Unit Testing with Isolator++ Professional vs. Open Source Products

There's nothing most of us, C++ developers, detest more than unit testing. Unit testing is a time-consuming mess of a struggle to decipher and isolate dependencies and mock frameworks via manual coding (required even when using third-party testing frameworks). Had we wanted to become testers, we wouldn't have mastered the art of programming, now would we?

That is why an excellent mocking infrastructure is vital in today's R&D environments. It eliminates the tedious aspects of unit testing while helping us to maintain the highest standards of continuous development. As always, the question is which of the mocking infrastructures to choose. Well, Typemock just made this decision much easier.

The trouble with open-source mocking frameworks

Googling "open-source mocking frameworks for C++" yields almost 250,000 results, including many pros and cons lists reviewing the most commonly used open-source solutions. The problem is none of these frameworks covers all bases.

Open-source solutions are forward-looking and are most often applicable to greenfield projects with zero constraints. They're simply not designed to address the pains and struggles of dealing with technical debt. Performing unit testing on legacy code isn't a "sexy" or satisfying challenge for young coders, but it is fundamental, and no mocking framework is complete without it.

On top of this, open-source software usually lacks refinement, especially on Linux platforms. Obviously, finesse is not the most critical element of development platforms. Still, it sure is nice to have a user interface that demands as little interaction as possible, freeing us to deal with our main task, which is developing software.

Most open-source solutions have limited abilities or require too much work to create new tests. Some require derived classes for mocking, some have issues working

with non-virtual methods, and others get messy when dealing with multi-level mocking. Finding a genuinely comprehensive solution is almost impossible.

Where does Isolator++ stand in comparison?

Isolator++ can fake non-virtual methods, members, fields and out-parameters, sequencing behavior, and sticky behavior that other open-source mocking frameworks cannot.

Let's take the Google C++ Mocking Framework, for example. Google Mock testing requires you to derive the original class and manually specify its signature. So, to create a mock, you have to write code similar to this:

```
class MockMyService : public MyService
{
public:
    MockMyService()
    {
    }
    MOCK_METHOD1(action, void(std::string));
};
```

In the meanwhile, Isolator++ makes everything easier, allowing you to use the FAKE template instead of declaring a derived class like so:

```
MyService* myservice = FAKE<MyService>();
```

No need to change the code, make sure methods are virtual, and so on. Isolator++ provides a clear arrange/action/assert syntax that makes writing and maintaining the testing code clearer and faster.

Another popular open-source mocking framework is Trompeloeil. Like most of its competitors, Trompeloeil cannot mock C-like free functions, whose calls have to be dispatched to mock objects. With Isolator++, however, this can be done quite easily. The following method for example:

```
void SafeDelete(const char* filename)
{
       FILE* fp = fopen(filename, "r");
       if (fp == NULL)
       {
```

```
                throw ("File does not exist");
        }
        else
        {
                fclose(fp);
                remove (filename);
        }
```

Would look like this in Isolator++:

```
TEST_F(FileHandlerTests, SafeDelete_ThereIsFile_RemoveWasCalled)
{
        FAKE_GLOBAL(fopen);
        FAKE_GLOBAL(fclose);
        FAKE_GLOBAL(remove);

        FileHandler* handler = new FileHandler();
        handler->SafeDelete("AnyFile");
        delete(handler);

        ASSERT_WAS_CALLED(remove(_));
}
```

And it will also fake the global functions along the way.

Isolator++ makes dealing with technical debt easier, with features such as mocking static and private functions that enable you to mock complex legacy code. This case study, for example, shows how a leading supplier of eHealth systems uses Isolator++ to unit test their extensive legacy infrastructure with minimal code changes.