Introduction to Unit Testing

The term “Unit Testing” refers to the practice software developers have of writing code to test their own code 'units', or the smallest amount of software required to fulfill a 'unit' of desired functionality.

The goal of unit testing is to locate smallest piece of testable software, isolate it, then determine if that unit behaves exactly as you, or one of your "re-users” might expect.

Over time developers discover that unit testing is valuable because of the large numbers of unexpected defects unit testing typically unearths.

When maintained as a stand-alone deliverable in its own right, unit-test code creates a repeatable battery of tests. -Tests that can be run over and over to ensure operational continuity during both scheduled, and unexpected, software changes. Many have discovered that creating a series of ever-ready testing scenarios can help take their software quality to the next level.

Units and Objects

Defining a 'testable unit' has long been a source of contention between software developers. Indeed, because of the complexity required in staging developer-centric unit testing, for decades the very idea of delivering developer-level unit tests seemed foolish.

With the rise of Object Orientation however, not only has the definition of the 'testable unit' finally reached virtually a universal consensus, but with the acceptance of a common granularity numerous unit-testing tools and frameworks have arisen. Today, there are many developer conventions and testing tools to consider. Practices and products that can help maintaining feature continuity.

Unit Testing Approaches

While the list of code-testing products and technologies can be impressive, from an abstract point of view there are currently three (3) different Unit Testing approaches. For lack of a better way to describe each, we will refer to them herein as *Self-Test*, *Signature Generation*, and *Auto Test Generation*. We will discuss the first two in this section.

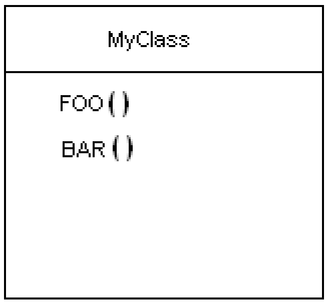
Self Test

Because much documentation treats the concept of unit testing as something new, we felt obligated to mention that software developers have been unit testing their code for decades. Indeed, in many respects our self-coined “Signature Generation” Technologies are considered by many seasoned developers to be pale imitations of the robust test cases professional developers need to create, even once using Signature Generation technologies. (Case in point: The ‘delegate’ testing model, as discussed later in this paper, was a common polymorphic framework testing technique in C++ well before the advent of Java.)

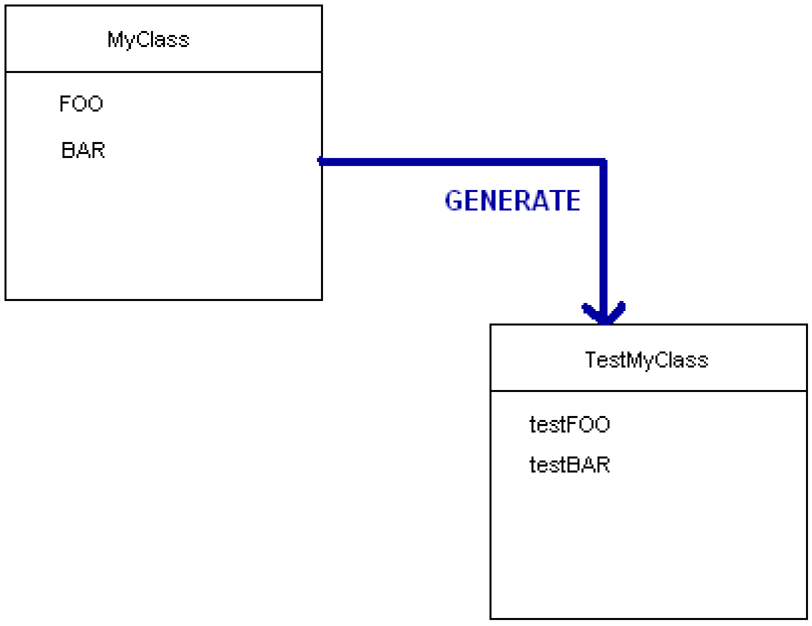
So even when Signature Generation tools are available, self-testing is the continuance of the practice of trusting in, and relying upon, software developers to both identify and exercise their own unit-software tests. The ability to quickly re-use extensively tested software components can decrease the need for extensive (if not expensive!) testing in highly visible software tools and / or applications. Indeed, many tools built from properly tested & maintained sources can be delivered with virtually zero-defects over time.

