Dr. Robert Koss and Jeff Langr
Test Driven Development in C

Introduction

Over the course of the last two years, our company has been promoting, conducting training, and consulting in XP (eXtreme Programming). We believe that XP is the best technique for delivering high-quality software because of its emphasis on communication, simplicity, testing, and rapid feedback. XP achieves its results through implementation of about a dozen proven practices. We are only going to focus on one those practices in this article — TDD (Test Driven Development).

Many companies fear they can't be eXtreme for a variety of reasons, but we can't imagine a company or process that would advocate not testing their software. The TDD technique can be, and should be, applied in any software development environment. You don't even need your boss' permission to do TDD because it is done as part of programming instead of a separate phase of the project. While testing, design, and programming are three distinct activities, we don't separate these activities in time — they are done concurrently and continuously. The example presented in this article demonstrates how this works.

Nearly all of the publications concerning XP or TDD use a programming language that supports OO (object oriented) development. This is understandable since XP had its birth in the Smalltalk community and many — but not all — shops are in fact using an OO language.

A recent client of ours wanted to do TDD. In fact, they wanted to do all of the XP practices. The problem was that they were a C shop, and they were going to be using C for the foreseeable future, so we had to roll up our sleeves and figure out how to get the benefits of TDD in C. Unfortunately C doesn't (directly) support polymorphism, a concept that is fundamental to the techniques used in TDD.

In TDD, one module is developed at a time. To properly test a given module, it must be isolated from most other modules with which it interacts. In OO languages like C++ and Java, we would use polymorphic interfaces to isolate the module under test. These interfaces also provide a testing environment for the module. We need these same capabilities in C.

We used our experience working in embedded systems projects where either the hardware wasn't ready or wasn't available to us. In these environments, we would have to stub out the function calls that provided access to the missing hardware. This stubbing technique is just what we need to get the benefits that polymorphism provides in languages that support it. In fact, stubbing is polymorphism, and we refer to it as link-time polymorphism.

TDD

TDD is a style of programming where we write a test before the code that it is supposed to test. The test shouldn't even compile at first because the code to be tested hasn't been written yet. We then write enough code to get everything to compile, but purposely code it so that our test fails. We then write code to make the test pass. Each of these steps is critical to the TDD process.

Finally, every time our tests pass, we refactor the code so that it is the best, resume-quality code that we can produce. We strive to make our code have the best possible design for the functionality we have implemented so far, but for nothing more than that functionality. Only after we have refactored toward these goals do we go on to

writing another test. We repeat this cycle of test-code-refactor in rapid succession throughout our development day.

How often have you spent an hour writing code, only to spend the entire day getting it to compile and debugged? We don't enjoy doing that because most of the day is spent feeling like a failure for not getting the code correct the first time. Instead of programming a (possibly large) piece of functionality, TDD dictates that we work on a very small problem, solve it, and then move on to the next very small problem. After all, every large complex problem is just a bunch of little problems. We've found that by doing this, we can be successful throughout the entire day. Solving small problems, one after another, is sustainable, enjoyable, and highly productive.

The key to these series of successes is rapid feedback. We run the compiler just as soon as we have enough code that stands a chance of compiling. Compiling after every single line may not be possible in practice, but it's not as far off as you might think. We also run our tests as often as possible. Our rule is that no more than 10 minutes may go by without having all of our tests running. Five minutes is better. One minute is better still. This obsession with running our tests frequently forces us to partition our work into a series of very small problems.

You might initially find this unnatural — we certainly did when we first learned about TDD. But after programming in this style for the last two years, there is no way we'll ever go back to our old ways of writing code and either testing it afterwards or handing it off to QA for testing. Have you ever tried to test code after it was written? It is very difficult, so difficult that it's often not done. By writing our tests first, we guarantee that there are tests and that the system is designed to be testable.

Since adopting TDD, we have almost never used a debugger on the code we have written. In fact, cranking up the debugger — or even adding a `printf` statement — is a sign of failure: we bit off a bigger problem than we could chew. Our response to the supposed need for debugging is to simply throw away the last few minutes worth of work. We then try again, taking even smaller steps.

Point-of-Sale System

As an example, we've chosen a point-of-sale system, like one might encounter in any retail store. A sales clerk passes items over a scanner that reads the barcode. A display monitor shows information on each item as it is scanned and also tracks the total for each sale of one or more items to a given shopper.

Our First User Story [1]

The total price of all items in the sale is available.

