Unit-Testing in Multithreaded Applications
Franco Chiappori
bbv Software Services AG
Luzern, Switzerland
chiappori@bbv.ch

Automated unit testing is a powerful tool for detecting defects and improving software quality. Frameworks for unit tests are readily available for many programming languages. However, these frameworks are not suited for multithreaded software. This article describes how we have extended the basic driver - stub design to implement automated unit tests in a multithreaded environment. The concept presented here has been successfully used in multiple applications over a variety of platforms.

1 Introduction

1.1 Unit Testing and the Multithreaded World

Automated unit testing is a powerful tool for detecting defects and improving software quality. First of all is it easier to track down defects in small units than on application level. By accessing directly the interface of a unit, the software can be exercised in ways that would not be possible in a system context. Well-tested units ease integration of the application and improve overall software quality. However, manual unit testing means a lot of effort. Many applications today are developed in iterations and deployed in multiple releases over a long period of time. Unit testing should take place at least once during each iteration before integration. Automation is the only reasonable way to keep the effort needed for unit testing at an acceptable level.

Frameworks for automated unit tests are readily available for many programming languages. Especially in extreme programming, there is a lot of focus on automated tests. Also a related field, refactoring, uses automated unit tests in order to maintain and improve the software architecture over the lifetime of an application.

However, these frameworks are not suited for multithreaded software. First they usually assume that the unit under test has no dependencies on the rest of the application. In a usual real-time application this holds true only for the components at the bottom of a layered architecture. Higher-level components depend on other components, which need to be replaced by stubs for effective unit testing.

Second, the available frameworks have no synchronization means. In multithreaded applications, the units usually communicate asynchronously via messages. Furthermore, units themselves can consist of multiple communicating threads. A testing framework for this class of applications must take synchronization into account.

Third, tests in a real-time environment also need the concept of time. Since a response is only correct if it arrives in time, unit tests have to take this into account.

1.2 Overview of Article

This article describes how we have extended the basic driver - stub design to implement automated unit tests in a multithreaded environment. The stubs can be made either intelligent, i.e. they closely mimic the component they are replacing and take an active role in the tests. Or they can be made primitive, i.e. they only log actions and give back return values. We argue in this paper, that there are a
number of benefits in keeping the stub simple and concentrate the logic of a test in the driver. Ways of synchronizing the tests and taking the time into account are shown. Here the underlying architecture plays an important role and we give hints on what the requirements are and how to chose a suitable mechanism.

The concept presented here has been successfully used in multiple applications over a variety of platforms. We will share the experiences learned from these projects. Depending on the kind of application different issues arise. It makes also a difference whether an object oriented or a procedural paradigm is used in the software development. We will show what the differences are and how we dealt with them. In our experience it is also difficult to maintain the test cases over multiple releases. We briefly show how these problems can be addressed in part by generating the stubs automatically.

This is an industrial project paper targeted at software designer, test designer and test manager.

2 The Test Driver – Stub Design

A first challenge when performing unit tests is to effectively separate the unit under test from the rest of the system. This can be done using the Test Driver – Stub design. Since this paper relies heavily on this design, I will describe it here briefly. Readers already familiar with stubs can skip this section.

To aid my discussion, I have depicted a simple elevator controller in Figure 1. It shows six units organized in three layers. The arrows show the dependencies between the units. As a rule, units in higher layers should only depend on units in lower layers. In my example, the door unit depends on the CAN driver for communication with the hardware.

```
Control

Services

Trips

-------------------------------

Devices

Door

Pulley

----------------------------------

Drivers

CAN

STL

Figure 1: Simple elevator controller with three layers
```

To test the door unit, we first of all need a test driver, software that exercises the door unit, issuing commands to it and checking the results returned by it. But we also need a stub to replace the CAN driver unit. The stub has to provide the calls needed by the door unit, returning appropriate values. Figure 2 shows the resulting test bed.
There are several reasons why we would rather have a stub than the original CAN unit. First, in a real project the CAN driver might not be available when the unit test should be performed. Or it might be implemented but not fully tested. Bugs in the CAN driver might cause the test cases of the door unit to fail, such hindering an effective unit test. A stub also might have errors, but since it has only little functionality compared to the real thing, bugs are much less likely. Third, certain tests might be hard or even impossible to implement with the real CAN driver. With a stub it is much easier for example to simulate different communication failures and test the reaction of the door unit.