Signature Generation

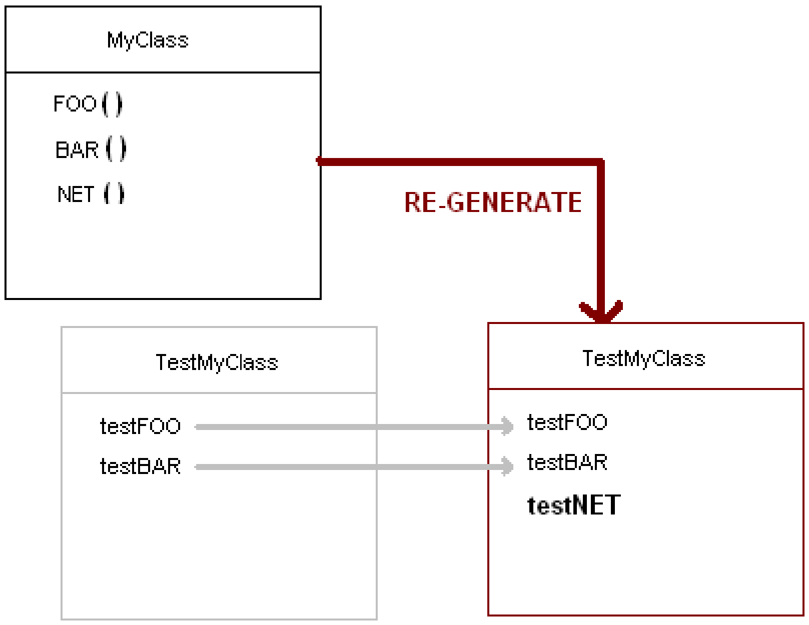
In the Microsoft software development arena, entering the auto-test-case-generation world will barrage us with terms like Classes, NUnit, Frameworks and / or Team Foundation Server (TFS.) Regardless, today the realm of automatic test-case-generation begins by generating software based upon a class:



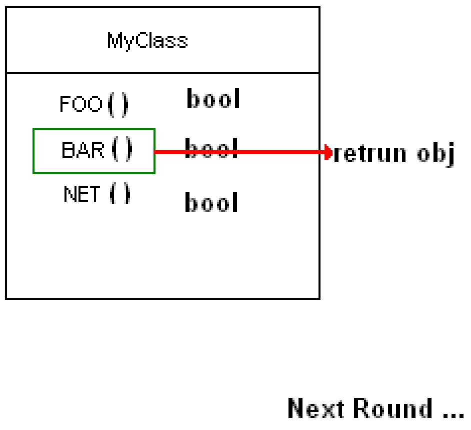
Once created, tools like NUnit / Visual Studio can parse the above 'unit' to create a **test** class:



While simple enough to understand, the early version of Java's JUnit testing framework (the pattern upon which .Net's NUnit framework was based) would perform the same operation over and over again: Class in – class out. In order to re-use code for a previous unit testing class, developers were required to copy their code into each newly-generated testing class:

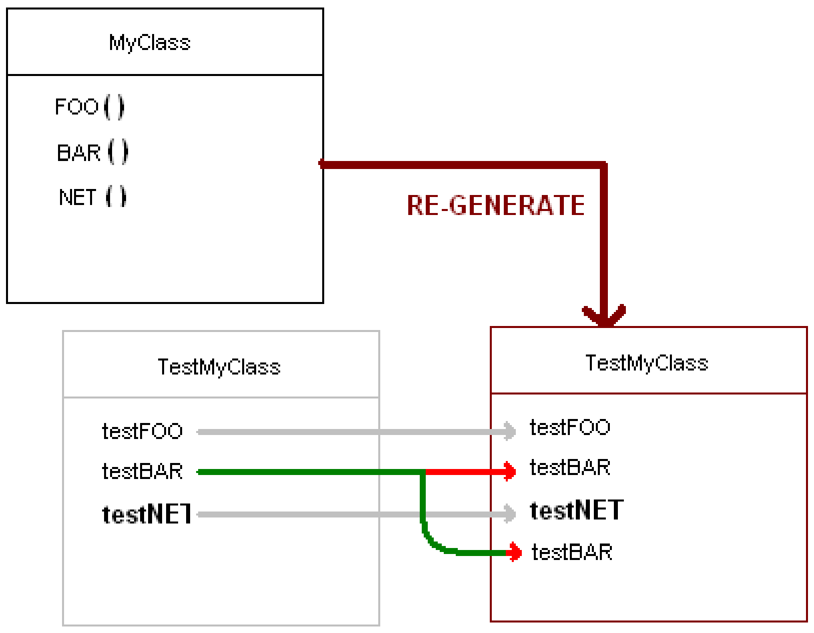


Test class generation in Visual Studio 2008 took a slightly different approach to unit testing. Rather than forcing the developer to copy previously authored testing-code, the integrated development environment (IDE) attempts to update our previously-generated test class for us:



In the above, notice how the result of changing the return result from a member function, without changing the name. Little information is given to the generator. -Depending upon the implementation language, legacy test cases might still operate (e.g. a pointer to an object can can often be compared to zero on non-zero), or, in more strongly-typed languages, simply fail to compile.

