The ‘8 Commandments’ For Choosing a Unit Testing Solution

Companies who perform unit testing on a regular basis are perceived to be more reliable, professional and advanced. But what do you need to consider before choosing a unit testing solution? Typemock, the pioneers of easy unit testing solutions, have developed the ‘8 commandments’ below as a guide for ensuring you select a unit testing solution that is right for your development.

1 Thou shalt not waste time on the learning curve

When choosing a unit testing solution, you will want one that will require minimal time for implementation. It may be worthwhile to time a new developer within your team with the framework to get an accurate idea of how long it takes to get started. For example - how long will it take them to write the first three tests for some class in your system? Is the API clear and simple? Is there a single point of entry in the API? Is there clear guidance on what to do at each step of the way? How often, if at all, do you need to check the docs and tutorials? How easy is it to look for the next step when you're not sure what to do? Some tools offer guidance within the IDE, while some provide extensive help. Some don't do either.

2 Thou shalt not waste time fixing your tests.

How resistant to change do you need your tests to be? Different frameworks provide varying levels of change resistance (when production code changes). Change resistance can be measured according to the number of tests needed to modify for a single piece (method) of changed production code. Does the tool support these production changes without affecting the tests? Do the frameworks support recursive fakes? Is it non-strict by default or does it throw exceptions on unexpected interactions by default? How does it handle method overloads (if your production code changes to call a different overload?). How are arguments verified? Are all arguments ignored? Are actual values used for expectations? All these things affect the fragility of your test. The more fragile it is, the less change resistant it is. If your production code changes a lot you will need to take this into account.

3 Thou shalt design your code the way you need it

Your project might be a greenfield project (fresh new code) or have a lot of legacy code. Not all isolation frameworks were designed, or have the same support for all (or any) legacy code scenarios. If your unit tests need to be written against legacy code (existing code without tests): See if your production code contains static constructors, internal or private classes that might need to be faked. Does the framework support faking them? Does your code instantiate objects directly all over the place, or does it use a dependency injection framework or factory of some kind that would need to be faked? Make sure the framework you choose supports these scenarios, or that you have the time needed to refactor your code for testing such cases.

4 Thou shall make your test code readable

How will your tests look when you use the framework? Test code is still code and needs to be readable. When you jump into it to debug, you need to be able to see at a glance what you're testing. Are your tests messy? Does the framework cause your tests to be longer than a page? Can you understand the expected behavior of the code under test if the framework is involved? Can your team members understand without a great deal of explanation what you are using the framework for in each test? Can you still write test code according to accepted industry best practices such as AAA (arrange-act-assert) when using this framework? Not all frameworks support the easier-to-understand AAA tests, for example. It is important that the solution you choose guides you in writing tests, (incorporating the best-practices in test writing), which will make the process of unit testing quicker and easier.

5 Thou shall not need to replace tools for the sake of the tool

How well would the tool integrate with your current coding environment and ecosystem? Can you use the framework from your various test runners (TestDriven.NET, Resharper, MS Test etc.) or in combination with profiling and code coverage tools? Some frameworks incorporate profiling, and it is important to see that they work effectively with other profilers and runners. Can you run your tests with code coverage or with other profiling technologies? How well does the framework integrate with various versions of Visual Studio? Do you need it to support VS 2005? VS 2008? What .NET versions should it support? Some of the frameworks require .NET 3.5 and up.

6 Thou shall not forsake interaction testing

Do you need the ability to verify interactions between objects in your tests? Some frameworks may not have the built in ability to verify interactions, requiring you to provide your own manual flagging mechanisms (hand rolled mocks).
For complicated interfaces this can grow cumbersome and result in an un-maintainable piece of code. Can you verify method calls between the code under test and its 3rd party dependencies? For example, that a SharePoint method gets called at the end of a test.

7 Thou shall not settle for an incomplete solution

A complete solution is not only one that meets all your requirements for the tasks it needs to carry out, but one that provides you with support from the moment you show an interest in seeing a demo until long after the product has been implemented in your system.
Ask yourself, what kind of technical support do you require? How fast do you need to get a response to a question if you are not sure how to accomplish a specific task with the tool? How many people are on staff at the vendor who makes the framework and at what hours of the day? Is support provided by a company or an independent individual?
A company is often more invested in supporting you than an individual – or they stand to lose money. What is the company's main expertise? Are unit testing and agility at its core business? If so, the support team is more likely to well versed be on these subjects. There is nothing more frustrating for a development project manager who is trying to complete a project on time and in budget then being delayed because the team cannot get the support or answers they need from the solutions experts in real time.

8 Thou shall consider the solution’s TCO

Often, one of the biggest factors in any development project is the Total Cost of Ownership for the tool. The TCO should consider price, time and effort required to work and implement the tool in your project or organization.
Compare the following:

- Price of tool
- Time to get started and reach RTM test (readable, trustworthy, maintainable)
- Time to write tests
- Time to fix test because of production change (change resistance)

Typemock have recently released Isolator 2010, a unit testing tool that helps abstract dependencies from the tested code, supports Visual Studio 2010 and .Net 4.0 and is the only solution for testing SharePoint applications, as well as Test Lint Pro, the first solution to enable entire development teams to collectively implement best practice in unit test writing. These solutions ensure faster, quality code development and are integral parts of Typemock’s unit testing solutions catalogue.