Many existing frameworks do not support stubs. They do not prohibit you from writing them, but offer no guidance in writing them and no place to fit them. Therefore these frameworks provide not suitable help for testing multithreaded applications.

3 Intelligent vs. Primitive Stub

When designing a unit test framework, we can choose to make the stubs either intelligent or primitive. Intelligent subs have built in logic, i.e. they mimic part or all of the unit they are replacing. Figure 3 shows an example of an interaction with an intelligent stub. A door open command is issued to the door, and the CAN stub actively waits 2 seconds before returning the appropriate answer “opened”. The test driver has a slightly longer timeout until it checks the door state.
While this approach might seem both natural and elegant, there are a number of issues that arise:

- How do we additionally test an error case (e.g. opened state is not received)
- Was test really correct (e.g. was open message sent to CAN stub)
- Test logic (e.g. timeout) is spread over multiple objects

Especially the last point proved in my experience to be a big drawback of intelligent stubs. This might be not obvious in our example with only one stub. In a more realistic case the door unit might depend on 5 to 10 stubs. Implementing and maintaining a test logic spread over 10 different objects can grow rapidly to a nightmare.

In contrast to this, a primitive stub contains no test logic. All it does is log the methods that are called and return previously defined return values. Figure 4 shows the same example as above with a primitive stub. The interactions might be slightly more complex, but one can clearly see that the test logic is completely contained in the test driver.
With a primitive stub it is very easy to implement additional test cases, (e.g. when the opened state is not received), and to check whether the correct action was performed on the stub. In addition, the pulley from Figure 1 could be tested with the same CAN stub as the door, without any extension to the stub. Finally, since the stubs contain no test logic, it is possible to generate them automatically. This we will discuss further in section 6.2.

4 Synchronization and Timing

Multithreading is a powerful concept used both in embedded and desktop applications. Different parts of the application run in a separate context, a thread, thus reducing the interactions and therefore the complexity of the application. Each thread can run as if it had the processor to itself. The underlying operating system is responsible to switch between threads and providing each thread with a separate processor context.

Our elevator example is most likely composed of a number of threads. If an open command is issued to the door unit, it is well possible that a thread switch occurs. Thus the calling thread can go on with its own business, while a separate door thread handles the open command. Figure 5 shows such an interaction in the context of our testing environment.
After issuing an open command, the test driver has to wait before it checks the result, allowing the thread switch to occur. This can either be done with a timeout, or by using a message or some other synchronization mechanism provided by the underlying operating system. Whatever way we chose, this is a basic functionality needed for the unit test, and should be provided by a framework. A number of requirements have to be satisfied. First of all, it should not be possible to block the test driver forever. Always there should be a timeout specifying the maximum time allowed for a reaction. Second, the mechanism should be easy to use. The developer should not need to access the operating system directly. This also allows easy porting to a different platform. Third, it should be possible to pass some information, e.g. which stub has received a command and what command it was. Existing frameworks usually provide no support for this kind of synchronization.

Related to synchronization is the question of timing. In many cases, a system has to react within a specified time. For some applications, like database reports, using too much time is a nuisance. In other applications, such as control systems, missing a deadline might be a fatal error. An automated unit test can use the features of an underlying operating system to check for the timely reaction of a unit. Although this is arguably much better than a human with a stopwatch, there are also limitations to the precision that can be obtained. A typical operating system can measure time in the order of a few milliseconds. A test case asking for more precision can only be implemented with additional hardware support.

If the target system does not provide any means to achieve the desired precision, you need additional, external hardware. This adds to the cost and complexity of the test setup, and makes automating the tests harder. If there are only a few such test cases, it usually pays to try and rework them, using only the features of the operating system, or the underlying hardware.

