

MATLAB[®] and TDD

Richard Johnson

Test-Driven Development (TDD) is a key element of Extreme Programming and a very useful technique in other software development processes. This paper is an introduction to the use of TDD for MATLAB[®], a topic that has been little discussed. There is a lot more to be said about this topic, and much of it is covered in the Extreme MATLAB Workshop.

Test-Driven Development (TDD) is an agile software method that emphasizes iterative coding and constant testability. According to Kent Beck, there are two basic rules in practicing TDD:

- Write new code only if an automated test has failed.
- Eliminate duplication.

Following these rules, the code development steps are:

- Write a little test that doesn't work.
- Make the test work quickly.
- Eliminate all duplication in the code.

Then you repeat these steps in order until the tests capture all the requirements.

This approach can seem very strange to developers who have only used traditional methods. The code is not derived from requirements, specifications or design documents. The tests serve as executable specifications. The production code is written in response to the test code. The design of the production code evolves as it is refactored (modified to improve quality). Both test and production code are written in very short iterations.

In the TDD approach, thinking through and writing test code first helps to define the functionality and interface of the production code. Test first development helps assure that the production module is written for testability and that every feature is tested. This practice of writing tests early is likely to reduce the number of defects in production code.

TDD may be easiest to appreciate in the context of a function and its unit tests. Unit tests provide confidence that the function code works, and confidence that it works after changes.

MATLAB projects are often on short or immediate schedules. Requirements are often incomplete and almost never "frozen." Many of these development projects are constrained by developer time. TDD is an effective approach in this situation.

Development steps

Collect user stories to describe what the production code should do.

Write a test.

Write a simple test that is based on a user story and fails. Having the test fail initially increases your confidence that it will not always pass and that it is doing what you think it is.

In development environments with a graphical status display, writing a failing test is called the “red bar” stage.

Make it pass.

Write enough production code that the test passes. Check that it passes all previous tests. Commit to version control.

The hardcore advocates of TDD assert that you should write just enough code to pass the test. The problem with writing more code than is required is that it may be a waste of development time, or it may have a defect that goes untested.

In development environments with a graphical status display, adding or changing code to pass a test is called the “green bar” stage.

Remove duplication.

Eliminate code duplication. A block of code should appear in only one place. You are more likely to get it right and keep it right. If there are similar blocks of code, try to consolidate them into one block.

Work toward a design. Refactor (rewrite and redesign) your code for better testability, clarity and maintainability. Apply your coding standards. Check that it passes all its tests. Commit to version control.

This is called the “refactoring” stage.

You may spot a problem while you are refactoring. Be sure to write a test that will fail for that problem. This practice supports an automated approach to help prevent faults from recurring during development.

You may write code that is more general while you are refactoring. Be sure to write a test that will check that generality. This practice supports an automated approach to help prevent faults appearing in deployment.

Look at the test code for this module. If there is duplication, consider refactoring the test code. Refactoring can make the test code more maintainable, but don't do it if the change

makes the test code less clear. If you do refactor, check that all tests still pass. Commit any changed code.

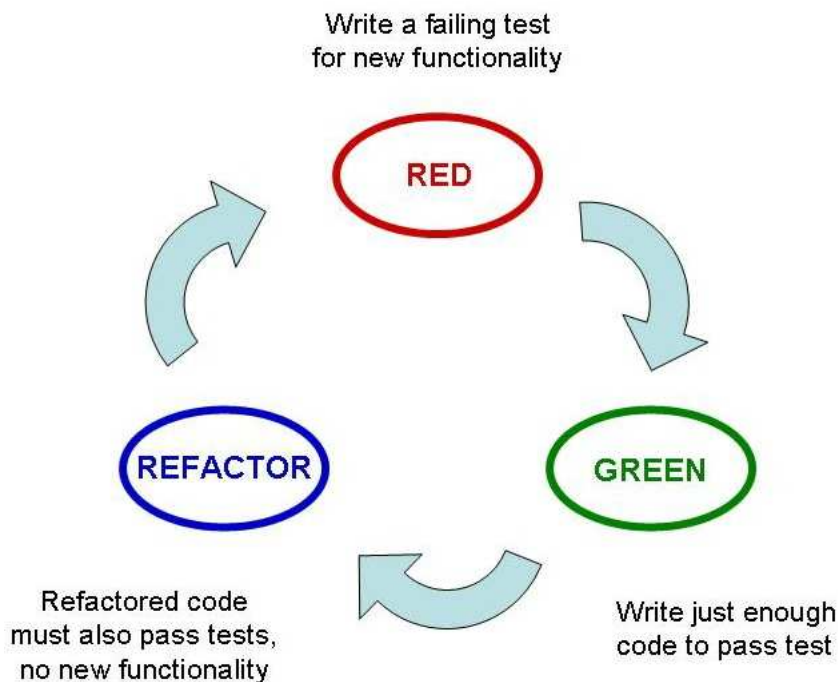
Repeat.

