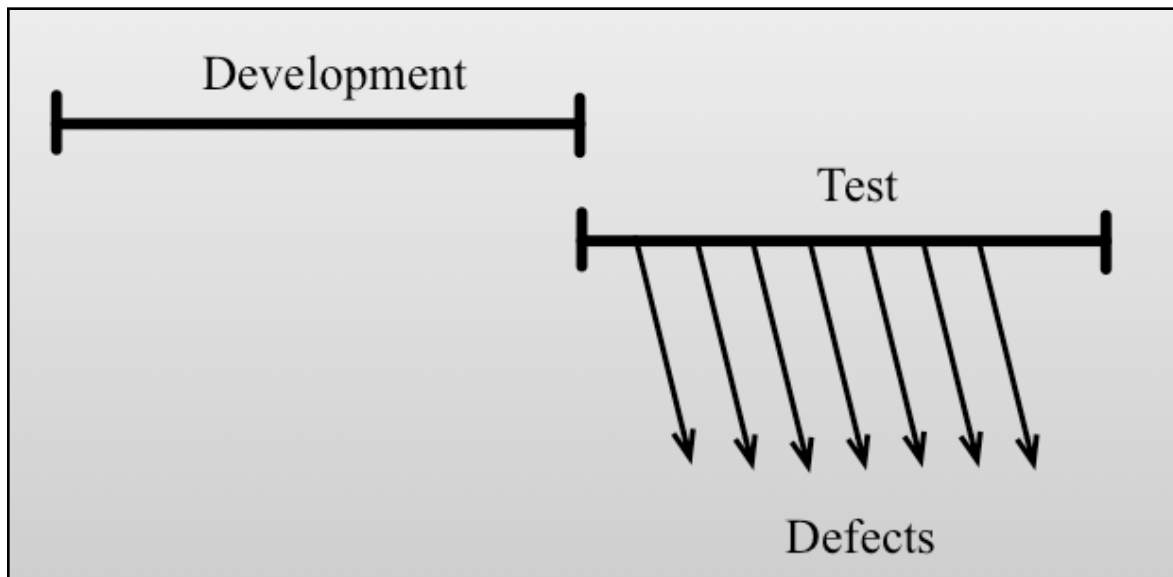# Test-Driven Development for Embedded C[1]

By James W. Grenning

We've all done it—written code and then toiled to make it work. Build it; then fix it. Testing was something we did after the code was done. It was always an afterthought, but it was the only way we knew.

We would spend about half our time in the unpredictable activity affectionately called debugging. Debugging would show up in our schedules under the disguise of test and integration. It was always a source of risk and uncertainty. Fixing one bug might lead to another and sometimes to a cascade of other bugs.

We'd keep statistics to help predict how much time we would need to get the bugs out. We measured and managed the bugs. We would watch for the knee of the curve, the trend that showed we finally started to fix more bugs than we introduced. The knee showed that we were almost done—but we never really knew whether there was another killer bug hiding in a dark corner of the code.

Why do these bugs happen to us? There is a simple answer, we put them there. It's baked into the way we work. When test follows development, it will find defects.

---

[1] Based on my Book Test-Driven Development for Embedded C (www.wingman-sw.com/tddec)

www.wingman-sw.com
james@wingman-sw.com

We make mistakes when we develop; the tests' job is to find the defects. If we are any good at testing, we'll find bugs. Following development by test means we will have to find, fix and manage a boat load of defects.

Let's talk about Debug Later Programming (DLP), the most popular way to program known today. DLP is very risky. It means that we will have to find bugs. We can't be sure when the bugs will show themselves. We are not sure how long it will take to find them. This diagram illustrates the physics of DLP:

T_d T_{find} T_{fix}

Time

Mistake made
(bug injection)   Bug discovery

Bug found   Bug fixed
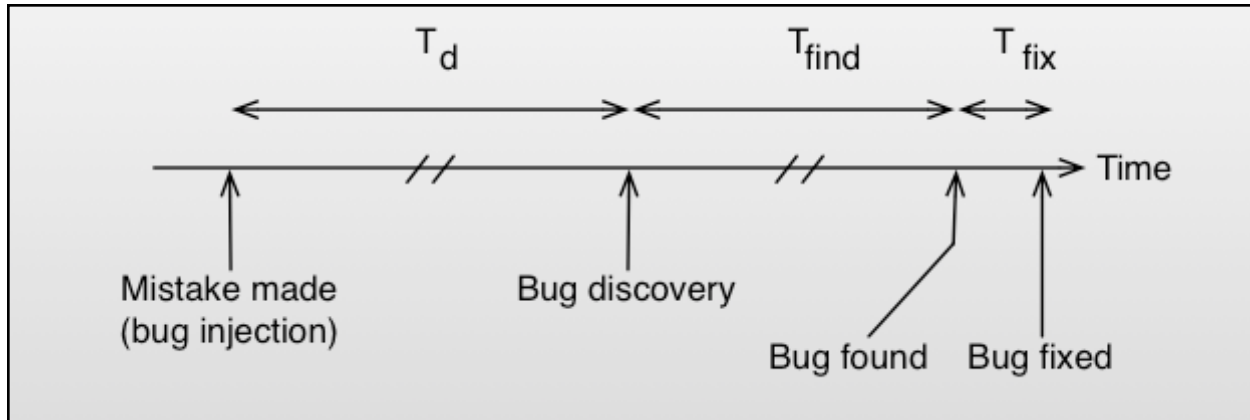
When the time to discover a bug (T_d) increases, the time to find a defect's root cause (T_{find}) also increases, often dramatically. If it's a few hours, days, weeks or months from introduction to discovery we lose context and must start the bug hunt. For the cases where defects are found outside of development, or the current phase, then the bug also has to be managed.

For some bugs, the time to fix the bug (T_{fix}) is often not impacted by T_d. But if the mistake is compounded by other code building on top of a wrong assumption, T_{fix} may increase dramatically as well. Some working features, may also depend on the bug! Some bugs lay undetected or unfound for years.

In the quest to avoid the rush at the bottom of the waterfall, QA started to write regression test so they could quickly run regression tests and find new side effect defects. But we still got surprised; a small mistake could take days, weeks, or months to find. Some were never found.

