Unit Testing a C++ Database Application with Mock Objects

Ray Lischner
rlischner@proteuseng.com

Abstract

Sometimes, an idea isn't valuable because it's new but because it's old. Unit testing isn't new. Databases aren't new. Database client libraries aren't new. Writing a database client library that directly supports unit testing via mock database classes and the injection of mock results—that's valuable.

Unit testing is not new, and has long been a common element of software testing. Various design and development techniques, such as extreme programming and test-driven design, have reinvigorated unit testing as a best practice. In order to unit-test an application effectively, the developer must be able to isolate the code under test. The most common method of isolating code for unit testing is to use mock classes or objects to stand in for other classes, especially external services and interfaces.

Java and similar languages offer language support for easy definition and construction of mock objects. C++ presents additional challenges. Some C++ utilities help you write mock classes, but in order to unit-test effectively, a C++ library must be designed for testability. Given the evident value of unit testing, it is unfortunate that so many important libraries are not designed with unit testing in mind.

This paper presents a new C++ database client library, one that has been designed from the start with unit testing as a primary design goal. This library uses abstract interfaces to enable the developer to follow a clean database interface, and to substitute a mock implementation for unit testing and a real implementation for production use. The application code needs to be compiled only once, so you can be sure that you are unit testing the real application code, even when linked with the mock database library.

The resulting library enabled delivery of a database application with zero known defects. The paper presents the design decisions and shows how testability drove the library design, and how the library and resulting applications benefited from the emphasis on unit testing.

Biography

*Mr. Lischner has been designing and developing software for about three decades, starting at the California Institute of Technology, where he earned his B.S. in computer science, and subsequently at large and small companies on both coasts of the United States. Meanwhile, he earned his M.S., also in computer science, from Oregon State University and performed stints as an author (*C++ in a Nutshell*,* Exploring C++*,* Shakespeare for Dummies*, and other books), consultant, teacher, and stay-at-home dad. Many years ago, Ray served on the PNSQC board and various committees. He currently develops software for Proteus Technologies, where he holds the title of Distinguished Member of Technical Staff. Ray is a senior member of IEEE and a member of ACM.*

1. Introduction

This paper introduces a database client library, unimaginatively named *libdb*, that was deliberately designed and written to support unit testing of database applications. None of the ideas introduced in this paper is new, novel, or different. The combination, however, is unique. We hope that this case study can serve as a model for other C++ service libraries to support unit testing of applications.

Before examining the library in depth, a review of unit testing in general is in order.

1.1. Unit Testing

Unit testing, or testing in the small (Hetzel 1988), is testing of the smallest pieces of testable code (the unit). Typically, a unit is a single member function or free function because in C++ it isn't possible to execute a single statement outside of any function. Unit testing is a form of developer-led testing (McConnell 2004), and is just one of many layers of testing that a typical development team performs. The immediate benefactor of a test failure is the developer, who can quickly fix the code (or the test).

A key distinguishing feature of unit testing is isolating the Code Under Test (CUT). When testing a single class or a single member function of a class, you want to ensure that you are testing only that class or function, and not other classes or functions that the CUT relies on. Therefore, you want to design your code to reduce coupling between classes, which is a desirable trait independent of unit testing (Weinberg 2008, Briand 1997), and you want to be able substitute simpler classes to emulate the classes—called *mock* classes—that surround and support the CUT (Mackinnon 2001).

Further isolating the CUT is to remove all ties to the outside world, in particular, ties to external databases. The ideal unit test has no connections to outside services, no files to read or write, no user interaction, and no ambiguity. A test passes or fails, with no other states in between. One advantage of unit tests over other kinds of testing is that a unit test is often the best way to check error conditions. A unit test can set up artificial error states and verify that the CUT performs as designed.

Unit testing has been around for decades (Hetzel 1988), and long been considered one of the key best practices of software development. With the advent of extreme programming (XP) (Beck 2004) and test-driven development (TDD) (Beck 2002), unit testing gained renewed prominence. The value of thorough unit testing is now clear and well-supported by evidence. Furthermore, TDD has been shown to be effective as a technique to facilitate the delivery of high-quality software by ensuring the software is testable and well-tested.