We have the concept of a `Sale`, and we know that it has a total. Let's capture that in a test. We are essentially coding our requirements document so that it executes as a suite of unit tests! This is an important concept when practicing XP. We do not produce thick paper requirements or design documents that never seem to match the working code. Yet documenting requirements and design is essential to software development, so we do it in code instead.

Writing tests in code is essential for a second reason: executing tests manually is extremely time consuming. Ultimately it becomes impossible to keep up with executing all tests against even a modest body of code. We have to automate the tests; otherwise they don't get run. As soon as tests aren't being run, we're shipping defects.

In TDD, a testing framework is an essential tool. It not only provides us with a means for automating the tests, but it is also an inte-

gral part of the minute-to-minute development process. Such a tool need not be an expensive investment. The tool we use here — CppUnitLite — is a macro-based tool that is freely available from our website (<www.objectmentor.com>). CppUnitLite was developed for C++, but since C++ can call C, we use it for development of C applications as well. Also, its use of C++ is minimal so that C developers with little or no knowledge of C++ can quickly learn how to use it.

For our user story, we first create the file `SaleTest.cpp` to contain the tests for `Sale`. CppUnitLite requires a little boilerplate code:

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

int main()
{
   TestResult tr;
   TestRegistry::runAllTests( tr );
   return 0;
}
```

The `main` function acts as the test driver. The basic idea is that the `runAllTests` method is responsible for extracting and executing all of the tests we have defined. It isn't necessary to know how the testing framework works to use it — in fact we use it in our C++ course starting on the first day of class, long before students know enough C++ to understand how it works.

Now that the boilerplate code is in place, we can write our first test. Each test is constructed with the `TEST` macro that is part of CppUnitLite. Anything defined with this macro is automatically picked up by `runAllTests`. The macro takes two parameters: an arbitrary name used to group the tests and a descriptive name for the test.

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

int main()
{
   TestResult tr;
   TestRegistry::runAllTests( tr );
   return 0;
}

TEST( Sale, totalNewSale )
{
   CHECK( 0 == GetTotal( ) );
}
```

The name of our first test is `totalNewSale`. It represents the state of a new `Sale` before any `Items` are added to it. It seems reasonable that before anything is purchased the total of the `Sale` should be zero, so we express this in our test.

Note that all tests contain one or more `CHECK` macros. We use the `CHECK` macro to ensure that a Boolean condition holds true. A test may have more than one `CHECK` statement; each of these is executed regardless of whether a prior `CHECK` passed or failed.

`SaleTest` will of course not compile, since the function `GetTotal` doesn't exist yet. We compile anyway as our first level of feedback. If `SaleTest` compiles, we know we have a serious configuration problem.

```
g++ -c SaleTest.cpp
SaleTest.cpp:
   In method 'void totalNewSaleSaleTest::run(TestResult &)':
```

```
SaleTest.cpp:12:
   implicit declaration of function 'int GetTotal(...)'
```

Once we get our compile errors, we create just enough of `Sale.h`:

```
#ifdef __cplusplus
extern "C"
{
#endif

int GetTotal();

#ifdef __cplusplus
}
#endif
```

We include `Sale.h` in `SaleTest.cpp` and create just enough of `Sale.c`.

```
#include "Sale.h"

int GetTotal()
{
   return -1;
}
```

Note the use of the linkage specifier `extern "C"` in `Sale.h`. This is necessary because our testing framework is written in C++ and `#includes` this file; `Sale.c` is written in C and also includes this file. This is the mechanism needed for C++ code to call C code.

Our purpose is to get to compilation so that we can link and run our tests. To run the test, we simply execute `SaleTest`:

```
Failure: "0 == GetTotal( )" line 13 in SaleTest.cpp
```

There was one failure.

This is as expected: we have written a test but no production code yet. We want to see a failure, though, as part of our feedback-driven, test-first design cycle. A reminder of the cycle:

- Write a test for nonexistent code.
- Expect that the test will fail.
- Write just enough code to make the test pass.
- Refactor.

Every once in a while, the test that we expect to fail passes. This should be a shock!

For our `Sale` application, we can make the test pass by simply returning `0` from `GetTotal`.

```
int GetTotal( )
{
   return 0;
}
```

As ridiculous as it looks to just return a hard-coded value to make the test pass, this is part of TDD. We have a failing test case, and we want to get it to pass as quickly as possible. Once the test is passing, we'll worry about making it "right." Returning `0` seems to be "wrong" in the sense that the hard-coded zero will have to be replaced very shortly. But the fact that we provided just enough code to make the current test pass means that we will have to write more tests. These tests will fail, proving that a more complicated

algorithm is necessary. We increase our test coverage this way at a steady pace. More tests are a good thing.

There were no test failures.

