From Details to Done

A Test-Driven Approach to Software Development

Steve Jewett
Systems & Software Technology Conference 2011
## Report Documentation Page

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

| 1. REPORT DATE | MAY 2011 |
| 2. REPORT TYPE | |
| 3. DATES COVERED | 00-00-2011 to 00-00-2011 |
| 4. TITLE AND SUBTITLE | From Details to Done. A Test-Driven Approach to Software Development |
| 5a. CONTRACT NUMBER | |
| 5b. GRANT NUMBER | |
| 5c. PROGRAM ELEMENT NUMBER | |
| 5d. PROJECT NUMBER | |
| 5e. TASK NUMBER | |
| 5f. WORK UNIT NUMBER | |
| 6. AUTHOR(S) | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) | Boeing Defense, Space & Security, PO Box 516, St. Louis, MO, 63166 |
| 8. PERFORMING ORGANIZATION REPORT NUMBER | |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | |
| 10. SPONSOR/MONITOR’S ACRONYM(S) | |
| 11. SPONSOR/MONITOR’S REPORT NUMBER(S) | |
| 12. DISTRIBUTION/AVAILABILITY STATEMENT | Approved for public release; distribution unlimited |
| 13. SUPPLEMENTARY NOTES | Presented at the 23rd Systems and Software Technology Conference (SSTC), 16-19 May 2011, Salt Lake City, UT. Sponsored in part by the USAF. U.S. Government or Federal Rights License |
| 14. ABSTRACT | |
| 15. SUBJECT TERMS | |
| 16. SECURITY CLASSIFICATION OF: | |
| a. REPORT | unclassified |
| b. ABSTRACT | unclassified |
| c. THIS PAGE | unclassified |
| 17. LIMITATION OF ABSTRACT | Same as Report (SAR) |
| 18. NUMBER OF PAGES | 25 |
| 19a. NAME OF RESPONSIBLE PERSON | |

---

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18
Topics

- Moving Tests Forward
- 3 Rules of Test-Driven Development (TDD)
- TDD in Unit, Integration and Acceptance Testing
- Comprehensive TDD Process
- Pros and Cons of TDD
- Q&A
Traditional Development Cycle

Testing Follows Implementation:

Unit tests are executed after modules are completed.

Integration testing follows implementation.

Acceptance testing begins at the end of integration.
Testing Occurs Before Implementation:

Acceptance tests are developed as part of the requirements.

Integration tests are developed as part of the design.

Unit tests are developed as part of the implementation.

Test are executed throughout implementation; test failures drive what to do next.
Agile Development is not phase-oriented, so tests are executed throughout the cycle, not just during implementation.
Test-Driven Development (TDD) says to create tests first and let them drive implementation. The three rules of TDD demonstrate how to do that.

**3 Rules of TDD**

1. Write production code only to pass a failing test.
2. Write only enough test code to fail.
3. Write only enough production code to pass.
Unit tests are created by developers to add functionality to a class or module.

At the unit test level the three rules are manifest in the “red-green-refactor” approach:

Red-Green-Refactor

Write a unit test that fails.

Write production code to make the test pass.

Clean up both test and production code.

for example …
Using the Red-Green-Refactor approach, developers create unit tests for individual modules as they add functionality.

External dependencies are handled by creating mock objects.
Initial integration tests are the unit tests with real components replacing mock objects.

Additional integration tests may be needed to address scaling, loading or speed.
Acceptance tests may take the form of use case scenarios executed via a user interface …

… or they may be the integration tests from an external application.
Develop acceptance test scenarios from groups of related features.

Develop integration tests for components of a simple, initial design.

Develop unit tests and components using the red-green-refactor approach and mock objects.

Integrate components by replacing mock objects with actual components and executing unit and integration tests.

Execute acceptance test scenarios to ensure all functionality is complete.
**Benefits of Test-Driven Development**

- **Test-Driven Development**
  - **Testable Designs**
    - Creates inherently testable designs
  - **Complete Test Suite**
    - Creates a test suite that can be retained for regression testing
  - **Reduced Scope Creep**
    - Fights developer-induced scope creep by limiting efforts to what needs to be developed
  - **Lean Code – Simple Designs**
    - Emphasis on writing just enough code drives lean and simple solutions
  - **Definition of Done**
    - Up front test definition provides a concrete “definition of done”
  - **Customer Acceptance Tests**
    - Allows customers to write acceptance level tests without needing to understand technical details
Drawbacks of Test-Driven Development

- **Simple Designs**
  Creates a solution, but not necessarily the best or most efficient solution