1.2. Unit-Testing Frameworks

It is no surprise, therefore, to learn that a variety of libraries provide unit testing frameworks. In C++, we have CppUnit (2011), Boost.Test (2011), Google Test (2011), and many others. These frameworks share a number of common attributes. You can:

- define a suite of independent tests;
- write a test as a series of assertions about the state of the code and test environment;
- craft a test-runner program that can invoke one, some, or all of the tests; and
- set-up and tear-down the test environment uniformly for all tests.

Each framework has its individual advantages and disadvantages; the details are not germane to this paper. The examples in this paper use Boost.Test, but any framework is acceptable.

1.3. Mock Classes and Objects

A mock object (Mackinnon 2001) is an object that supports unit testing by replacing the CUT's neighbors with simple objects that record their interactions with the CUT and by providing mock results to the CUT. In this way, the CUT is isolated from other classes and functions, so you can be sure that your unit tests are truly testing your unit, and not some neighboring, or even distant, unrelated code.

For example, suppose the CUT opens a file, reads some text from it, and closes the file:

```
std::string readline(std::string const& path)
{
   try {
      File f(path);
      std::string result( f.read_line() );
      f.close();
      return result;
   } catch (PermissionError const& ex) {
      return "No permission";
   } catch (FileError const& ex) {
      return path + ": " + ex.what();
   }
}
```

How do you test such a function? Maybe you create a bunch of files, one empty, one with one line of text, another with multiple lines of text. What about error conditions? What happens if the file permissions do not permit reading? You can test these conditions by creating the right kinds of files. But what about other conditions?

What if an I/O error interrupts the read mid-line? Will your code catch the right exception and handle it the right way? That's where mock objects come in handy. Instead of using the real `File` class, use a mock implementation of `File`. The `MockFile` class lets you specify the exact behavior of the object, from exceptions that it throws to results that each function returns. So you can write tests such as the following, which arranges for the `File` class to throw an exception, and then tests that the `readline()` function catches and handles the exception correctly.

```
void test_readline_errror()
{
   MockFile.set_exception(FileError("mock error"));
   BOOST_CHECK_EQUAL(std::string("filename: mock error"),
                     readline("filename"));
}
```

Using the `MockFile` class instead of real files eliminates a dependency on external files. You don't run into the situation in which someone accidentally renames a file, and suddenly your tests stop working.

In languages such as Java, mock objects are commonly created by frameworks such as Easy Mock (Easy Mock 2011), jMock (jMock 2011), and others. The framework allows you to create mock objects according to your tests' requirements, rather than hand-crafting mock classes to emulate your classes.

Using mock objects in C++ is harder due to the richer compilation environment. It isn't enough to create mock objects on the fly, based on reflection. C++ also has templates, requiring additional complexity from any mock framework. On the other hand, some of the common C++ mock frameworks impose restrictions, such as the use of pure abstract interfaces, such as Automatic Mock Object for C++ (amop 2011) and Mock Objects for C++ (mockpp 2011). For these reasons, the use of mock objects and classes is less common in C++ than in Java.

Most C++ programmers cope by ignoring the issue. Well-designed code exhibits loose coupling between classes, so pure isolation of the code under test is not critical for successful unit testing. But when the code relies on external services, such as database access, the lack of mocks or other isolation techniques presents a serious challenge. With the dearth of general mock frameworks, it is incumbent upon the authors of third-party libraries to ensure their libraries support effective unit testing.

Unfortunately, such libraries are rare. This leaves the programmer in a quandary. You know that unit testing is important. You know that designing your application for testability will result in a higher quality design. You know that writing unit-tests while you write the code is an effective, efficient means of designing and developing software.

So when you need to use third-party C++ libraries, such as database client libraries, how do you proceed? The next section takes a look at how the author coped with this situation.

2. Database Libraries

The author recently needed a database library for writing a small database application. Such libraries are common—for example, OTL (2011), SOCI (2011), and SQLAPI++ (2011)—but not one of these libraries offers direct support for unit testing. Many developers face such restrictions by foregoing unit testing, relying instead on big-bang testing of the completed database application.

Forward-thinking developers may try to apply a modified form of TDD by incrementally developing the application, but this approach is still limited because the application must always interact with a real database. When the real application interacts with a real database, for example, how easy is it to test that the application rolls back correctly when the network connection is disrupted in the middle of the transaction? Even a library that supports an in-memory database is not designed to allow this kind of testing.