Repeat the cycle until the test suite covers the user stories and other requirements. Your cycle through the three development steps should take less than 30 and preferably less than 10 minutes. Short coding cycles are important for fast and stable feedback.

By making frequent commits to the version control system, you can be confident that you will be able to recover if things go south.

The basic cycle as illustrated by Camille Bell:

The Test Driven Development Cycle



Hints

Make your tests order independent. No test should depend on another test having been run.

Each test should have an assertion that is true or false. Be sure that you can easily see what the assert is testing.

Keep your tests simple. Each unit test should do one thing well. You should not have if or switch constructs in unit test code.

The test should be written based on a required feature. The first time you run a test, it fails because it tests a feature that has not yet been implemented. Any subsequent failure means that the feature is not fully implemented.

Put the unit tests for one function together in one file (a “test harness”). Putting all the unit tests in one file supports a simple test file naming convention and means that you will have fewer test files to manage. Also you can use a simple name for the test file with more descriptive comments for each test. An exception is that tests that depend on some machine dependent configuration should be in a separate file.

Retain tests that pass and keep them passing. When an old test fails there can be two situations: You broke it with a code change or the code is not compatible with this test. If the second is true, is the old test still appropriate? If so you will need to change it. (Perhaps there has been an API change.) If the code has already been in use (the curse of the installed base), it is better to face this issue at the unit test stage than after release.

Consider code coverage. If you can remove a line of code and all tests still pass, you need another test. If you can't write a failing test, why do you need that line of code?

When a test fails unexpectedly it is often more effective to revert to the last version and use that as a code base rather than spend excessive time on debugging.

Additional Practices

You follow many typical software development practices during TDD such as version control, code reviews, thorough testing, and coding standards.

Version control

The professional way to manage and document code changes is to use a version control tool. It is helpful for solo development and almost essential for team development. Version control takes the anxiety out of changing the code.

Change code in small steps, particularly when refactoring. Check in working code early and often. Also include your test software in version control.

MATLAB currently supports direct integration with only a few version control applications. You may need to find a third party interface or use a manual process.

Code reviews

Discuss your code. Code review is a form of testing. If you can't explain the code to another programmer, how likely is it to be correct? How likely will you be to understand it six months later, after you have moved on to another project?

Show someone your code often so that it is relatively easy for them to remember and understand it. In Extreme Programming the practice of pair programming provides a continuous code review.

Coding standards

Coding standards specify common elements for the code that make it easier to share and use code and easier to understand the ideas expressed in the code. They are programming rules that reduce the likelihood of introducing errors into the programs. They go beyond style guidelines to address topics like use and behavior. Examples include specific variable names, input and output formats, and processing algorithms.

Coding standards are usually organization or site specific. Style guidelines are more broadly applicable. Many of the style practices of other languages are being adopted in MATLAB. The items in this abbreviated style list are guidelines, not rules. There will certainly be cases where not all of them apply. Some practices include:

Modularize.

Write the code in small modules. Short blocks of code tend to be easier to read and understand. This organization encourages better code structure and helps to localize the inevitable changes. Code blocks can be visually delineated using blank lines, comments or Editor cell mode.

Write simple functions. Decompose complicated functions into simple ones. Short simple functions are easier to understand, test, and reuse.

Use functions rather than scripts.

Functions avoid cluttering the base workspace with variables. They also present well-defined interfaces. If you want the convenience of scripts for running examples, write a script that calls a function.

Write readable code.

Write code that others will understand and that you will understand in the future. Use a consistent and effective style to aid that understanding. Use the program domain language when it helps.

Provide user help.

Well written test and production code supplies the most important documentation for other developers, but you also need documentation for users. Users need to know the interface and sometimes a description of what the function does or how it works.

Make this information available through the Help Browser. Format header comments in the production code so that they publish to the standard help page format. Make your help pages searchable in the Help Browser.

Include appropriate generality.

Too little flexibility can lead to a lot of work in the future. Too much leads to a lot of work now. Code that will be part of a toolbox or shared with others should typically be more flexible than code with a short lifetime and for your own use. A good program will be used, used in unplanned ways, and probably modified. Consider supporting some flexibility. Your experience will guide you toward the right amount of flexibility.

Appropriate flexibility might include working with nan values or simple arrays rather than just scalars. Unnecessary flexibility might include working with unlikely inputs such as multidimensional arrays and complex numbers.

Write for automation.

Avoid keyboard, input, pause, etc. commands when possible. Use a parameter file instead. This approach is particularly important for automated test.

Support test automation by providing a program that runs all the tests. Automated tests provide continuous or near-continuous feedback on the status of the code.

Write unit tests in a function.