Some insightful people saw that short cycles led to fewer problems. They saw that aggressive test automation saved time and effort. Tedious and error-prone work did not have to be repeated. Tests could be run without the great expense incurred when mobilizing a small army of manual testers. Side effects were detected quickly; debug sessions were avoided. The hidden defect, a root cause of schedule variability, was contained, and more predictable schedules emerged. Less time was spent chasing bugs at the end of development.

I first learned of Test-Driven Development from Kent Beck's book Extreme Programming Explained [bec00]. In TDD test and production code are developed concurrently in a tight feedback loop.

The TDD Microcycle looks like this:

1. Write a test
2. Watch it not compile
3. Make it compile, but fail
4. Make it pass
5. Refactor (clean up any mess)
6. Repeat until done

The writing of test code and production code is integrated. If we make a mistake and the new test does not pass, we know right away and can fix it. If we get the new test to pass, but introduce an unintended consequence (a bug) the tests tell us. In this activity we write unit test not in prose, but in unambiguous code. Tests are automated and plugged into a unit test harness. Running a retest is free!
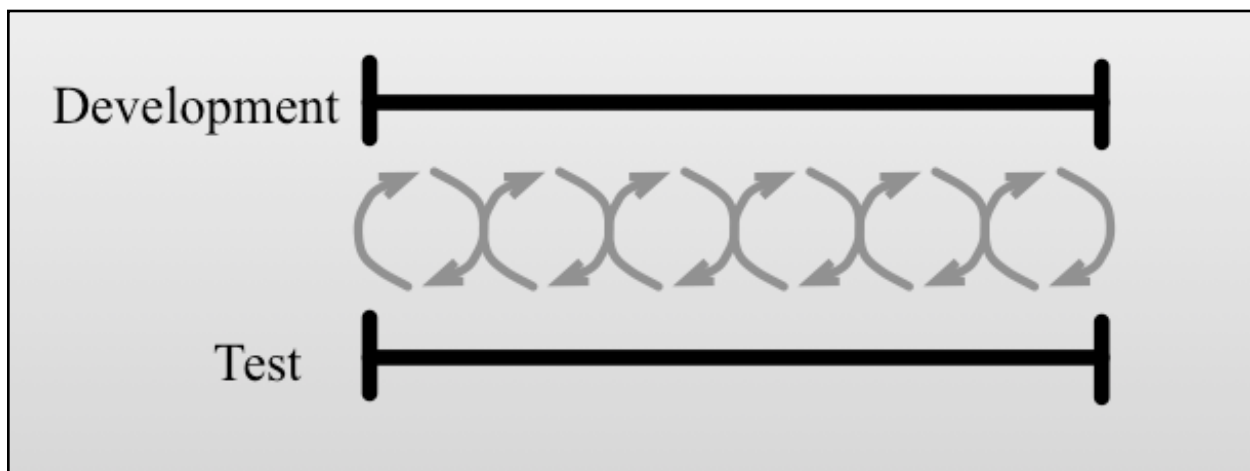
When test and code writing are integrated, we prevent defects. Not all defects, but many of them. This process is designed to prevent defects, and it has a profound effect on design and how we spend our time.

When we have tests drive our code and designs, the physics of development is different.
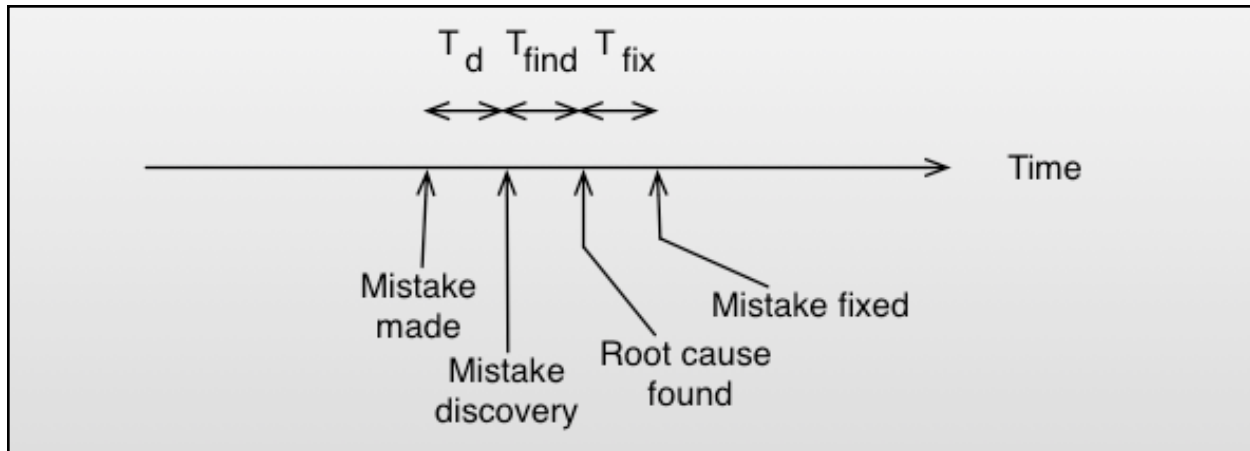
When the time to discover a bug (T_d) approaches zero, the time to find the bug (T_{find}) also approaches zero. A code problem, just introduced, is often obvious. When it is not obvious, the developer can get back to a working system by simply undoing the last change. $T_{find} + T_{fix}$ is as low as it can get, given that things can only get worse as time clouds the programmer's memory and as more code depends on the earlier mistake. In comparison, TDD provides feedback immediately! Immediate notification of mistakes prevents bugs. If a bug lives for less than a few minutes, is it really a bug? No, it's a prevented bug. TDD is defect prevention. DLP institutionalizes waste.

The 30G Zune Bug

Test-Driven Development might have helped to avoid an embarrassing bug, the Zune bug. The Zune is the Microsoft product that competes with the iPod. On December 31, 2008, the Zune became a brick for a day. What was special about December 31, 2008? It's New Year's Eve and the last day of a leap year, the first leap year that the 30G Zune would experience.