Another drawback to live-database testing is to ensure that multiple developers can run tests simultaneously. Thus, each developer needs a distinct database instance, and indeed probably needs multiple database instances. Another project that uses a full MySQL installation to perform “unit” tests takes a full sixteen minutes to run through a few dozen tests. This is after heavy optimization reduced the time from twenty-five minutes. Ideally, unit tests should run much faster than twenty-five or even sixteen minutes. A number of development teams require unit tests to pass prior to committing changes to the source code repository. In such an environment, unit tests that are burdened by real database connections would be infeasible.

Faced with the abundance of evidence that real unit-tests are necessary for effective development, the author sought an existing database library that lent itself to effective unit testing, but found none.

3. Mocking a Library in C++

When faced with the need to substitute a mock library for a real one, the author first tried to write a mock library with the same API. By directing the compiler to the mock headers instead of the real headers, the CUT was compiled for unit testing. When linked with the test-runner, a working unit-test was created. This approach has several drawbacks:

1. The CUT must be recompiled with the mock headers instead of the real headers. This doubles the compilation time, which may be a factor on large projects.
2. The CUT is not the real application code. By compiling with different headers, the mock library can introduce or mask errors. For example, we successfully hid a memory leak in our code by deleting an object in the mock library that was not deleted in the real library. (The mock library implemented the API incorrectly, but the error sneaked past our code review.)

3. The library may not lend itself to unit testing. Usually, this means the library is probably hard to code to, but the application developer may not have a choice of library. If the library had been designed with unit testing in mind, it would probably have had a better API.
4. Unit testing and production code must never be linked together. That is, suppose the application has two object files. If file A is compiled for unit testing and file B is compiled for production, if you mistakenly link files A and B, the results would be disastrous. In C++ terms, this would violate the one-definition rule (C++ 2003, [basic.def.odr]), resulting in undefined behavior. In practical terms, the result was usually a segmentation fault. Careful attention to build environments can prevent this kind of error, but in the author's experience, carefully designed and executed build environments are uncommon.

With a simple library, it may be possible to craft a mock library that has the same headers as the real library, but with different object files. In this case, recompilation is not necessary—only relinking. With a modern C++ library, however, this is rare. It is highly unlikely that a useful C++ library would entirely eschew templates and inline functions. It is even less likely that an accomplished C++ developer would want to use such a primitive library.

When faced with the database client library dilemma, the author decided that the best solution would be to write a new database library from scratch, this time, designing the library explicitly to support unit testing. The goals were as follows:

1. The library must facilitate unit testing. The application developer must be able to emulate a rich variety of error conditions that are hard to reproduce in an integration or system-level test.
2. The real application code must be unit-testable, without the need for recompilation. That is, when the database application is compiled, we should be able to use the same object files in a unit test that we do in production. We may choose to compile with more debugging and less optimization during testing, but the option is always available. In unusual, but not-rare-enough cases, we encounter compile defects that are difficult to track down if they never manifest during unit tests.
3. The library must be easy to write. The customer's requirements were for the database application, not a database application plus a reusable library. I had to be sure that writing the application and the library would together cost less than writing just the application (taking into account the extra time that would have been required due to the lack of unit testing).

I decided to use abstract interfaces for the library. The abstract interface would define the application programming interface (API). A mock implementation would be used for unit testing. Another concrete implementation would be used for the real database. We used only MySQL at the time, but other concrete implementations could support other databases.

4. Abstract Interfaces in C++

C++ can do anything Java can do, but sometimes it's harder in C++ than in Java. As we've already seen, implementing an on-the-fly mock object framework is harder to accomplish in C++ than in Java. But abstract interfaces are just as easy to write in C++ as in Java. C++ also has the advantage that interfaces do not have to be pure. Sometimes, it is advantageous to write a class that is mostly an interface, with just a hint of implementation thrown in. The database library has a couple of instances where impure interfaces provide clear benefits.