- **Exhaustive Testing Not Addressed**
  Difficult and/or inefficient for projects requiring exhaustive testing

- **Paradigm Shift/Learning Curve**
  Can affect productivity due to a lack of necessary skills and experience, as well as resistance to culture change

- **Drop in Perceived Productivity**
  Feature productivity is traded off for stability, quality and maintainability

- **Not a Silver Bullet**
  Bad Requirements → Bad Tests → Bad Software
End
Very Simple TDD Example – Hello World

```java
public class Greeter_Test {
    [TestMethod]
    public void TestDisplayHelloWorld() {
        Greeter myGreeter = new Greeter();
    }
}
```

Compilation Error
<table>
<thead>
<tr>
<th>Greeter_Test</th>
<th>Greeter</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Test]</td>
<td>public class Greeter</td>
</tr>
<tr>
<td>public class Greeter_Test</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>{ public Greeter()</td>
</tr>
<tr>
<td>[TestMethod]</td>
<td></td>
</tr>
<tr>
<td>public void TestDisplayHelloWorld()</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>Greeter myGreeter = new Greeter()</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
public class Greeter
{
    Greeter()
}

[Test]
public class Greeter_Test
{
    [TestMethod]
    public void TestDisplayHelloWorld()
    {
        Greeter myGreeter = new Greeter()
        Assert(myGreeter.getGreeting(), "Hello World")
    }
}
Very Simple TDD Example (cont’d)

```
[Test]
public class Greeter_Test
{
    [TestMethod]
    public void TestDisplayHelloWorld()
    {
        Greeter myGreeter = new Greeter()
        Assert(myGreeter.getGreeting(), "Hello World")
    }
}
```

```
public class Greeter
{
    Greeter()
    
    String getGreeting()
    {
        return ""
    }
}
```

Test Failure
### Very Simple TDD Example (cont'd)

#### Greeter

```java
class Greeter {
    Greeter()
    String getGreeting() {
        return "Hello World"
    }
}
```

#### Greeter_Test

```java
[Test]
public class Greeter_Test {
    [TestMethod]
    public void TestDisplayHelloWorld() {
        Greeter myGreeter = new Greeter();
        Assert(myGreeter.getGreeting(), "Hello World")
    }
}
```
### Greeter

```java
public class Greeter {
    const String greeting = "Hello World"
}
```

### Greeter_Test

```java
[Test]
public class Greeter_Test {
    [TestMethod]
    public void TestDisplayHelloWorld() {
        Greeter greeter = new Greeter();
        Assert(greeter.getGreeting(), "Hello World"),
    }
}
```
public class Greeter
{
    const String greeting = "Hello World"
    Greeter()
    String getGreeting()
    {
        return greeting
    }
}

[Test]
public class Greeter_Test
{
    const String expectedGreeting = "Hello World"

    [TestMethod]
    public void TestDisplayHelloWorld()
    {
        Greeter myGreeter = new Greeter()
        Assert(myGreeter.getGreeting(), expectedGreeting)
    }
}
public class Greeter {
    const String greeting = "Hello World"

    private Greeter() {
        static Greeter GetInstance() {
            return new Greeter();
        }
    }

    String getGreeting() {
        return greeting;
    }
}

[Test]
public class Greeter_Test {
    const String expectedGreeting = "Hello World"

    [TestMethod]
    public void TestDisplayHelloWorld() {
        Greeter myGreeter = new Greeter();
        Assert(myGreeter.getGreeting(), expectedGreeting);
    }
}
public class Greeter{
    const String greeting = "Hello World"
    private Greeter()
    static Greeter GetInstance()
    {
        return new Greeter()
    }
    String getGreeting()
    {
        return greeting
    }
}

[Test]
public class Greeter_Test
{
    const String expectedGreeting = "Hello World"

    [TestMethod]
    public void TestDisplayHelloWorld()
    {
        Greeter myGreeter = Greeter.GetInstance()
        Assert(myGreeter.getGreeting(), expectedGreeting)
    }
}
public class Greeter {
    const String greeting = "Hello World"

    private Greeter() {
        static Greeter GetInstance() {
            return new Greeter()
        }
    }

    String getGreeting() {
        return greeting
    }
}

[Test]
public class Greeter_Test {
    const String expectedGreeting = "Hello World"

    [TestMethod]
    public void TestDisplayHelloWorld() {
        Assert(Greeter.GetInstance().getGreeting, expectedGreeting)
    }
}

Greeter

public class Greeter {
    const String greeting = "Hello World"

    private Greeter() {
    }

    static Greeter GetInstance() {
        return new Greeter()
    }

    String getGreeting() {
        return greeting
    }
}