While Visual Studio represents a step forward in ease test case re-generation, whenever a tested class changes the parameters or return signature of a previously tested member (as illustrated above), human interventions can still be very much required. A good example can be seen when re-generating new test classes:



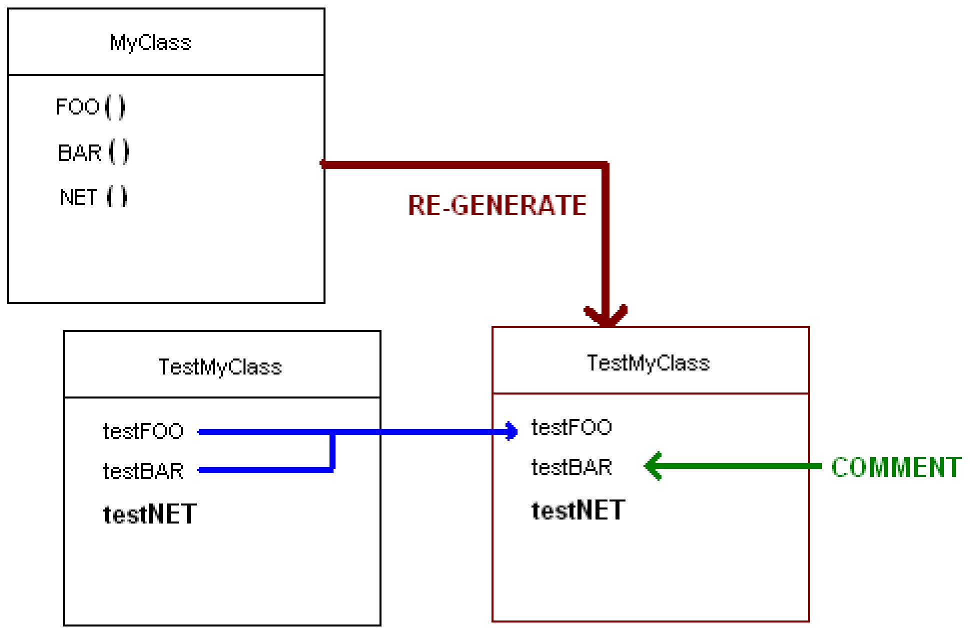
At the time of this writing, this writer has seen no attempt by any tooling to merge or otherwise intelligently re-use previously generated test cases. Human intervention may still be very much required.

The Need For Policies

Because automatically-generated test cases and the classes they test can be implemented in many different ways, those who have used Auto Testing tools extensively must also adopt policies and team strategies.

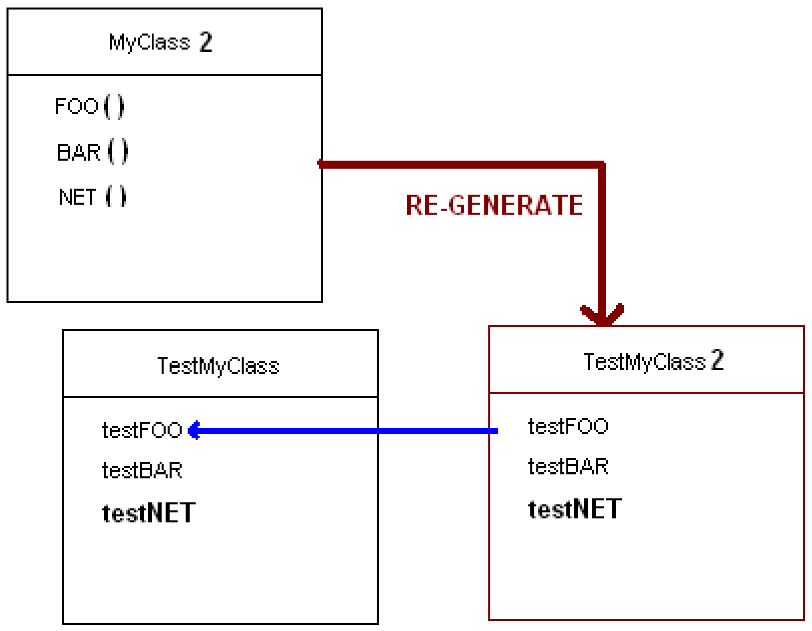
By way of example, one simple policy might be to agree that, whenever the use of auto-generated test-class-members are not used, that they never be removed. Rather, our policy might be to keep the class test-generator from perpetually re-generating test members.

