Observing Unit Test Maturity in the Wild

Ilja Heitlager, Tobias Kuipers, Joost Visser
Software Improvement Group, Amsterdam, The Netherlands

Presentation at the 13th Dutch Testing Day, hNovember 29, 2007.

Abstract

This talk provides an anecdotal account of unit testing maturity and its evolution in Dutch IT organizations as we have observed it during the last 7 years in our IT consultancy practice.

During these years, we have performed independent Software Risk Assessments on scores of mission-critical information systems, typically in the domains of finance, government, and logistics. A wide variety of technologies were used to construct, and to test these systems.

Unit testing, specifically, was observed by us as an initially unknown phenomenon, which has been met by all sorts of misunderstandings, but is nonetheless gaining popularity.

Based on our observations, we distinguish 5 evolutionary stages. In contrast to the 5 maturity levels of the Testing Maturity Model (TMM, presented at the 12th Dutch Testing Day, 2006), these stages are not normative or prescriptive, but simply descriptive of actual practice in the Dutch IT industry. Call our study *unit testing phenomenology*, if you will.

Stage 1: no unit testing

Not surprisingly, unit testing still does not enjoy widespread uptake. Many systems are devoid of unit tests. Some systems have unit testing only in name: a unit testing framework is actually used as an instrument for functional or integration testing, or simply as a sandbox for experimental code.

In case of legacy systems built in technologies that predate the rise of unit testing, the unavailability of a unit-testing framework may be a valid reason for not doing unit tests. But what about the many C# and Java systems that we observed in recent years for which no unit testing was employed?

Resistance to unit testing comes in many shapes and forms. Often, unit testing is perceived as a cost factor only, while their benefits go unnoticed. “We had no time to do unit testing”, “the client does not pay for it”, are among the reasons we have heard. In other cases, the team simply missed unit-testing skills and felt unable to retrofit unit tests to an existing system, or did not know how to organize their code into testable units. Not to be discounted is the argument never voiced explicitly, that unit tests improve maintainability, which cuts into hours billable in future.

Stage 2: Unit tests, but no coverage measurement

Unit testing is increasingly adopted, but the accompanying best practice of measuring unit test coverage lags behind. In these systems, a unit-testing infrastructure is in place, unit testing is encouraged or even demanded, but the lack of coverage measurement means that there is no shared awareness of how good the unit testing is.

Stage 3: Coverage measurement, but not approaching 100%

Once coverage measurement is monitored, full coverage is generally not attained easily. Code may need reorganization to be made testable, testing GUI code remains out of reach, and advanced unit testing skills are lacking.

Stage 4: Approaching 100%, but no test quality measurement

Eager to close the gap and approach full coverage, some teams produce unit tests of low quality. Testing large chunks of code at a time, or writing test code without any assert statements are among the phenomena we observed. These unit tests attain coverage, but miss some of the main benefits of true unit tests: traceability of errors and documenting value.

Stage 5: Measuring test quality

To ensure that the drive for coverage does not hurt the quality of the unit tests, the quality of tests must be made tangible. Several measures for unit test quality can be employed, such as the ratio between asserts in a test and the number or linear execution paths of the production code it covers. Or the number of production statements covered by a single test. Systems in this evolutionary stage, we must admit, have been rarely observed in the wild.

Short profile of the authors

Ilja Heitlager is responsible for Software Risk Assessment operations at the Software Improvement Group (SIG).

Tobias Kuipers is co-founder and CTO of SIG. Tobias is co-designer of tools and methods for software quality and risk assessments.

Joost Visser is R&D lead at the SIG, in charge of developing a new generation of tools and methods for software quality and risk assessment.

All three authors provide CIO-level management consulting for a large number of public institutes and multi-national companies.