Many people looked into the Zune bug and narrowed the problem down to a function in the clock driver. Although this is not the actual driver code, it does suffer from exactly the same bug. See if you can find the cure for the Zune's infinite loop in this code:

```
static void SetYearAndDayOfYear(RtcTime * time)
{
    int days = time->daysSince1980;
    int year = STARTING_YEAR;
    while (days > 365)
    {
        if (IsLeapYear(year))
        {
            if (days > 366)
            {
                days -= 366;
                year += 1;
            }
        }
        else
        {
            days -= 365;
            year += 1;
        }
    }

    time->dayOfYear = days;
    time->year = year;
}
```

Many code-reading pundits reviewed this code and came to the same wrong conclusion that I did. We focused in on the boolean expression (days > 366). The last day of leap year is the 366th day of the year, and that case is not handled correctly. On that day, this function never returns! I decided to write some tests for SetYearAndDayOfYear( ) to see whether changing boolean to (days >= 366) fixes the problem, as about 90 percent of the Zune bug bloggers predicted.

After getting this code into the test harness, I wrote the test case that would have saved many New Year's Eve parties:

```
TEST(RtcTime, 2008_12_31_last_day_of_leap_year)
{
    int yearStart = daysSince1980ForYear(2008);
    rtcTime = RtcTime_Create(yearStart+366);
    assertDate(2008, 12, 31, Wednesday);
}
```

Just like the Zune, the test goes into an infinite loop. After killing the test process, I apply the popular fix based on reviews by thousands of programmers. Much to my surprise, the test fails; SetYearAndDayOfYear() determines that it is January 0, 2009. New Year's Eve parties have their music but the Zune would still have a bug; it's now visible and easily fixable.

With that one test, the Zune bug could have been prevented. The code review by the masses got it close, but still the correct behavior eluded most reviewers. I am not knocking code reviews; they are an important part of software development. But running the code is the only way to know for sure.
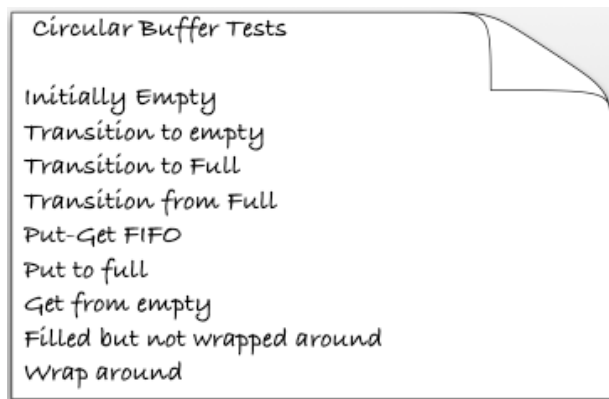
You wonder, how would we know to write that one test? We could just write tests where the bugs are. The problem is we don't know where the bugs are; they can be anywhere. So, that means we have to write tests for everything, at least everything that can break. It's mind-boggling to imagine all the tests that are needed. But don't worry. You don't need a test for every day of every year; you just need a test for every day that matters.

Finally, let's get around to answering "Why do we need TDD?" We need TDD because we're human and we make mistakes. Computer programming is a very complex activity. Among other reasons, TDD is needed to systematically get our code working as intended and to produce the automated test cases that keep the code working.

Getting Ready to Test-Drive

Before starting to write tests its a good idea to know where you are going. You don't need all the answers, but must have a general idea and know some specifics (like an architectural vision and the specific module or feature you are starting with). Some say there is no high-level design in TDD. TDD can fit into many software development processes, though its roots are in evolutionary design. TDD does not dictate how much high-level design you need; every development effort has its own needs. Though with TDD, you probably need less upfront design than you currently do. TDD supports an evolving design and avoids analysis paralysis.

Let's say we are developing something that needs a circular buffer, a FIFO that holds integers. We've decided we need it and its time to work on it. Devising a list of tests is a good way to get the ball rolling. We know we'll put integers in, and take them out. We will be able to check of the buffer is full or empty. We can size it to our needs. Grab a note pad and make a list of tests like this:

Circular Buffer Tests

Initially Empty
Transition to empty
Transition to Full
Transition from Full
Put-Get FIFO
Put to full
Get from empty
Filled but not wrapped around
Wrap around

Don't worry if the list is not 100% complete. It never will be, and if it is, you spent too much time one it. You will discover and expand the tests as you go. Do not get stuck in analysis paralysis composing a test list.

Once you have the test list, get your test fixture ready. I'll use CppUTest, a C and C++ test harness. In my book I use a C-only test harness in the early chapters. CppUTest is a more convenient to use, so I use it here. The CircularBuffer tests will be organized around a TEST_GROUP in a file called CircularBufferTest.cpp.

```cpp
#include "CppUTest/TestHarness.h"

extern "C"
{
#include "common.h"
#include "CircularBuffer.h"
}

TEST_GROUP(CircularBuffer)
{
    CircularBuffer buffer;

    void setup()
    {
        buffer = CircularBuffer_Create();
    }

    void teardown()
    {
        CircularBuffer_Destroy(buffer);
    }
};
```

The parameter of the TEST_GROUP macro is the name of the test group. It also happens to be the name of the module we are testing. There are two special functions in the TEST_GROUP, setup() and teardown(). setup() is run before every TEST, and teardown() is run after every TEST. This way each TEST gets a fresh copy. I have not shown you what a TEST is, so we better get to that.