In C++ an abstract interface is a class with pure virtual functions, written as follows:

```
class this_is_abstract {
  virtual ~this_is_abstract();
  virtual void function() = 0;
};
```

The = 0 tokens tell the compiler that the function has no implementation, and that a derived class must override the function. The compiler prevents any code from constructing an instance of a class that has a pure virtual function. Instead, the developer must derive a concrete class that overrides and implements every pure virtual function. The compiler allows construction only of concrete classes.
The database library is defined primarily in terms of abstract interfaces. In particular, the `db::sql` and `db::statement` classes are abstract interfaces to represent the main interface to a SQL database and to a prepared statement, respectively. The database application code (almost) never refers to any concrete classes, using only the abstract classes.

The only time the application code names a concrete class is when it instantiates a class that derives from `db::sql` to kick things off. In our application, we have a small `main()` function that instantiates the necessary application objects as well as the `db::mysql::sql` object.

We never unit-test `main()`. Instead, we put all the application logic in application classes, and unit-test the application classes. Most of our applications have abstracted `main()` to the degree that we encapsulate it in a single macro that just names the main application class, and any concrete classes that we wish to inject into the application code. In this way, we have nothing to test in `main()`.

Unit tests create a `db::mock::sql` object and hand it off to the application code under test.

5. Design of libdb

Although we used only MySQL, I tried to keep the design flexible to allow for other concrete classes. I implemented the mock and the MySQL concrete implementations in parallel. Sometimes, the mock requirements drove design decisions, and other times MySQL drove design decisions. In both cases, I ensured that the decisions were consistent with the other implementation.

The main interface is the `sql` class. It represents a single database connection. An application can have any number of connections in any number of threads. The `sql` class manages transaction commitment and rollback, and it creates prepared statements, which are represented by the `statement` class. Figure 1 illustrates the class hierarchy.

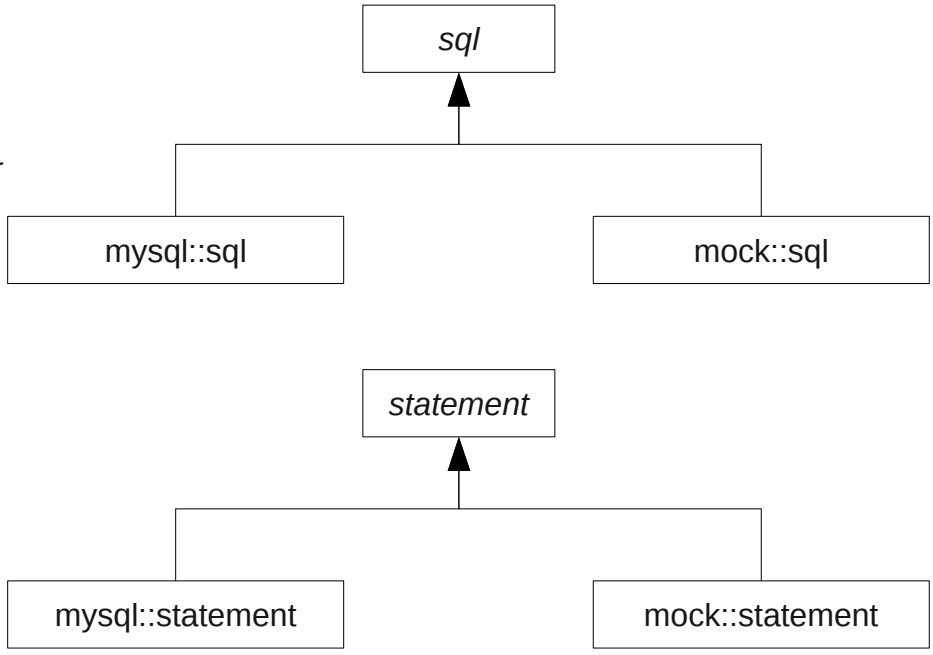

Figure 1: Class hierarchy

A `statement` object lets you bind rvalues to parameters, bind result lvalues, execute the statement, and fetch result records directly into the bound variables. (For those not conversant in C++-ese, think of an lvalue as a variable, and an rvalue as an expression. For those who are C++ experts, please forgive my oversimplification of these terms.) I made the decision early that the goal is to support specific applications, not general-purpose tools. Thus, there is no interface that returns a row as

an array of values to be queried by name or index. Instead, you must know exactly what the query will produce and bind lvalues of the correct type.