Write tests as functions that have a Boolean output. Consider putting all the unit test code for a production function in a single test function. Its output would be true if all tests pass, false if any does not.

Place the test code in a separate folder that is not on the path. This will prevent the main code from unintentionally becoming dependent on the test code.

Develop test patterns.

In general the test code should require less time to develop than the code to be tested. Yet test code is as difficult to write correctly as project code. The answer is to develop test code patterns. The patterns could for example produce interesting inputs or evaluate expected outputs. When the inputs are common to multiple tests, write setup code to produce them.

Thorough testing may require writing tests that go beyond user stories. Patterns can help to define these tests. Test for coverage of the valid domain. Look for problems with boundary data such as extreme values and empty variables.

Beware of errors in the test code that can produce misleading results. Beware of defects that occur in both your project and test code. You won't be testing for problems that you don't think of. We all have blind spots. This is a role for code review.

Consider using a framework.

Testing frameworks can help to make higher quality tests that support automation. However their use is also controversial. Some people feel that frameworks tend to decrease readability, especially the use of setup and teardown code. Test frameworks are more important for objects than functions.

Helpful MATLAB features

Use the MATLAB integrated development environment features to ease the generation of quality code. These tools can improve code, enforce consistency, and shorten development time. They support iterative development through incremental changes.

There are several MATLAB features that support TDD:

Editor

The MATLAB Editor provides color syntax coding, code execution through Evaluate Cell and Evaluate Selection, and automatic indentation and parentheses matching.

It provides editor cells to clarify blocks of code and support publishing. It also includes M-Lint to help improve code quality and integration with the Debugger for trouble shooting.

Place each block of test code in an editor cell of the test file. This makes it easy to identify individual tests and to execute one at a time.

Debugger

Use the Debugger effectively. Turn on Stop on Error when debugging functions. With this selection the editor opens the file and puts a cursor close to the problem site. The Stop on Nan or Inf selection is also a helpful option for finding the origin of an error.

M-Lint

The M-Lint feature provides prompt feedback on syntax errors and questionable usage. Heed the M-Lint error and warning messages. Catch errors and poor usage quickly and improve the code.

Some of the warning messages can actually reveal errors. Messages like: “The value assigned here to variable 'readOne' might be unused.” can be caused by an inconsistently spelled variable name.

MATLAB Path

Keep your production and test code in separate folders. Put only the production code folder on the MATLAB path. This will make it easier to release the production code and reduce the possibility that it depends on the test code.

Configurable Desktop layout

Find a desktop layout that works for you. Reduce that distraction of window focus changes by arranging your most used windows in the MATLAB desktop. Many programmers find that the most useful desktop includes the Editor, Command Window, Workspace browser and the Directory browser. Many MATLAB demonstrators use this arrangement.

Publish

Publish both production and test functions during development. This will help you to keep the comments helpful and consistent with the code.

Profiler

In general delay optimization for speed as long as you can. The slow parts of your code may not be where you expect. Use the Profiler to find bottlenecks in execution speed. Often scalability and fast performance come not so much from optimization as from design and selection of the right algorithm.

You can also use the profiler to assess code coverage. With TDD the coverage should always be high as a result of the process. Actually it is more important that the tests cover all the required cases than all the lines of code. Covering all of the lines of code may be more trouble than it is worth.

Dependency Report

Check that the production code does not depend on the test code. Check that each unit test function depends on (calls) only one production function.

Testing frameworks

The MathWorks does not provide a testing framework. There are at least four user contributed frameworks available through the File Exchange at MATLAB Central. Search for “unit test.”

When you are new to TDD, you will probably learn more by deferring the use of test frameworks until you have more experience.

Issues

The MATLAB assert function generates an error rather than an exception and has no output argument, so it must be wrapped in a try catch block to achieve full automation.

Refactoring multiple try catch blocks into a subfunction may be the best way to handle this situation.

MATLAB users may expect more general functionality, even if it is not included in the user story. This is a good trigger for conversation.

The current Editor lacks a few features that would help TDD: a flexible version control interface, a replace inside selection capability, a replace across files capability, and support for automated refactoring.

Readings

Relevant websites include

wikipedia.org

See Test-driven development and related articles.

agiledata.org (Scott Ambler)

Scott Ambler is the author of several style guides and the source of several surveys of development practices.

construx.com (Steve McConnell)

Steve McConnell is very commonly cited on coding for professional quality development.

martinfowler.com (Martin Fowler)

Martin Fowler is the author most strongly associated with refactoring.

threeriversinstitute.org (Kent Beck)

Kent Beck is the author most strongly associated with TDD. He is also one of the most visible authors in the field of eXtreme Programming.