The CircularBuffer module is one of the easier things to test because there are no external dependencies. It can be tested through its interface. How it behaves is a function of its state. For example, immediately after the buffer is created, it should be empty. Here's the TEST that checks that.

```cpp
TEST(CircularBuffer, EmptyAfterCreation)
{
    CHECK_TRUE(CircularBuffer_IsEmpty(buffer));
}
```

The first parameter of the TEST macro matches its TEST_GROUP. Keep in mind that we are test-driving, so `CircularBuffer_IsEmpty` does not exist. So now we start following TDD. This state machine shows the steps we go through.
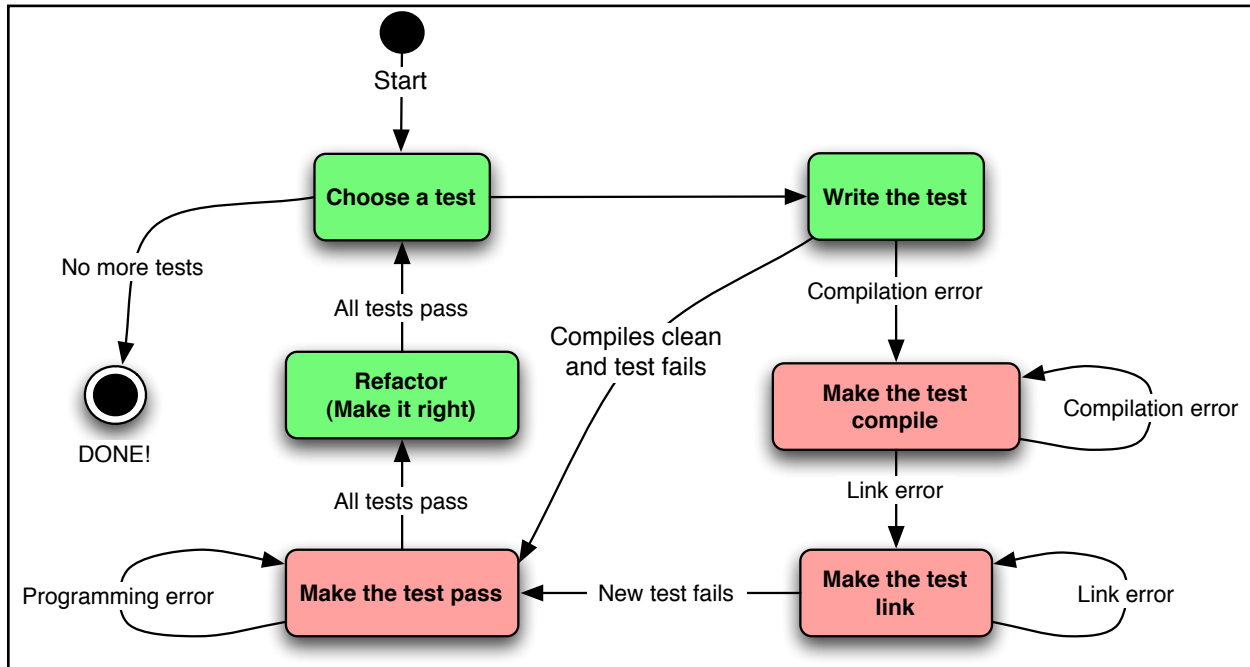
After writing the test and compiling, we are at the "Make the test compile" state. That makes us add the `CircularBuffer_IsEmpty` function prototype to the header file, like this:

```
#ifndef D_CircularBuffer_H
#define D_CircularBuffer_H

typedef struct _CircularBuffer * CircularBuffer;

CircularBuffer CircularBuffer_Create();
void CircularBuffer_Destroy(CircularBuffer);
int CircularBuffer_IsEmpty(CircularBuffer);

#endif
```

I added the create and destroy functions, as well as other boilerplate code, before we started using a shell script (I like to automate repetitive and boring tasks). It is a common starting point for a multiple-instance module.[2] The multiple-instance module

---

[2] See my other talk ESC-204, SOLID Designs for Embedded C

www.wingman-sw.com
james@wingman-sw.com

uses an opaque datatype (or abstract data type [Lis74]) that hides the structure details in the C file.

Now when you compile, you move to the next step with a link error. In TDD we want to focus on solving one problem at a time, so we don't have to go chasing problems all over the place. Notice in the state machine that when leaving the "Make the test link" state that we expect the new test to fail. This assures us that the test is capable of detecting the wrong results. We resolve the link problem with this implementation; it is hardcoded with a return result that fails the test.

```
int CircularBuffer_IsEmpty(CircularBuffer self)
{
    return FALSE;
}
```

The test harness now gives this result:

```
tests/util/CircularBufferTest.cpp:86: error:
    Failure in TEST(CircularBuffer, EmptyAfterCreation)
    CHECK_TRUE(CircularBuffer_IsEmpty(buffer)) failed

.
Errors (1 failures, 1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 1 ms)
```

The test result output shows the exact problem and it's location. Next in the TDD state machine is to make the test pass. We make it pass with a simple and incomplete implementation like this:

```
int CircularBuffer_IsEmpty(CircularBuffer self)
{
    return TRUE;
}
```

Not the test run reports that all tests pass.

```
OK (1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 1 ms)
```

OK, now we are in maintenance. Not so fast you say? Am I going to leave the hardcoded return result? Yes! As we add more tests the implementation will grow and become more complete. Let's go a little further. Here is the next test:

```
TEST(CircularBuffer, NotEmpty)
{
    CircularBuffer_Put(buffer, 31415926);
    CHECK_FALSE(CircularBuffer_IsEmpty(buffer));
}
```

