Implementing an Efficient Test Execution Process

By José Cornado

Quality as a strategic advantage

Creating great software, on time and under budget, is the goal of any software development team. All too often, however, one of these parameters is compromised by inefficient development practices. Since budget tends to be a hard number, and software companies are reluctant to miss an advertised release date, software quality is frequently the place where compromises are made. This document illustrates how a shift in the approach to quality management is integral to a winning economic model for software development, and offers a realistic solution.

The Problem

"Software companies have to change. We need to be more like the auto industry -- quicker and better at development."

--Henning Kagermann, CEO of SAP

According to the National Institute of Standards and Technology (NIST), 80% of the software development costs of any project are spent on identifying and fixing defects. According to the Mythical Man Month, this figure is around 50%. If we take into consideration that MMM was written 25 years ago and the NIST figure is from 2002, it is safe to assume that over a 20-year period, software development has become less efficient and less competitive. Clearly, the process is broken and we need to fix it. It is our opinion that high transactions costs are the culprit. For example:

- Inefficient practices and methodologies: high levels of manual testing and inadequate QA automation tools and frameworks.
- Test development efforts: the costs associated with writing new tests and maintaining existing tests.
- Under-utilization of assets: CPU and memory utilization (% of capacity used per unit of time) have been historically low. In fact, to our knowledge, this key metric has been largely ignored.
- Hidden costs: recent studies by Cap Gemini Ernst & Young have found that up to 65% of the in-memory image of an application belongs to code developed by third parties. The net effect is that up to 52% of your budget is spent running somebody else’s software. Are you getting rewarded for that effort?
- “Reaction Time”: the time it takes QA infrastructure to forward quality deviations to the appropriate personnel.

The Typical Situation…

When formulating a QA plan, the first step that many companies take is to buy one of the expensive QA tools based on “capture-and-replay” architecture. However, these tools fail in their objective of test automation. The best evidence of this failure is the number of jobs, requiring exactly the same skill set, that are currently being outsourced. An inherent technical problem is that the ‘replay’ part of the architecture barely copes with changes to the GUI from one cycle to the next, which creates a maintenance nightmare. In addition, the capture and replay approach is entirely dependent on the PC architecture. We are approaching a stage where applications should run not just on the PC platform, but also on a variety of devices.

Another approach that software companies take is hiring a group of black-box/manual testers to tackle the growing code base. This step has pernicious financial effects because, in essence, companies have to increase their QA investment over time instead of reducing it. The “increasing returns” principle that economists apply to software development is broken in this case.

Let us illustrate: manual testing is a slow, inaccurate task. It relies on good use case writing and strict manual execution of those use-cases. Too often, there is not enough time to write use cases that cover the entire code base, and the repetitive nature of the task leads to human error in execution of the cases that are available. Additionally, as new code is developed, an increase in output capacity is required from a resource that has a natural, fixed capacity: two hands, two eyes, and one mind. This forces companies to pay manual testers over and over to repeat the same tasks, and then hire new black-box/manual testers to do the same for the new code. This could not be further from the “increasing returns” principle, and in fact is a financially losing practice.

…Just Gets Worse

To keep things simple, only one aspect of the “identifying and fixing bugs” process was included in the previous section: testing of the end product through the GUI. However, the GUI represents just a small portion of the entire code base. A considerable share of the code must be tested using ‘white box’ testing methodologies and frameworks.

Some of these frameworks integrate with the IDE, creating a convenient environment for developing tests. Unfortunately, they are cumbersome to use, slow to execute, and incompatible with popular methodologies like “Refactoring”. With schedule often driving the development process, developers simply do not have the time to create and maintain unit tests, thus hampering the white box effort.

One methodology that tries to remedy this problem is Extreme Programming (XP), which advocates "pair programming" or a 1:1 ratio of developer to unit tester. Microsoft Corporation has implemented XP programming over the past few years: it is reported that they have a 1:1.1 developer to tester ratio, and include XP friendly testing tools in their Visual Studio product line. However, even with their vast resources, they have not been able to claim success with this process, as evidenced by the recent delay in the release of Vista due to quality concerns.