Books

Code Complete: A Practical Handbook of Software Construction by Steve McConnell
2004

Elements of MATLAB Style by Richard Johnson 2010

Extreme Programming Examined by Giancarlo Succi and Michele Marchesi 2001

Extreme Programming Explained by Kent Beck and Cynthia Andres 2005

Growing Object-Oriented Software, Guided by Tests by Steve Freeman and Nat Pryce
2010

Questioning Extreme Programming by Pete McBreen 2003

Refactoring: Improving the Design of Existing Code by Martin Fowler and others 1999

Test-Driven Development: A Practical Guide by David Astels 2003

Test-Driven Development: By Example by Kent Beck 2002

Example

Kent provides an example of TDD for a Fibonacci number computation in an appendix of his book. This is a MATLAB example of the same task. The code does not include comment markup to keep the printed blocks short.

The user story is: "I need to get the *i*th Fibonacci number."

A simple version of the initial test function is

```
function test_fibonacci
assert(fibonacci(0) == 0)
```

Running this function gives the result

```
test_fibonacci
??? Undefined function or method 'fibonacci' for input arguments of
type 'double'.
```

which is not surprising since the fibonacci function does not yet exist. Implement

```
function fibonacciNo = fibonacci(n)
fibonacciNo = n;
```

Now the test passes. Extend the test

```
function test_fibonacci
assert(fibonacci(0) == 0)
assert(fibonacci(1) == 1)
```

This test also passes.

The test function has almost duplicated code that will only get worse with more cases, so refactor it to

```
function test_fibonacci
index = [0 1];
expected = [0 1];
for n = 1:length(index)
    assert(fibonacci(index(n)) == expected(n))
end
```

The function also passes this version of the test code. The next test step is

```
function test_fibonacci
index = [0 1 2];
expected = [0 1 1];
for n = 1:length(index)
    assert(fibonacci(index(n)) == expected(n))
end
```

This test fails

```
>> test_fibonacci
???: Error using ==> test_fibonacci at 17
Assertion failed.
```

You could start making special cases for $n > 2$, but this is a good time to look into how the Fibonacci numbers are defined. The function can be modified to implement the standard recurrence relation as

```
function fibonacciNo = fibonacci(n)
if n < 2
    fibonacciNo = n;
else
    fibonacciNo = fibonacci(n-1)+fibonacci(n-2);
end
```

By induction we can say that this function will be correct for all integers greater than or equal to 0. This capability may be enough to satisfy the customer.

But MATLAB is a technical language with sometimes inconvenient users. The case of a negative number

```
function test_fibonacci
index = [0 1 2 -2];
expected = [0 1 1 -1];
for n = 1:length(index)
    assert(fibonacci(index(n)) == expected(n))
end
```

gives the result

```
>> test_fibonacci
???: Maximum recursion limit of 500 reached. Use
set(0,'RecursionLimit',N)
to change the limit. Be aware that exceeding your available stack
space can crash MATLAB and/or your computer.
```

```
Error in ==> fibonacci
```

The function can be modified to

```
function fibonacciNo = fibonacci(n)
if n==0 || n==1
    fibonacciNo = n;
elseif n > 1
    fibonacciNo = fibonacci(n-1)+fibonacci(n-2);
else
    an = abs(n);
    fibonacciNo = (-1)^(an+1)*fibonacci(an);
end
```

which passes. The pesky user might enter NaN or Inf.

```

function test_fibonacci
%%
index = [0 1 2 -2 nan];
expected = [0 1 1 -1 nan];
for n = 1:length(index)
    assert(fibonacci(index(n)) == expected(n))
end

```

This case produces the same recursion limit error. The function can be modified to

```

function fibonacciNo = fibonacci(n)
if ~isfinite(n)
    fibonacciNo = nan;
elseif n==0 || n==1
    fibonacciNo = n;
elseif n>1
    fibonacciNo = fibonacci(n-1)+fibonacci(n-2);
else
    an = abs(n);
    fibonacciNo = (-1)^(an+1)*fibonacci(an);
end

```

but this results in an assert error.

```

>> test_fibonacci
??? Error using ==> test_fibonacci
Assertion failed.

```

The problem is with the test function, since NaN does not equal NaN. The test code can be modified to

```

function test_fibonacci
index = [0 1 2 -2 nan];
expected = [0 1 1 -1 nan];
for n = 1:length(index)
    if isfinite(index(n))
        assert(fibonacci(index(n)) == expected(n))
    else
        assert(isnan(fibonacci(index(n))))
    end
end

```

which will work.

Refinements can continue. The user might enter a non-integer value, which will also generate a recursion problem. The function could be modified to detect this case, but it is not clear what the correct Fibonacci number would be.

The customer might change requirements so that the function would work with an input array in addition to a scalar. The recursion solution would not work in that case without a kluge. The function would have to be rewritten. Fortunately the test function would still be valid, so that any rewrite could be easily checked. For a variety of solutions to various forms of the Fibonacci task, search MATLAB Central.