This test drives us to implement another interface, `CircularBuffer_Put`, and has us test another boundary condition, the transition from empty to not empty. Now the hard coded return result will not work. So we can think about how a CircularBuffer works (I know, we've already been thinking about it). Let's look at this diagram to cement our thinking.
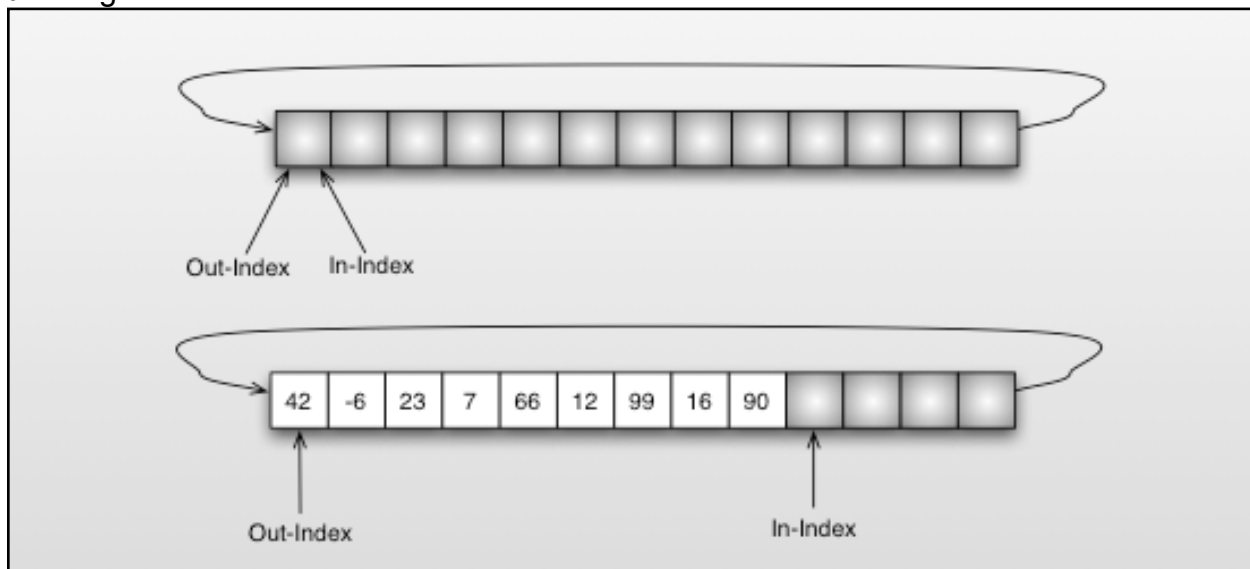
We can anticipate that we will need two indexes. For the current tests a single index will do. If the index is zero, the buffer is empty (at least for our current test scenarios). Following the state machine, we'd add the interface and see the link error, then add an empty implementation for `CircularBuffer_Put` and watch the test fail. I'll leave that to your imagination this time. Here is the minimal implementation that makes both the tests pass:

```
int CircularBuffer_IsEmpty(CircularBuffer self)
{
    return self->index == 0;
}

void CircularBuffer_Put(CircularBuffer self, int value)
{
    self->index++;
}
```

This is obviously not the final code, but it is a step in the right direction. Lets make it a little more complete by transitioning back to empty after a put. This will get both the in and out indexes in place.

```
TEST(CircularBuffer, NotEmptyThenEmpty)
{
    CircularBuffer_Put(buffer, 42);
    CircularBuffer_Get(buffer);
    CHECK_TRUE(CircularBuffer_IsEmpty(buffer));
}
```

Like usually, we walk through the state machine, solving one problem at a time. This implementation passes all three of our tests:

```
int CircularBuffer_IsEmpty(CircularBuffer self)
{
    return self->index == self->outdex;
}

void CircularBuffer_Put(CircularBuffer self, int value)
{
    self->index++;
}

int CircularBuffer_Get(CircularBuffer self)
{
    self->outdex++;
    return -1;
}
```

In reflection, the first few tests are driving us to define the interface, trying the interface out and writing boundary tests. The implementations are incomplete, but the tests are correct. The tests are the safety net as we grow the CircularBuffer's capabilities.

We are doing what Kent Beck calls, "Fake it 'til you make it". How long do we fake it, isn't that wasteful? The fakes are so easy, there is little waste. Getting the feedback, and solving one problem at a time keeps us from having to hunt for mistakes. We choose partial implementations that are a step in the right directions. When should you stop faking it? When it is easier to make it.

This next test will cause us to stop ignoring the input parameters and store them into an integer array. The test drives out faking the return result.

www.wingman-sw.com
james@wingman-sw.com

```
TEST(CircularBuffer, GetPutAFew)
{
    CircularBuffer_Put(buffer, 1);
    CircularBuffer_Put(buffer, 2);
    CircularBuffer_Put(buffer, 3);
    LONGS_EQUAL(1, CircularBuffer_Get(buffer));
    LONGS_EQUAL(2, CircularBuffer_Get(buffer));
    LONGS_EQUAL(3, CircularBuffer_Get(buffer));
}
```

The implementation needed now would not worry about a full buffer, wrapping around or other edge conditions yet. We need tests for those. The supporting implementation could use a big enough linear array. Once we make this test pass, we move on to dealing with those other cases.

While you are getting used to TDD it is hard to resist writing code that is not called for by the tests. It is likely that at least some of the code that gets ahead of the tests will not be thoroughly tested. You will also discover that sometimes your idea of what is needed is more complex than what is really needed. We do TDD to make sure our code is thoroughly tested and is simple.

I'll leave completing the CircularBuffer as an exercise.

What about code with dependencies?

When we are test driving code with dependencies, sometimes we need to stub out the depended upon code. It means we must carefully manage dependencies, and this means programming to interfaces. Let's say we are building a home automation system where the homeowner can schedule lights to turn on and off on specific days, at specific times. This diagram shows our initial design ideas:
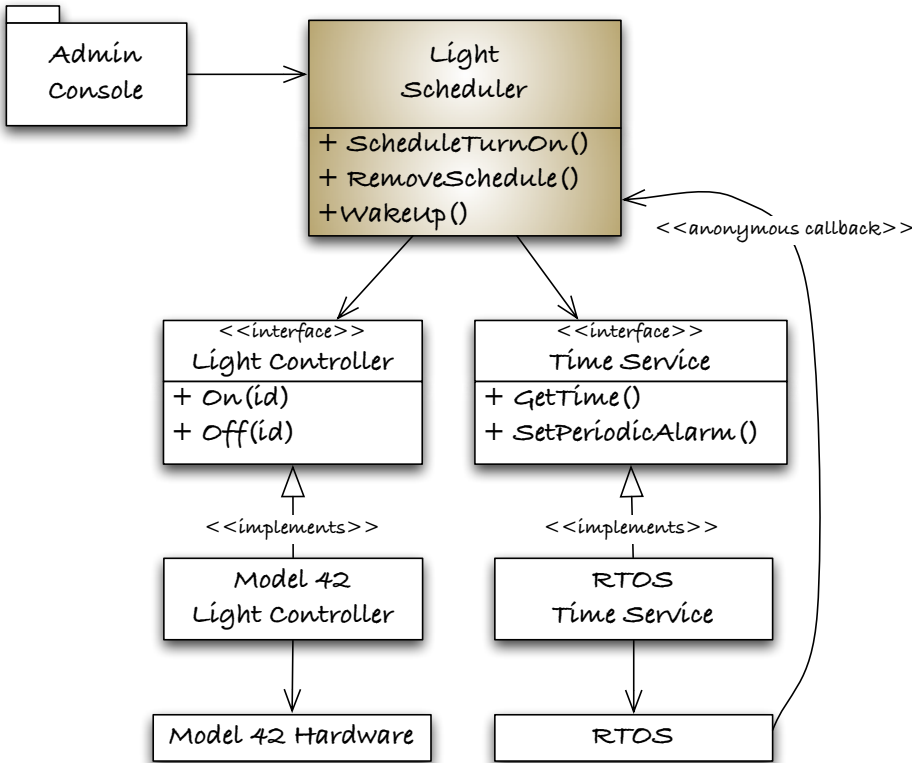
The console can schedule a light to turn on at a specific time. The LightScheduler
keeps the schedule. The TimeService will wake up the LightScheduler through a
callback once a minute. If there are any events scheduled for the current minute, the
scheduler tell the LightController to turn on the scheduled light by its ID.