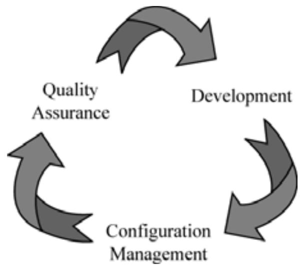
When white box testing does not cover the code base appropriately, it is still necessary to rely at least in part on the black box testers and /or capture and replay solutions. At that point, the complexity of the situation explodes:

- At least three sets of QA skills and people are now involved: the developers/white-box testers, "capture-and-replay" tools/testers, and the manual testers.
- The work of these groups needs to be well coordinated. With each group working at its own pace, quality can be uneven and create a negative user experience.
- Dependence upon a particular programming language for development and unit testing creates a situation where porting to a more appropriate or advanced programming language requires testing the tests, not just the code.

Our View: It is a Matter of Creating Value

It is easy to see that current practices in software QA are inefficient and do not create value. In order to reverse this trend, it is necessary to look at the areas where resources are being wasted. Beyond the obvious waste of money and time represented by the methods discussed above, there is another metric that is ignored throughout the industry: wasted hardware capacity. In our experience, average CPU and memory loads of Development and QA hardware rarely reach 15%.

Most software R&D groups exhibit the following production cycle:



Concentrating on the Configuration Management (CM) to Quality Assurance part of the cycle, assume that CM finishes building the product images at 7:00 pm. Based on standard personnel schedules, QA will not get to work until the next morning, so for at least 12 hours those machines accomplish nothing. If QA infrastructure uses 4 PCs (running Pentium 4 processors at 3 GHz per second), this represents 518,400 GHz gone to waste. How many tests could have been executed with even 50% CPU and memory utilization?

Now, take a closer look at development. Development computers are taxed only during the few seconds that code is being compiled. The rest of the time, work is done primarily in a text editor, leaving the bulk of the computing capacity unused.

In The Mythical Man Month, it is asserted that every change in the source code has a 30% likelihood of breaking something else downstream, and the only way to be absolutely sure that quality does not deteriorate is to re-test everything repeatedly. The book goes on to state that this is not economically feasible. If you consider *just* man-hours, we agree that this is true. But by more efficiently using the millions of CPU cycles or memory bytes per second that are currently wasted, we can create a situation where it becomes economically feasible to ensure excellent quality in real time, even in the face of significant source code changes.

Our Solution

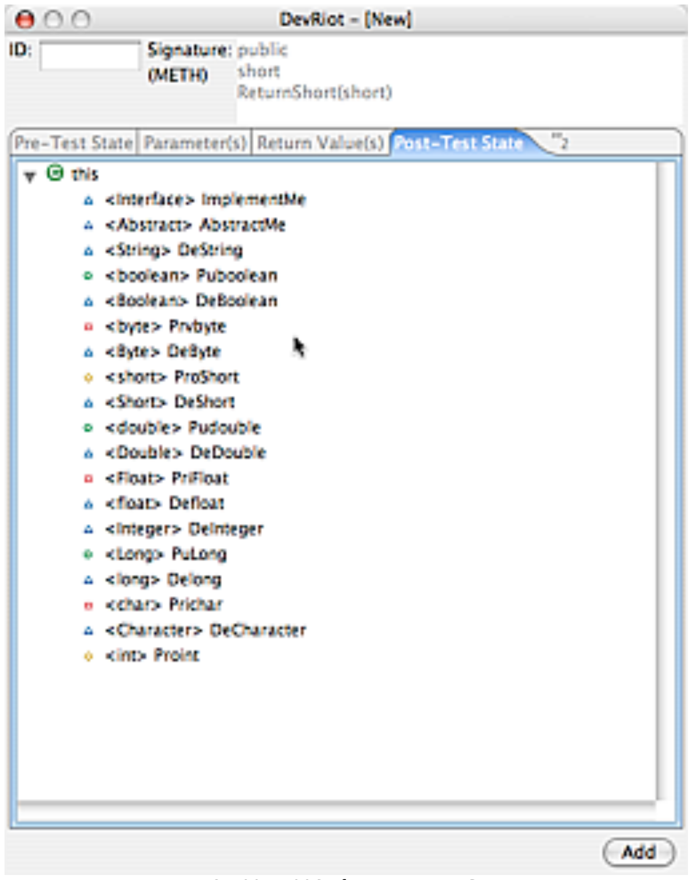
The focus of our research and development has been to create the tools necessary to implement a test architecture that constantly keeps the code base at near release quality, but does not impede the momentum of code development. This architecture can be implemented at the individual contributor level and the team level in such a way that:

- Execution, analysis, and reporting of results are completely automated and at the user’s fingertips.
- Total test matrix coverage is constantly and automatically executed with each build released from CM.