Member regeneration can be inhibited in code by providing comments, faux 'if' statements, and / or opting to re-call members performing duplicate tests during test-time:



Regardless, many feel that making it a policy to manually re-use legacy and dispose of test cases is often sage advice. One need only recall the problems associated with the Y2K code conversions to understand how maximizing code re-use can make software far more maintainable over time.

If legacy test cases are to be maintained, then the practice of allowing *backward* re-use is a very maintainable policy. Enforcing conventions that, for example, require re-use to resemble the table of functions (much like a C++ *vtable*), a more rational re-use of legacy objects can be maintained:



Ensuring that previous test-classes are re-used in a more "polymorphic" manner is far easier to maintain.

(NOTE: *We will revisit a variation of the pattern above when we discuss the Moles* *delegate-testing-model in the next section.*)

Self & Signature Summary

While class-centric test-code generation provides on with a great "check list" of things one needs to test, unit testing retains much the same requirement of Self-Test. Indeed, in many respects, self-test has not (and in the opinion of this author, will not) ever fully replace the testing oversight of the code author, if not another just-as-savvy problem-domain developer.

So while auto-testing tools can indeed make life easier for many, it is not too difficult to understand how tenured communication-protocol developers (for example) will typically write better protocol unit-tests than do database developers who routinely write data-access software operating beyond ODBC Level 3 (and vice versa.)

Finally, while Visual Studio 2010 maintains the most integrated set of test-creation and reporting capabilities of any IDE which this author is aware, the ability to adapt and employ other unit-testing technologies (like NUnit) also has the potential to help your software-maintenance and / or inter-IDE quality improvement challenges.

Auto Discovery & Testing

While first generation unit testing code generators vie to out-do each other in finding ways to generate turn-key test-classes, to date there are no 'silver bullets'; No artificial intelligences that can fabricate test cases as well as the original software developer may, in their native problem domain.

Amongst the crop of second-generation test-code generators however we were pleased to recently review the operation of Microsoft Pex, a tool so much farther ahead of the JU*nit-ish* pack that we felt the need to cover the highpoint of what we learned about this tool in a class by itself.

Pex

As an automated testing tool, Pex provides more code automatically. By testing member function parameters and related close-relationships, the goal of Pex is to create more extensive test cases by exploring far deeper into the relationships, or associative paths between, any single unit of code.

Unfortunately, managing the code produced by Pex can be understandably more complex. Indeed, while a few hours is all one needs to spend with Visual Studio or NUnit to apply the bulk of what the tool can do for us, mastering Pex is cleanly a far more involved proposition.

For example, while using Pex, at one point we attempted to comment-out the testing of an array-laden member function, only to find that same function test being generated differently by a future unit test-case re-generation attempt. While such behaviors can be fine-tuned from within Visual Studio and / or XML, discovering how to mange array testing, and other test case generation options, can be a time consuming process!

In short, when we need to use other than the default Pex unit test cases, the Pex novice will find our previous code-only testing-decisions being ignored and / or over-written by the tool. With deeper testing-detection it seems, comes increased configuration complexity.

> NOTE: *Microsoft documentation warns that Pex will attempt to exercise every path in our code. For this reason, if our code is connected to external resources (such as databases or low-level devices), such resources should be disconnected prior to unit testing. -As values such as unexpected double-byte strings (can contain control characters, etc.), maximum integral values, and other stress cases are passed to a resource via any legacy interfaces, damage to system resources can occur.*

Because Pex attempts to exercises common failure points (like passing NULL objects to members) and deeper software inter-dependencies, it focuses more upon **destructive** unit testing. While deeper, more stressful unit testing has obvious benefits, forcing finer-grained services to do things like validate each and every parameter can have just as obvious performance trade-offs. Again, while Pex allows savvy Pex users to selectively disable it's deeper-testing stress-cases, some thought should be given as to the appropriateness of enforcing Pex findings on code where performance is far more importance than any anticipated re-use.