Testing this in the embedded target will be rather tedious and that means we won't do
the tests very often. Here is a specific manual test procedure:

1. Schedule a light for Friday at 8:00 PM
2. Reset the system clock for Friday at 7:58
3. Wait until 7:59 and see that the light is not turned on
4. Wait until 8:00 and see that the light is turned on

Doing this once is OK, but we won't just do this procedure once… BORING! There are
many more tests needed to check each day, weekday schedules and weekend
schedules.

Instead we can build a test fixture so we can automate all the needed tests for the
LightScheduler and run them in the blink of an eye. Here is the LightScheduler's unit
test environment.
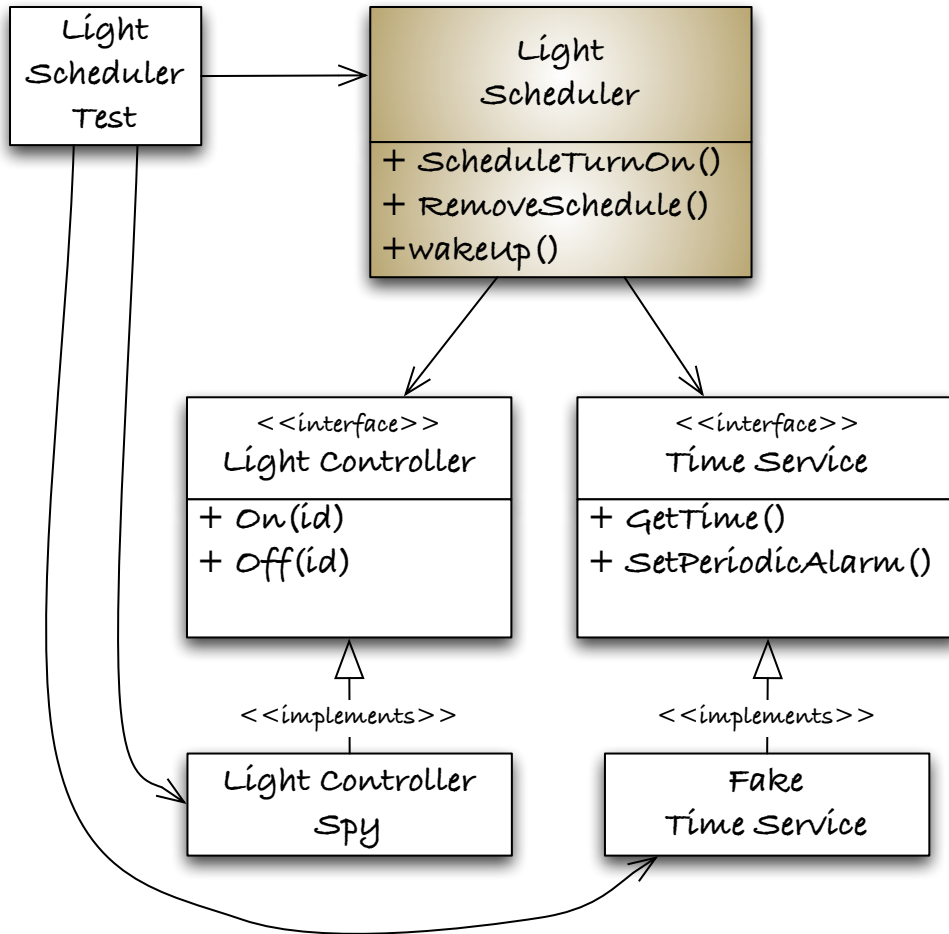
The test case takes the place of the client of the LightScheduler. The FakeTimeService replaces the TimeService during test as the LightControllerSpy does for the LightController. The LightScheduler has no idea that it is not working with test versions of its collaborators.

Test Doubles
The test stub's official name is a Test Double, as defined in Gerard Meszaros book xUnit Testing Patterns [Mes07]. Test doubles are usually simple. They implement the interface of the thing they are substituting. For example, here is the LightController interface:

```
#ifndef D_LightController_H
#define D_LightController_H

void LightController_Create(void);
void LightController_Destroy(void);
void LightController_On(int id);
void LightController_Off(int id);

#endif
```

The interface includes everything needed to turn on or off some light by its ID. It is independent of any particular hardware implementation. The LightControllerSpy has the same interface, plus a few more functions to access what the spy learns. The test double has a secret interface known only to the test cases. Here it is:

```
#ifndef D_LightControllerSpy_H
#define D_LightControllerSpy_H

#include "LightController.h"

enum
{
    LIGHT_ID_UNKNOWN = -1, LIGHT_STATE_UNKNOWN = -1,
    LIGHT_OFF = 0, LIGHT_ON = 1
};

int LightControllerSpy_GetLastId(void);
int LightControllerSpy_GetLastState(void);

#endif
```

The spy implements test stub versions of the LightController interface and the secret interface like this:

```c
#include "LightControllerSpy.h"
#include "memory.h"

static int lastId;
static int lastState;

void LightController_Create(void)
{
    lastId = LIGHT_ID_UNKNOWN;
    lastState = LIGHT_STATE_UNKNOWN;
}

void LightController_Destroy(void)
{
}

int LightControllerSpy_GetLastId(void)
{
    return lastId;
}

int LightControllerSpy_GetLastState(void)
{
    return lastState;
}

void LightController_On(int id)
{
    lastId = id;
    lastState = LIGHT_ON;
}

void LightController_Off(int id)
{
    lastId = id;
    lastState = LIGHT_OFF;
}
```

A spy implementation is usually very simple. The FakeTimeService allows the tests to override the time. It's pretty simple. The test case in the next section will give you an example of its usage.