Our first test is complete. We know the state a `Sale` should be in when nothing has been done to it. Now we want the capability to sell at least a single item, so we write the test `sellOneItem`. For our test, we make up an item whose barcode is `"a123"` and cost is 199 cents. (It's easiest to count pennies to avoid any floating-point rounding problems later.)

```
TEST( Sale, sellOneItem )
{
   BuyItem( "a123" );
   CHECK( 199 == GetTotal( ) );
}
```

To sell an item, we need to pass in its barcode string to the `Sale`. We devise the function `BuyItem` to accomplish this. Buying item `"a123"` should increase the total of the `Sale` to 199. We compile, expecting failure:

```
SaleTest.cpp:18:
   implicit declaration of function 'int BuyItem(...)'
```

To pass compilation, we must modify the header:

```
int GetTotal( );
void BuyItem( char* barCode );
as well as Sale.c:
#include "Sale.h"

int GetTotal( )
{
   return 0;
}

void BuyItem( char* barCode )
{
}
```

We can now run all the tests, expecting `sellOneItem` to fail.

```
Failure: "199 == GetTotal( )" line 19 in SaleTest.cpp
```

There was one failure.

Time to add the code to make this test pass. Each shopper's purchases must start with a total of `0`. At this point, the simplest thing that will work is to introduce a variable to track the total. We declare it in `Sale.c` as a `static`. This gives the variable visibility only at file scope.

In `Sale.c`, we return `total` from `GetTotal` and update it in `BuyItem`.

```
#include "Sale.h"

static int total = 0;

int GetTotal()
{
   return total;
}

void BuyItem( char* barCode )
{
```

```
   total = 199;
}
```

Both tests now pass [2].

After each new test passes, we always pause and look over the code to be sure that it communicates. We see the string literal and magic number 199 in `sellOneItem` and feel that the code isn't as clear as it could be. Since the tests will evolve and be maintained along with the code that is being tested, we want them to serve as excellent documentation on how to manipulate the `Sale` functions. This obsession we have with clearly communicating code is far better than placing comments in the code, which quickly become outdated. Introducing local variables in the test yields:

```
TEST( Sale, sellOneItem )
{
   char* milkBarcode = "a123";
   int milkPrice = 199;

   BuyItem( milkBarcode );
   CHECK( milkPrice == GetTotal( ) );
}
```

We can now move on and write a third test, which we expect to fail. Note that we already have a need to reuse the constant declarations of `milkBarcode` and `milkPrice`, so we move them out of `sellOneItem` and declare them as `static`.

```
TEST( Sale, sellTwoItems )
{
   BuyItem( milkBarcode );
   BuyItem( milkBarcode );
   CHECK( 2 * milkPrice == GetTotal( ) );
}
```

It fails, as expected:

```
Failure: "2 * milkPrice == GetTotal( )" line 29 in SaleTest.cpp
```

There was one failure.

Building the correct production code in `Sale.c` appears to be straightforward. Instead of just setting the total to the price of milk, we add the price into the total:

```
void BuyItem( char* barCode )
{
   total = total + 199;
}
```

We compile and run our tests again, expecting success. This time we are (somewhat) surprised, as we receive the same error again.

After scratching our heads for a few seconds, we suspect that the static total is probably not getting initialized. We probe for an answer via our tests (not the debugger):

```
TEST( Sale, sellTwoItems )
{
   CHECK( 0 == GetTotal( ) );
   BuyItem( milkBarcode );
   BuyItem( milkBarcode );
   CHECK( 2 * milkPrice == GetTotal( ) );
}
```

Aha. This time we receive two failures. The first one confirms our suspicion — `total` is not initialized at the start of `sellTwoItems`:

```
Failure: "0 == GetTotal( )" line 27 in SaleTest.cpp
Failure: "2 * milkPrice == GetTotal( )" line 30 in SaleTest.cpp
```

We just love that rapid feedback that TDD gives us!

The way the test harness works is that each `TEST` is supposed to be independent of one another. In fact, you cannot guarantee that the tests will run in any particular order. (This actually introduced some problems for us — see [2].) Each test needs to build its own setup.

We hate day-long debugging sessions that result when one test is unwittingly dealing with the remnants of some previous test. However, if you follow the rule of writing just enough code to make tests pass, you will be forced to add more tests. Usually these additional tests will uncover such insidious problems.