The biggest benefit of this approach is that personnel can concentrate on high value tasks, such as writing code or tests, rather than wasting time on chores that can be more efficiently executed by computers.

How It Works

DevRiot takes a streamlined approach. We work with a concept called “instrumentation unit” (IU) which is an abstract representation, like abstract syntax, of a test. IUs are automatically provided and tailored for each method, and to create a usable test the developer simply needs to fill in the inputs, and expected outputs (Figure 1.1).



ID:
Signature: (METH) public short ReturnShort(short)

```
▾ this
      <Interface> ImplementMe
      <Abstract> AbstractMe
      <String> DeString
      <boolean> Puboolean
      <Boolean> DeBoolean
      <byte> Prvbyte
      <Byte> DeByte
      <short> ProShort
      <Short> DeShort
      <double> Pudouble
      <Double> DeDouble
      <Float> PriFloat
      <float> Defloat
      <Integer> DeInteger
      <Long> PuLong
      <long> Delong
      <char> Prichar
      <Character> DeCharacter
      <int> ProInt
```

Figure 1.1

The IU maintains language, platform, and device portability, increasing its usability over time. For instance, it is possible to create IUs while developing a GUI application in Swing/Java. Subsequently those same IUs can be used against a Winforms/Pocket PC/.Net application. Similarly, it is possible to create IUs while developing a library in C# and use them against a library written in Java or C++.

The lifetime of the IU may be long or short, depending on the needs of the user. The IUs are automatically grouped together to form a test suite and the suite is then instrumented and executed on the fly or stored to disk. If the suites are stored to disk, we provide a testing server, which can gather them for further processing at the integration level after CM finishes processing the build.

DevRiot's approach to GUI testing is similar (Figure 1.2):

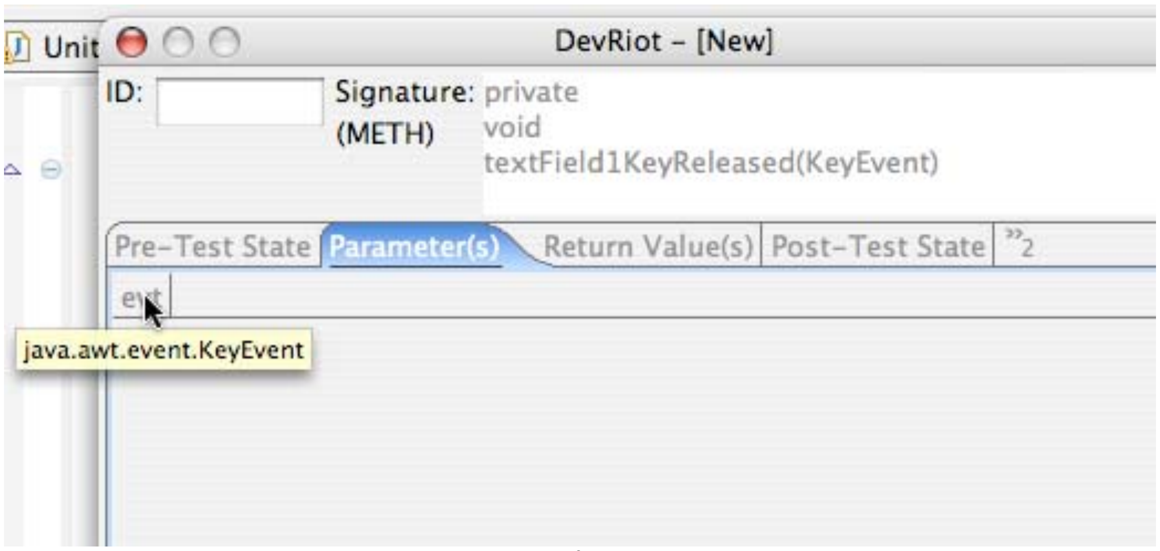
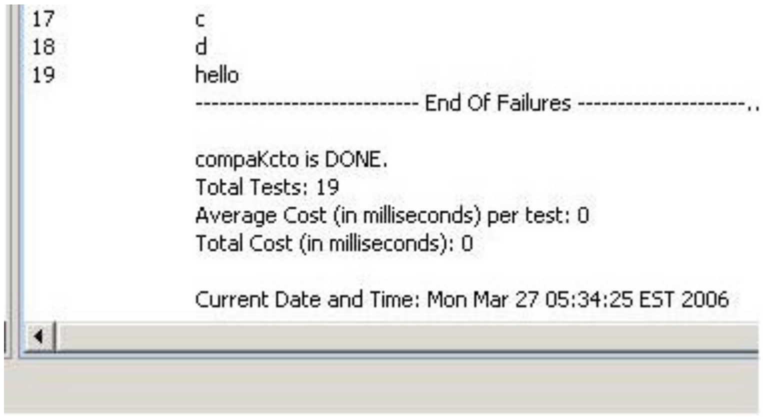