Test Cases Using Test Doubles

Recall the manual test procedure for the light scheduled on Friday night. Here is a CppUTest version of the test case. I won't show the TEST_GROUP. It basically has setup() call the initialization code needed for each test.

```
TEST(LightScheduler, ScheduleOnFridayNotTimeYet)
{
    LightScheduler_ScheduleTurnOn(3, FRIDAY, 1200);
    FakeTimeService_SetDay(FRIDAY);
    FakeTimeService_SetMinute(1199);

    LightScheduler_Wakeup();

    LONGS_EQUAL(LIGHT_ID_UNKNOWN, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_STATE_UNKNOWN, LightControllerSpy_GetLastState());
}
```

In the above test case, a schedule is set to turn on light number 3 every Friday night at the 1200th minute (8:00PM). The time is faked to be 7:59 on Friday. The scheduler's callback function is called as the RTOS will do in the product. The last two lines interrogate the spy to make sure that no lights have been told to change their state. The test passes when they are still in their initial state of unknown.

This test check that the scheduler can tell when it is the scheduled time to turn on a light.

```
TEST(LightScheduler, ScheduleOnFridayItsTime)
{
    LightScheduler_ScheduleTurnOn(3, FRIDAY, 1200);
    FakeTimeService_SetDay(FRIDAY);
    FakeTimeService_SetMinute(1200);

    LightScheduler_Wakeup();

    LONGS_EQUAL(3, LightControllerSpy_GetLastId());
    LONGS_EQUAL(LIGHT_ON, LightControllerSpy_GetLastState());
}
```

The code that passes this test must turn on a light at the right time on the right day. Notice that the spy has remembered the ID of the light and the state the light was put in. I explore this example in a lot more detail in my book [Gre11].

With this test fixture we could write all the tests we need and run them in the blink of an eye, keeping our code running. I can see tests like these paying for themselves on their first run.

TDD Adapted to Embedded

The TDD demonstrated so far could and would be run on your development machine. It is a good way to make concrete progress without being inhibited by hardware. I recommend using this dual-target approach where tests and code are first written and run on your development machine. Periodically, the test should also be run on the

target environment. Dual-targeting has advantages and risks. I'll cover those in the next subsections.

Target Hardware Bottleneck
Concurrent hardware and software development is a reality for many embedded development efforts. If software can be run only on the target, you will likely suffer unnecessarily from one or more of these time wasters:

- Target hardware is not ready until late in the delivery cycle, delaying software testing.
- Target hardware is expensive and scarce. This makes developers wait and build up mounds of unverified work.
- When target hardware is finally available, it may have bugs of its own. The mound of untested software has bugs too. Putting them together makes for difficult debugging, long days, and plenty of finger pointing.
- Long target build times waste valuable time during the edit, compile, load, and test cycle.
- Long target upload times waste valuable time during the edit, compile, load, and test cycle.
- Long target upload times lead to batching numerous changes in one build, which means that more can go wrong, leading to more debugging.
- Compilers for the target hardware are typically considerably more expensive than native compilers. The development team may have a limited number of licenses available, adding expense and possible delays.

You likely experience some of these problems. They slow progress and reduce feedback needed to build today's complex systems. TDD and dual-targeting can help open the bottleneck, but it has risks.

Risks of Dual-Targeting
Testing code in the development system builds confidence in your code before committing it to the target, but there are risks inherent in the dual-target approach. Most of these risks are because of differences between the development and target environments. These include:
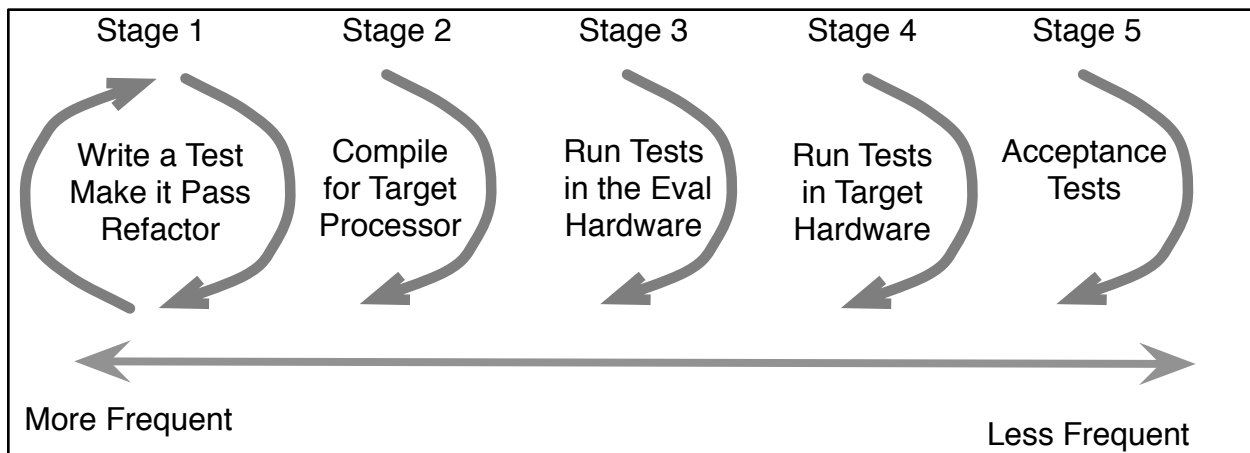
- Compilers may support different language features.
- The target compiler may have one set of bugs, while the development system native compiler has another set of bugs.
- The runtime libraries may be different.
- The include filenames and features may be different.
- Primitive data types might have different sizes.
- Byte ordering and data structure alignments may be different.

Because of these risks, you may find that code that runs failure free in one environment experiences test failures in other environments.