5 Target Needs

The unit test cases can be run both in a simulation environment, usually a workstation, and on the actual target. Existing frameworks are aimed towards a workstation environment. They usually rely on dynamic memory allocation and dynamic thread creation. Those features might not be present on an embedded target.

The first point, dynamic memory allocation, can be replaced with a specialization of the factory pattern. This is commonly used to create objects in an embedded environment. All the test drivers and stubs are contained in a test factory. During startup, one test factory is created in static memory.
therefore also creating the test drivers and stubs. In addition to creating these objects, the factory can also provide pointers to the test drivers and stubs.

![Diagram of Test Factory and Test Executor]

The second point, dynamic thread creation, can be replaced with a dedicated object, the test executor. This is a thread that runs one test driver at a time. It provides all the synchronization means needed by the test drivers.

There are other approaches than the ones presented here, which can also effectively overcome the limitations of an embedded target. Whatever you chose, make sure that it fits well into the existing architecture of your target. In addition to a test factory and a test executor some form of terminal output is needed, where the test results can be dumped. So far in my experience, it has never been a problem to route the output to a suitable place.

6 Experiences

6.1 Unit Tests in a C++ Elevator Controller

After looking at some existing unit test frameworks, we found that none fully solved the problems stated above. We then put together a simple C++ framework with factory, executor, and base classes for test driver and stub. The design closely followed the original JUnit framework. Thus we were also able to fit the framework well into an OS abstraction layer used in our architecture. For the first version of the framework we spent roughly one person-month.

We were able to gain first experiences with this framework when developing a new release for an elevator control system. This is a multithreaded C++ application aimed at a family of elevators, which has been evolving over several years now. A team of about 20 software and test engineers works on this control software. The software is developed with the aid of a simulation in a Windows environment, and is tested and deployed on a Motorola and an ARM platform.

When we first applied the framework, the software was undergoing major reworks in order to accustom new hardware components. For some time, the unit-testing framework was our only means for testing, since not all units were ready to integrate, and the hardware was not stable yet. In this early state unit testing proved valuable in detecting errors. The multithreading environment was effectively wrapped with the framework, and posed no major problems.

We found that unit tests are effective at finding functional defects of multithreaded applications. They are not so effective at finding defects specific to multithreading, such as deadlocks or starvation. A nice possibility is to test the timing of a unit, although the precision was limited by the underlying operating system.

However, one big problem was convincing the software developers to apply the framework. Schedule pressure was high, and testing was delayed or even skipped for some units. Not surprisingly, these untested units proved to be unstable during the first integrations. Some units were re-assigned to different developers in order to be tested. Only then the units became more stable, and we were free to focus on the real integration issues, i.e. problems at the interfaces between units.
Lesson learned: Software developers as a rule are hard to convince of the virtues of early unit tests. The paradigm of “first program all, then test all” seems to be still deeply rooted. Therefore a unit test framework has not only to solve to platform specific problems, but also it has to be very easy to use.

6.2 Evolving the Test Bed

Still working on the project above, we soon came across a different problem. During the reworks, the interfaces of the units were not fully stable. In effect we had to modify the stubs for every interface change. Even if the unit under test did not use those changed interfaces, we still had to do work, since the stub has to satisfy the entire interface. This might not be needed in other programming languages, but the strong compile time checks of C++ will not let you get away with an implementation that does not fit the interface declaration.

Soon, the burden of keeping all the stubs up to date was thrown over board, and many of the test beds fell into decay. Only if really needed to solve some quality problem, effort was put into maintaining the test bed.

In a quieter phase of the project, we looked at this issue in detail. To me, automated stub generation is the only easy way to solve the maintenance problem. We decided to push along this path and spent about one person-month for a small application that generates stubs that fit into our framework. Another month was used to enhance the framework and put together a little sample unit test. For the sample it worked fine. So far we have not gathered enough experience in a full project to present it here.

Lesson learned: Evolving the stubs along with the code means a lot of effort. If I had known this from the start, I would have looked more closely at tools that provide automatic stub generation. It could mean the difference between a living test bed and a heap of ill maintained stubs.