Figure 1.2

After the user “fills in the blanks”, DevRiot takes care of the rest. It works closely with the internal build process of the IDE in a way that is transparent to the user. It kicks in only when the IDE tells it that building was successful, generating the IUs in the specific language currently in use. Once the IUs are executed, any unexpected results are reported automatically before the user even has time to notice that the IDE has finished building (Figure 1.3, for a demo in QuickTime format click on image).

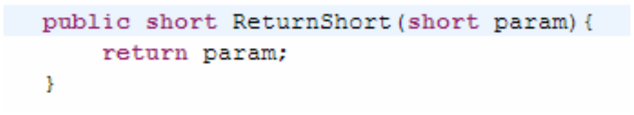


```
17              c
18              d
19              hello
                --------------------------- End Of Failures ---------------------..

                compaKcto is DONE.
                Total Tests: 19
                Average Cost (in milliseconds) per test: 0
                Total Cost (in milliseconds): 0

                Current Date and Time: Mon Mar 27 05:34:25 EST 2006
```

Figure 1.3

DevRiot provides execution speeds that are orders of magnitude faster than any competitor. This frees up time to test downstream dependencies with the most up-to-date code, reducing the likelihood that defects make it into the integrated build.

To illustrate the difference between current white box testing solutions and DevRiot, consider this simple code example:



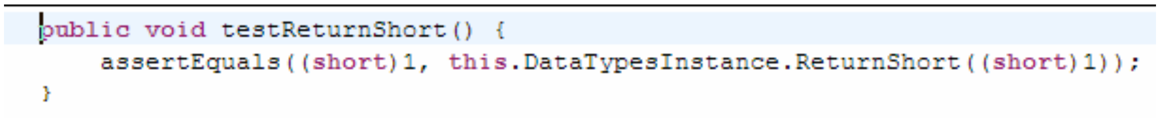
```
public short ReturnShort(short param){
    return param;
}
```

The only concern for a user should be the test inputs and expected outputs (for example, 1 and 1). In a more explicit way, in this example, the user should just need to type two *keystrokes*: 1 for the input and 1 for the expected output.

If we were to use any of the available unit testing tools, we would have to start with these steps (taken directly from Junit documentation):

1. Create a subclass of TestCase.
2. Create a constructor that accepts a String and passes it to the superclass.
3. Add an instance variable for each part of the fixture.
4. Override setUp() to initialize the variables.
5. Override tearDown() to release any permanent resources you allocated in setUp.

After this preliminary work, the user will need to type at least *71 more keystrokes* to create *each* test case for this method:



```
public void testReturnShort() {
    assertEquals((short)1, this.DataTypesInstance.ReturnShort((short)1));
}
```

If you compare this effort to DevRiot's two-keystroke solution for the same problem, you will see that it is time to approach QA in a new way. (Click here for a demo)

Summary

If we assume that value is created when the outputs from development (source code) and from QA (tests) interact to create a product ready for consumption, we can say that by increasing the frequency and speed of this interaction, i.e. reducing the friction in the process, a software company can avoid economic inefficiency, and run an "increasing returns" operation.

DevRiot utilizes an abstract representation of a test to create a simple but rigorous approach to test automation. With minimal effort by the user, it achieves everything the competitors can only promise. Test coverage is complete and fully automated. Resources are used efficiently. Risk is minimized. The bottom line is maximized. Most importantly, quality is a strategic advantage instead of an impediment to growth and success.