The fact that there are potential differences in execution environments should not discourage you from dual-targeting. On the contrary, these are all workable obstacles on the path to getting more done. But it's best to take this path with eyes open and knowledge of some of the spear-filled pits that await further down the path. With the benefits and risks enumerated, let's see how the embedded TDD cycle overcomes the challenges, without compromising the benefits.

The Embedded TDD Cycle
TDD is most effective when the build and test cycle takes only a handful of seconds. A longer build and test time usually results in taking bigger steps; with the bigger steps come more things that can be broken, leading to more debugging when the test finally is run. The need for fast feedback leads us to move the TDD microcycle off the target to run natively on the development system. The TDD microcycle is the first stage of the embedded TDD cycle shown here:

Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
Write a Test Make it Pass Refactor	Compile for Target Processor	Run Tests in the Eval Hardware	Run Tests in Target Hardware	Acceptance Tests

More Frequent ⟶ Less Frequent

Stage one gives you fast feedback while you are programming. Each change can be quickly verified. But you are building and running on your host development machine, so there can be differences. Stage 2 makes sure that your code compiles in both environments. Stage 3 makes sure that the code runs the same in both the host and the target processor. Sometimes stage 3 makes up for very constrained memory in the target. Stage 3 is not always needed if there is a reliable target with space to run the unit tests. Stage 4 runs the tests in the target. We could introduce some hardware dependent unit tests in stage 4. Stage 5 is what you are already used to doing, seeing if your system works as it should when it is fully integrated. It's a good idea to automate at least some of stage 5.

By going through these stages, we expect to find problems at the appropriate stage. For example we would expect each stage to help find these problems.

Stage	Problems Likely to Find in Stage
1	Logic, design, modularity, interface, boundary conditions
2	Compiler compatibility (language features) Library compatibility (header files, prototypes)
3	Processor executions problems (bugs in compiler and standard libraries) Portability problems (word size, alignment, endian)
4	Ditto stage 3 Hardware integration problems Misunderstood hardware specifications
5	Ditto stage 4 Misunderstood feature specification

The embedded TDD cycle won't prevent all problems. Though it should help to find the majority of the problems close to the time they are introduced and in an appropriate stage.

Stages 2 through 4 should be able to be executed manually and they should be executed automatically upon checkin or at least nightly. A continuous integration server (such as Cruise Control or Jenkins) can watch your source repository and initiate builds after checkin.

Something not mentioned here, is the role of functional tests. The unit tests you have seen in this paper are feedback to you, the programmer, that the code does what you think. Functional tests (also known as integration tests, Story Tests or Executable Use Cases) can take advantage of the test point developed for TDD to write higher level tests that show that specific features or system functionality meets the customer needs.

You can find out about Story Tests and Executable Use Cases in my paper presented at ESC Boston 2010 (http://www.renaissancesoftware.net/papers.html, http://bit.ly/gIPQjn)

Closing Thoughts

TDD helps you, the programmer make sure that your code does what you think it does. How can you build a reliable system if it does not? TDD helps you get the code right in the first place, but it does more. It creates a regression test suite that helps you keep your code working.

One of TDD's powers is defect prevention. We waste considerable effort in our industry finding, chasing and fixing bugs. Many people are preventing bugs with TDD. TDD fundamentally changes how you program. Is the potential of defect prevention tempting you to give TDD a try? Give it a try. Finish the CircularBuffer. Try it on your next bit of new code.

You've got legacy code? Try to get the legacy code into your test harness. Write tests for the current behavior. Test drive in the new behavior. Changing legacy code is dangerous. It's best to go carefully. [Fea04]

Tis table summarizes some of the benefits people using TDD enjoy:

Fewer bugs	Small and large logic errors, which can have grave consequences in the field, are found quickly during TDD. Defects are prevented.
Less debug time	Having fewer bugs means less debug time. That's only logical, Mr. Spock.
Fewer side effect defects	Tests capture assumptions, constraints, and illustrate representative usage. When new code violates a constraint or assumption, the tests holler.
Documentation that does not lie	Well-structured tests become a form of executable and unambiguous documentation. A working example is worth 1,000 words.
Peace of mind	Having thoroughly tested code with a comprehensive regression test suite gives confidence. TDD developers report better sleep pat- terns and fewer interrupted weekends.
Improved design	A good design is a testable design. Long functions, tight coupling, and complex conditionals all lead to more complex and less testable code. The developer gets an early warning of design prob- lems if tests cannot be written for the envisioned code change. TDD is a code-rot radar.

Progress monitor	The tests keep track of exactly what is working and how much work is done. It gives you another thing to estimate and a good definition of done.
Fun and rewarding	TDD is instant gratification for developers. Every time you code, you get something done, and you know it works.

There is more to using TDD with C than this short paper can convey. My book covers the topic thoroughly. My other ESC talks cover other aspects of embedded development.

ESC-204 SOLID Design for Embedded C - This talk and paper get into some of the other constructs you can use to make flexible and testable designs.

ESC-214 Agile Embedded Software Development - This talk provides an overview of agile development. TDD comes from Extreme programming, on of the original agile techniques

Other papers and presentations for prior Embedded Systems Conferences can be found at http://www.renaissancesoftware.net/papers.html, and on my blog at http://www.renaissancesoftware.net/blog.

If you want to learn more, the bibliography has good places to go for more information.

Also, to get into the discussion, come to Agile Embedded yahoo group.
http://tech.groups.yahoo.com/group/AgileEmbedded

www.wingman-sw.com
james@wingman-sw.com

Bibliography

[Bec00]	Extreme Programming Explained: Embrace
[Fea04]	Michael Feathers. Working Effectively with Legacy Code. Prentice Hall, Englewood Cliffs, NJ, 2004.
[Gre11]	James W. Grenning, Test-Driven Development for Embedded C, Pragmatic Bookshlef, 2011. www.pragprog.com/titles/jgade
[Lis74]	Change. Addison-Wesley, Reading, MA, 2000.
[Mar02]	Robert C. Martin. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, Englewood Cliffs, NJ, 2002.
[Mes07]	Gerard Meszaros. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, Reading, MA, 2007.
[Mey97]	Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.