For the `Sale` example, we wrote just enough code to get it to work for one item and one item only. That forced us to write a test for selling two items, which exposed our error. To fix the problem, we introduced an initialization function that each test — and therefore each client that will use `Sale` — will have to call.

```
static char* milkBarcode = "a123";
static int milkPrice = 199;

TEST( Sale, totalNewSale )
{
   Initialize( );
   CHECK( 0 == GetTotal( ) );
}

TEST( Sale, sellOneItem )
{
   Initialize( );
   BuyItem( milkBarcode );
   CHECK( milkPrice == GetTotal( ) );
}

TEST( Sale, sellTwoItems )
{
   Initialize( );
   CHECK( 0 == GetTotal( ) );
   BuyItem( milkBarcode );
   BuyItem( milkBarcode );
   CHECK( 2 * milkPrice == GetTotal( ) );
}
```

After declaring `Initialize` in `Sale.h`, we add its definition to `Sale.c`:

```
void Initialize()
{
   total = 0;
}
```

All tests run at this point.

Dealing with Databases

So far, we are only supporting selling one product at a $1.99. We add a fourth test, to ensure that we can deal with selling two products.

```
TEST( Sale, sellTwoProducts )
```

```
{
   Initialize( );

   char* cookieBarcode = "b234";
   int cookiePrice = 50;

   BuyItem( milkBarcode );
   BuyItem( cookieBarcode );

   CHECK( milkPrice + cookiePrice == GetTotal( ) );
}
```

We compile and run the tests, which fail:

```
Failure: "milkPrice + cookiePrice == GetTotal()" line 46 in
SaleTest.cpp
```

There was one failure.

Getting this test to pass is simple:

```
void BuyItem( char* barCode )
{
   if ( 0 == strcmp( barCode, "a123" ) )
      total += 199;
   else if ( 0 == strcmp( barCode, "b234" ) )
      total += 50;
}
```

We know that we have to do a lookup someplace using the barcode. Since we have passing tests, we can now refactor to do a lookup, all the while ensuring that all our tests still pass. Our refactoring is very incremental: we change only a very small amount of code before getting feedback by running our tests. It is otherwise too easy to introduce a defect while making dramatic code changes.

Our small step for now is to introduce a placeholder for our lookup in the form of a call to a `GetPrice` function. The code in `BuyItem` will demonstrate what we want it to do. The real details of the how will come later.

```
void BuyItem( char* barCode )
{
   int price = GetPrice( barCode );
   total += price;
}
```

In other words, our intent is that some other function will do the job of looking up the item price by barcode. This is known as programming by intention. It is a very powerful technique because it forces us to work on smaller and smaller problems, until the how is obvious. For now, we move the conditional logic from `BuyItem` to the `GetPrice` function:

```
int GetPrice( char* barCode )
{
   if ( 0 == strcmp( barCode, "a123" ) )
      return 199;
   else if ( 0 == strcmp( barCode, "b234" ) )
      return 50;
}
```

We add the declaration of this function to `Sale.h` and run our tests to ensure they still pass.

Now where is `GetPrice` actually going to find the price of the item? In the database, of course.

But using a real database for development is a major undertaking. We don't want to go up against a live database, so we have to get a working snapshot. Then we'll have to track any changes that the DBA makes to the real database to keep our snapshot up to date.

There's also the performance consideration of working with a database. We'd have to connect to it for each test, something that takes a lot of processing time. Yet we need to run our tests every few minutes. The slowness of establishing the connection would significantly decrease the amount of work we get done each day.

Instead, we want to stub out the calls to the database. All that `BuyItem` cares about is that `GetPrice` returns the price of the item. It doesn't care if the price comes from a database lookup or if `GetPrice` makes up a random price on the spot. We do, of course, have to test the real `GetPrice` when we use TDD to build the real database code.

In order for the call to `GetPrice` to be able to do both a database lookup in production code, as well as provide the numbers that we want for our tests, we have to access it indirectly. The usual way to get different behavior from a function is to use a function pointer. The test will set the pointer to its own stub lookup function, and the production code will set the pointer to a function that does the live database lookup.

In `SaleTest`, we start by supplying a stub function, `GetPriceStub`, for the lookup:

```
int GetPriceStub( char* barCode )
{
   if ( 0 == strcmp( barCode, "a123" ) )
      return 199;
   else if ( 0 == strcmp( barCode, "b234" ) )
      return 50;
}
```

Our tests then need to pass a pointer to this stub function as a parameter to `Initialize`. We change the call to `Initialize` in each test:

```
Initialize( &GetPriceStub );
```

This won't compile. We fix the prototype and definition of `Initialize` in `Sale.h` and `Sale.c`. Its new signature is:

```
void Initialize( int (*LookUpPrice)(char* barCode)  )
```

`Initialize` then needs to store that function pointer. Once again, we use a `static` variable declared in `Sale.c`.

```
static int (*LookUpPrice)(char* barCode);

void Initialize( int (*db)(char*) )
{
   total = 0;
    LookUpPrice = db;
}
```

Finally, `GetPrice` is altered so that it dereferences the database function pointer in order to call the database lookup function.

```
int GetPrice( char* barCode )
{
   return LookUpPrice( barCode );
}
```

We build, run tests, and succeed.

