS graduate students, but using XP. Study found that programmer productivity did not improve, and program reliability only increased slightly. (Müller, 2002)
Future Work

- Study on whether TDD with proper test coverage can reduce complexity and coupling
References

References