6.3 Acceptance Testing of Complex Unit

In one project we extended the framework to test a more complex unit of an elevator system, a frequency converter. A third party is developing this frequency converter. The tests are used for acceptance testing of new releases. We had to adapt the testing framework for this purpose. First, software developers are happy when they can use their programming environment to select the tests to be run, but test engineers are not. So we provided a terminal front-end to select and execute test cases. Further, the generated printout must not only show OK when all tests pass, but include more detail, showing each test case, either passed or failed. Thus the output provided by the test environment can be used directly in the test report.

Lesson learned: “One size fits it all” does surely not apply to unit testing. Depending on who the testers are and what the tests are needed for, more or less effort has to be spent on front-end and on reporting.

6.4 Unit Testing of Firewire Stack

In this project we had to port and test successive releases of a Firewire stack. The stack consists of a driver and a number of higher-level protocols, and estimates 300 kLOC of C code. The stack is provided to one of our customers by a third party. We port the stack to the customer’s hardware and execute acceptance tests on both driver and protocol level. Thanks to the automated tests we are typically able to port and test a new release within one week.

We did not use any stubs for the tests. Thus the maintenance effort for the tests could be greatly reduced. A quite stable interface on both driver and protocol level also helped. The tests were carried out on an ARM processor, with additional HW emulated in a FPGA. Rather surprisingly, the tests proved to be also effective for basic sanity tests of the emulated HW.
In the pure C environment we were not able to use our object-oriented framework. There are two major drawbacks of an procedural approach: you can not use a base class to inherit common functionality, and you can not use a class to group test cases and avoid namespace congestion. Instead we used normal functions for common functionality and naming conventions to group test cases and avoid conflicting names.

While designing our test suite, we came to some cases that could not be easily automated. It would have cost a considerable effort in external hardware and software to completely automate those test cases. In the end we decided to run those tests semi-automatically, with a human providing some input or checking some output. From a management point of view this was very reasonable, since we could reduce the overall effort. But personal experience taught me that these semi-automatic tests are very dull to execute. In a next project, I will do my best to convince management to push automation even further. This is only in part egoism. Dull work usually also means more errors, and errors should be avoided especially while testing.

*Lesson learned:* Semi-automatic tests can make sense to reduce the overall effort, but they are not the real thing. Try to push automation, not only to keep your people happy, but also to avoid lapses.

### 7 Outlook

Similar problems as the ones presented here haven been addressed with Mock Objects\(^vi\) or programmable stubs\(^iii\). There is always the question, whether a test should be automated or not. Some good answers can be found in an article written by Brian Marick\(^viii\).

Many commercially available unit test tools provide automatic stub and test driver generation to ease the creation and maintenance of test suites. Our next steps will be to explore this direction. The stub generation will be used in a current project, to see if it delivers on its promises.

### 8 Conclusion

Automated unit testing can be used in multithreaded, real-time applications successfully. There are some specific issues of multithreaded applications that can be addressed with a specialized framework. But we experienced also a number of problems that are not specific to these kinds of applications, but rather organizational or human in nature. The test framework has to be very easy to use, or chances are that it will not be used.

The field of unit testing is still evolving. It is agreed that it should be done, and there are a number of interesting approaches of how it can be done. Unit testing is used in real projects, and it provides real results. The big questions today are: Which approach suites which problems? How do we convince people to do it? How do we maintain the unit tests? Answers will be found, and will move software development one step closer to an engineering practice.

---

\(^i\) A collection of these frameworks can be found at http://xprogramming.com/software.htm  
\(^ii\) Martin Fowler. *Refactoring*, Addison-Wesley, 1999  
\(^iii\) A version of the described framework, placed under LGPL, can be found at http://www.bbv.ch  
\(^iv\) You might use this argument trying to get better timing means on your hardware. Good luck!  
\(^v\) Erich Gamma et all., *Design Patterns*, Addison Wesley, 1995  
\(^vi\) http://www.sidewize.com/company/Mockobjects.pdf  
\(^vii\) http://www.iplibath.com  
\(^viii\) http://www.testing.com/writings/automate.pdf