Now we can get the price of the item in any way that we choose. For the test, we use a simple if/else construct to return hard-coded prices based on the barcodes we know about. In the production code, whoever constructs the point-of-sale system will have to supply a function pointer to `Sale` that accesses the real database. (This function would of course have its own set of tests!)

Link-Time Polymorphism

Pointers are powerful but very dangerous, evidenced by the fact that the major languages introduced in the last decade have moved to eliminate them. The problem with pointers is that we must make sure that they refer to what we intend. It is all too easy for a stray pointer to cause the program to crash, inflicting late-night debugging sessions for the entire development team. Because dereferencing an invalid pointer is a certain bug and probable crash, many programmers test the validity of each pointer prior to dereferencing it. Such checking makes code difficult to read.

There is an alternative to using pointers pointing at different functions: link-time polymorphism. We instead use the linker to link in different functions. The function `BuyItem( char* )` will continue to call `GetPrice( char* )`, but there will be two `GetPrice( char* )` functions defined: one for testing, which we'll place in the test file `SaleTest.cpp`, and another, which we'll place in a file `GetPrice.c`.

When we're building for test every few minutes throughout the development day, we link in `SaleTest.o` and `Sale.o`. When we want to build the real system, we link in the real application and `GetPrice.o`, but not `SaleTest.o`. Here is a very simplistic `Makefile` to illustrate the two builds:

```
POSApp: Sale.o GetPrice.o main.o
   g++ -o POSApp Sale.o GetPrice.o main.o

SaleTest: SaleTest.o Sale.o
   g++ -o SaleTest SaleTest.o Sale.o
../TestHarness/TestHarness.a

SaleTest.o: SaleTest.cpp Sale.h
   g++ -c SaleTest.cpp

Sale.o: Sale.c Sale.h GetPrice.h
   gcc -c Sale.c

GetPrice.o : GetPrice.c GetPrice.h
   gcc -c GetPrice.c
```

Conclusion

An extremely important thing to note in this exercise is the very small steps that we took during our development. Our goal was to ensure that we incrementally improved the system at a steady rate. We wanted to see our tests pass every 5 to 10 minutes or less. This kept us from spending too much time going in the wrong direction. At the end of the day, the product does more than it did at the beginning of the day, and it does it correctly.

TDD, to put it mildly, is awesome. We also think it is the most important "movement" in software development today. We continually meet developers who tell us they have tried it and would never willingly give it up. And as we've demonstrated here, TDD is not just for objects: you can do it in C. In fact, our position is that since

C is such a powerful but dangerous language due to its low-level nature, you really must protect yourself with TDD.

The example we chose was admittedly scaled down so that we could illustrate the techniques of TDD. TDD has been proven to scale to large and complex systems. In fact, one of the chief goals of TDD is to ensure that we take simplified approaches to complexity. Properly managing complexity through TDD means that we can build systems that can be maintained over long periods of time at a reasonable cost.

`Saletest.cpp` (available for download at <www.cuj.com>) shows the final code demonstrating the link-time polymorphism technique.

Notes

[1] A user story is a requirement that: 1) has demonstrable business value, 2) can be tested, and 3) can be completed within an iteration (typically two weeks).

[2] We encountered some interesting results when we coded this `Sale` example on a second machine. The tests ran in a different order — not top to bottom — because the testing framework produces a linked list of static objects, and the order of static initialization isn't guaranteed across compilers and operating systems. In this case, `sellOneItem` ran prior to `totalNewSale`. The total variable thus held a non-zero amount when `totalNewSale` executed, causing it to fail. This would have ended up being slightly earlier feedback to tell us to initialize the variable properly. ❑

Dr. Robert S. Koss is a senior consultant at Object Mentor, Inc. He has been programming for 29 years, the last 15 years in C and C++. Dr. Koss has conducted hundreds of classes in C, C++, and OO Design throughout the world, training thousands of students.

Jeff Langr is a senior consultant at Object Mentor, Inc. He has authored several articles as well as the book *Essential Java Style* (Prentice Hall, 1999). Langr has over 20 years of software development experience, including 10 years of OO development in C++, Smalltalk, and Java.