For example, suppose the *person* table contains `first` and `last` columns for the person's name, and `id` column for a primary key. The code to look up a single person's name might look like the following:

```
std::string const get_name_query(
      "SELECT first, last FROM person WHERE id = ?");

class person_app
{
public:
  person_app(db::sql& sql) :
  sql_(sql),
  get_name_(sql.prepare(get_name_query))
  {}

  std::string get_name(int id)
  {
    std::string first, last;
    get_name_.bind_param(id);
    get_name_.bind_result(first, last);
    get_name_.execute();
    if (get_name_.fetch())
      return first + ' ' + last;
    else
      throw not_found("No such person: ", id);
  }
private:
  db::sql& sql_;
  std::scoped_ptr<db::statement> get_name_;
};
```

The `bind_param()` and `bind_result()` functions are template functions that are implemented in `db::sql`. The application calls `bind_param()` once, if necessary, to bind all parameters (corresponding to the question marks, ?, in the text of the prepared statement). Each function argument corresponds to a ? in the query. The library performs basic error-checking to ensure the correct numbers of values are bound.

In the db::statement class, there are overloaded `bind_param()` functions, with a different number of arguments. They all follow the same form, such as:

```
template<class T1, class T2>
void bind_param(T1 const& arg1, T2 const& arg2)
{
  params p;
  p.push_back(param(arg1));
  p.push_back(param(arg2));
  bind_params(p);
}
```

The `bind_params()` function is virtual. The MySQL implementation constructs a `MYSQL_BIND` instance for each parameter and maps the C++ type to the equivalent MySQL type. The type `std::string` is mapped to a MySQL string, for example, and `std::vector<unsigned char>` is the libdb storage for BLOBs (Binary Large OBjects).

If the statement returns a result set, the application must call `bind_result()` to bind a variable to each column of the result set. The library automatically assigns values to these variables when the application calls `fetch()`. Binding of results is slightly more complicated than binding of parameters because strings and BLOBs have varying length. Thus, libdb must determine the result size, then resize the lvalue that was bound in the call to `bind_result()`, and only then it is safe to copy the result to the bound lvalue. Thus, the caller simply binds, say, a local variable of type `std::string` to a column of SQL type VARCHAR(255). The `bind_result()` function tells libdb about the variable. When libdb fetches a result row, it calls the variable's `resize()` member function to set the string to the correct size; then libdb copies the result string into the variable.

The `bind_result()` function looks very similar to the `bind_param()` function, except it constructs a `db::result` object for each argument instead of a `db::param` object:

```
template<class T1, class T2>
void bind_result(T1& arg1, T2& arg2)
{
  results r;
  r.push_back(result(arg1));
  r.push_back(result(arg2));
  bind_results(r);
}
```

The template functions determine the argument types and values, and prepare the parameter or result in a general way before passing a vector of `param` or `result` objects to `bind_params()` or `bind_results()`, both of which are private, abstract functions that perform the actual binding. In this way, the impurity of the `db::sql` class avoids duplication of common code in each concrete class. Using template functions in the abstract class improves clarity and results in higher quality of the database library.

In the mock library, a unit test prepares the mock results by storing responses. A response is a list of parameter values that libdb copies into result variables to mimic a database query. Because an application can make several queries inside a single function, the unit test must be able to store multiple rows in the mock result set. Furthermore, the test must be able to manifest various error conditions. The key is the `store_response()` function.

An example of a unit test for `get_name()` is as follows:

```
void test_get_name()
{
   db::mock::sql sql;
   person_app app(sql);

   sql.store_response(get_name_query, "Jane", "Doe");
   sql.store_response(get_name_query, db::mock::end_of_result_set);
   sql.store_response(get_name_query, db::mock::error);

   BOOST_CHECK_EQUAL(std::string("Jane Doe"), app.get_name(1));
   BOOST_CHECK_THROW(app.get_name(2), not_found);
   BOOST_CHECK_THROW(app.get_name(3), db::exception);
}
```

The `db::mock::sql` class implements `store_response()`, which is how a unit-test prepares the mock result sets for the application under test. Result sets are keyed by the text of the prepared statements. Each statement text has associated result records that the mock `fetch()` function fetches and binds to the result lvalues in the application code.

Thus, the mock `sql` object stores the strings "Jane" and "Doe". When the test runs, the mock `statement` retrieves the values and copies them to the first and last variables in the application code. As far as the application is concerned, it executed a query against a database and retrieved the first record of a result set. It doesn't know or care that the result set is just an array of values in the mock `sql` object.