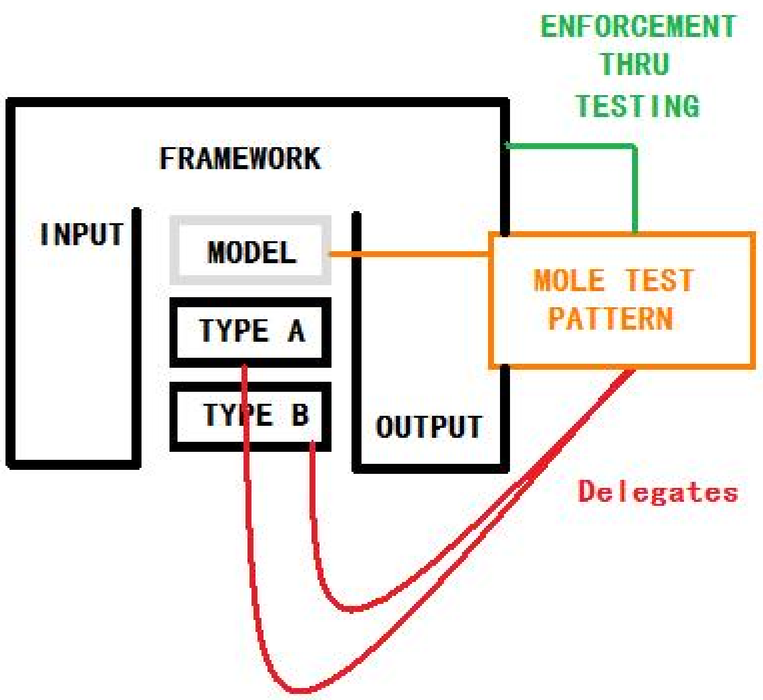
So despite the superior code generation and dependency detection capabilities of Pex, the final unit test remains only as good at the software developer using it. Indeed, in some cases (especially when maintaining a series of test-cases after test-code re-generation), in the hands of an inexperienced software developer, the misapplication of Pex could be downright dangerous.

Moles, Pex, and NUnit

Visual Studio 2010 is the preferred interface for Pex. By adding the ability to 'dig a little deeper' to discover common code-testing patterns, the Moles add-on for either NUnit or Pex affords us a more dynamic, delegate-based approach to unit testing.

> NOTE: *Using Moles with NUnit requires command-line access to the Pex Profiler from within the NUnit console session. Code generation can be done on-the-fly through the Moles.exe command-line tool. Because a plug-in and installer is readily available, using Moles with Visual Studio is easier… and recommended.*

Rather than creating a static signature-based test for each class, Mole's delegate-model allows us to pass "pointers to" our classes to test. When our classes-to-test support a common interface, Mole's delegate-testing-model allows a single test be applied to multiple classes via polymorphism. In short, several versions of a related code-hierarchy can be tested by a single, common, unit test.



Because the ostensible mission of Moles is to apply polymorphic patterns to testing (rather than the destructive testing and guard-condition stress often associated with Pex unit tests) the use of Moles will be far less dangerous and / or performance-disabling that an outright adoption of Pex. At a minimum, developers should want to become more familiar with using both the Visual Studio Team / ALM lineup, and Moles, to test their code.

Conclusion

In terms of cost, all of the unit testing tools – by virtue of either Open source or the ubiquitous MSDN Subscriptions – can be free. Furthermore, the author feels that developer time associated with learning enough to become productive with *Signature Generation* unit-test creation tooling should be negligible, as well.

For reasons of safety, convenience and inter-team test-case re-use, many believe that the application of the default Visual Studio Unit Testing will provide the safest, easiest, and most important unit-test tooling. Given the superior reporting and statistics generating capabilities of Visual Studio 2010, developers interested in learning more about ways to improve things like code cohesion and coupling between their units should run… not walk … to begin understanding and applying 2010 heuristics, as well as tactical unit testing, to their software.

Next, as software developers arrange their units into abstract models, adding Microsoft's Moles to any default tooling will ensure that interface integrity can be assured with minimal test-code creation.

By way of honorable mention, Open Source promises to ever continue to add innovative reporting and other features to tools like NUnit. Because NUnit is easier to maintain than TFS, the stand-alone nature of NUnit and its add-on community can make this tooling attractive. -Amongst other things, combining NUnit with a 'test-first' mentality can make projects easier to track. --When applied to contractual software evolution over time, even software / service deliveries between platforms, customers, and forts could become much easier to track, demonstrate, and manage.

Finally, while more tenured software developers should be encouraged to experiment with Pex, the general application of Pex should be discouraged until extensive stress / acceptance-testing needs require otherwise. For example, when accepting software from unknown developers, the application of the default Pex-generated unit tests can be a good way to insure that robust models, and / or a highly reusable set of interface objects, have been provided.

I hope that you have found this paper informative. While the task of maintaining a reasonable set of on-demand test cases is far from new, the application of a little Object Orientation know-how, combined with a little experience with modern tools can make test-case maintenance far easier. If not enjoyable!