The `store_response()` function can also store special values to represent the end of a result set or an error condition. When the mock statement object fetches one of these special values, it causes `fetch()` to return false (indicating the end of the result set) or to throw an exception. In pseudo-code, the mock fetch() function looks something like the following:

```
bool mock::statement::fetch()
{
  if (no more responses)
    return false;
  else if (response is pointer and pointer == mock::end_of_result_set)
  {
    ++response_iterator_;
    return false;
  }
  else if (response is pointer and pointer == mock::error)
  {
    ++response_iterator_;
    throw db::exception("mock database error");
  }
  else
  {
    copy_response(*response_iterator_, results_);
    ++response_iterator_;
    return true;
  }
}
```

Note how the mock statement keeps an iterator into the sequence of responses. Each time the iterator advances, it moves from one call to `store_response()` to the next. If the iterator points to a response of one element of pointer type, it checks the pointer value to determine whether that pointer matches a magic indicator. Otherwise, it copies the response to the bound result variable.

Here, the requirements of the mock library drove a critical design decision. We decided to store mock responses using the `param` class because it already implemented everything we needed for storing typed data. The only question was how to represent the magic indicators, such as `db::mock::error`. We could not use distinguished values because we would not be able to distinguish them from legitimate query responses. We decided that the best way to represent an indicator was to use pointers. We define a private object, and set `db::mock::error` to point to that object. The pointer value is unique and unambiguous. The requirement the mock library imposed on the `param` class, therefore, was to present pointer values.

The initial implementation of `param` stored a copy of the data. This seemed logical because it freed the developer from any concerns about lifetime of the bound value. In order to handle magic indicators, however, we had to change `param` to store pointers. This has the additional benefit of avoiding copying of strings and BLOBs, resulting in a performance gain, too.

In practice, lifetime issues are nonexistent. Our usage scenarios always place the binding with `bind_param()` in the same function as the calls to `execute()` and `fetch()`. Thus, it turns out lifetime issues were not an issue.

I wrote three test cases in one function for the sake of brevity. An alternative approach is to write three distinct unit tests for the three separate cases: found, not found, error. The advantage of having each test case in its own function is that most unit-test frameworks give you the ability to control which test functions to call, but not the ability to choose individual test cases within a function. Also, some unit-test frameworks (such as CppUnit) fail the test function if any assertion in the function fails. Test cases in the same function that follow the failure never run. Thus, you might not learn about a failure until after a complete editing cycle to fix another problem.

Sometimes, however, a unit test requires a lot of code to set-up the test. One setup may suffice for a series of related test cases. In that situation, it is often easiest to write a single function with multiple test cases. Whenever possible, however, do your best to isolate test cases in separate test functions.

6. Maintenance

Of course, software never stands still. Any change to the API requires updating the interface (such as `db::sql`) and the concrete implementation classes (`db::mysql::sql` and `db::mock::sql`). Fortunately, the costs are not triple because writing the abstract interface and mock implementations are usually much easier than writing the real, vendor-specific implementation. Once you know how the vendor-specific class works, it is usually easy to write the corresponding mock class and abstract interface.

The compiler even helps you by catching most omissions and mistakes, such as forgetting to implement a new member function or changing a function signature in a concrete class but not the interface.

It is feasible to write a tool that could extract the abstract interface for you, but we do not have such a tool. If we created many abstract interfaces, it would be worth acquiring or writing such a tool, but we don't do it often. So far, the burden of maintaining the abstract interface has been small and manageable.

Another concern for overhead is the increase in size of the code base. Compiling C++ is never fast; compiling three files instead of one is nearly three times slower. On the other hand, an earlier generation of the database application required instantiating a database server in order to run even the simplest test. By replacing many tests with true unit tests that use a mock database, the overall time required to compile and test the software was reduced by an order of magnitude (20 minutes to 2 minutes).

7. Legacy Code

An unsolved problem is tackling legacy code. Sitting in our legacy code base, for example, is an application that communicates over a socket. This application is no different than thousands of other socket-based applications. But we don't have a mock-socket class in our toolbox.

All the existing tests for the legacy code must run real application servers, and open real sockets. When a test takes 1 second instead of 1 millisecond (which would be a slow test using mock objects), it really drags down the test suite. A typical application can have hundreds of unit tests. This particular application takes about ten minutes to run through all its tests.

This application is due for some major changes in the near future, but we keep pushing the project down the to-do list because we are all hesitant to tackle it. Ironically, the problem is that we already have a socket library. If we didn't, we would probably design one from scratch using the principles outlined in this paper. Because we have a socket library and many developers have experience using it, we are reluctant to introduce another socket library, even if it does help us with our unit testing. Somehow, we have to figure out how to introduce mock sockets into this legacy socket library.

If we figure out how to solve the legacy code problem, we'll be back with another paper.

8. Conclusion

Developers test code as a way to deliver higher quality results to our customers. The proof of any technique, such as using a mock database library, is whether it helps improve quality or reduce costs. In this case, we delivered our application on time with zero known post-delivery defects in the database portion of the application. During development and integration, the database portion was so reliable, we were able to use the MySQL database as our oracle for testing other parts of the system.

The database was a small part of the overall project, and we did not break it out as a separate schedule item. Thus, we cannot definitively say that development of the unit-testable database library reduced costs. But it certainly had no negative impact on the schedule.

With no negative schedule impact, and zero defects discovered in the product after shipping, we consider this approach to be a complete success.

We certainly do not suggest that abstract interfaces are suitable for all C++ libraries. Many uses of C++ are especially sensitive to performance, and the overhead of even one virtual function can be too much (or one virtual function table in a class that is instantiated millions of times). One of the advantages of C++ over many other languages is the richness of the language, allowing for flexibility and choice in a way that Perl programmers can appreciate. There's more than one way to unit-test C++.

Other techniques for using mock libraries are possible. Each technique has its own drawbacks. We considered using compile-time substitution of a mock library, which complicates compilation and linking. We opted for abstract interfaces to avoid these issues and because we could live with the run-time overhead of a single virtual function call. For the database library, we are happy with our design decisions.

Adapting these techniques to existing libraries is much harder. If a library is not designed to be mockable, it can be hard or infeasible to introduce mock classes in support of unit testing. But other C++ service libraries can benefit from this approach, if you can find a way to introduce mock classes. Even if a different approach is taken, the idea of delivering a service library with a real and mock implementation can only benefit C++ application developers.

References

amop. 2011. *Automatic Mock Object for C++*. http://code.google.com/p/amop

Beck, K. 2002. *Test Driven Development: By Example*. Addison-Wesley.

Beck, K., and C. Andres. 2004. *Extreme Programming Explained, 2nd ed*. Addison-Wesley.

Boost.Test. 2011. *Boost test library*. http://www.boost.org/doc/libs/release/libs/test

Briand, L., P. Devanbu, and W. Melo. 1997. An investigation into coupling measures for C++. *ICSE '97 Proceedings of the 19th International Conference on Software Engineering*, pp. 412–21.

C++ standard. 2003. *ISO/IEC 14882:2003: Programming languages C++*.

CppUnit. 2011. *C++ port of jUnit*. http://sourceforge.net/projects/cppunit/

Easy Mock. 2011. *Easy Mock 3.0 Home*. http://easymock.org

Google Test. 2011. *Google C++ testing framework*. http://code.google.com/p/googletest/

Hetzel, B. 1998. *The Complete Guide to Software Testing, 2nd ed.* QED Information Sciences.

Jeng, B. and E. Weyuker. 1989. Some observations on partition testing. *TAV3 Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification,* pp. 38–47.

jMock. 2011. *An expression mock object library for Java*. http://www.jmock.org

Mackinnon, T., S. Freeman, and P. Craig. 2001. Endo-testing: Unit Testing with Mock Objects. *Extreme Programming Examined*. Addison-Wesley.

McConnell, S. 2004. *Code Complete, 2nd ed.* Microsoft Press.

Mockpp. 2011. *Mock Objects for C++*. http://mockpp.sourceforge.net

OTL. 2011. *Oracle, ODBC, and DB2-CLI Template Library*. http://otl.sourceforge.net/

SOCI. 2011. *The C++ Database Access Library*. http://soci.sourceforge.net/

SQLAPI++. 2011. *C++ library for accessing databases*. http://www.sqlapi.com/

Weinberg, G. 2008. *Perfect Software and Other Illusions About Testing*. Dorset House.