Software Testing Best Practices

Ram Chillarege
Center for Software Engineering
IBM Research

Abstract:

This report lists 28 best practices that contribute to improved software testing. They are not necessarily related to software test tools. Some may have associated tools but they are fundamentally practice. The collections represent practices that several experienced software organizations have gained from and and recognize as key.

1. Introduction

Every time we conclude a study or task force on the subject of software development process I have found one recommendation that comes out loud and clear. "We need to adopt the best practices in the industry." While it appears as an obvious conclusion, the most glaring lack of it's presence continues to astound the study team. So clear is its presence that it distinguishes the winners from the also-ran like no other factor.

The search for best practices is constant. Some are known and well recognized, others debated, and several hidden. Sometimes a practices that is obvious to the observer may be transparent to the practitioner who chants "that's just the way we do things." At other times what's known in one community is never heard of in another.

The list in this article is focused on Software Testing. While every attempt is made to focus it to testing, we know, that testing does not stand alone. It is intimately dependent on the development activity and therefore draws heavily on the development practices. But finally, testing is a separate process activity -- the final arbiter of validity before the user assesses its merit.

The collection of practices have come frohm many sources -- at this point indelibly blended with its long history. Some of them were identified merely through a recognition of what is in the literatures; others through focus groups where practitioners identified what they valued. The list has been sifted and shared with increasing number of practitioners to gain their insight. And finally they were culled down to a reasonable number.

A long list is hard to conceptualize, less translate to implementation. To be actionable, we need to think in terms of steps -- a few at a time, and avenues to tailor the choice to our own independent needs. I like to think of them as Basic, Foundational, and Incremental.

The Basics are exactly that. They are the training wheels you need to get started and when you take them off, it is evident that you know how to ride. But remember, that you take them off does not mean you forget how to ride. This is an important difference which all too often is forgotten in software. "Yeah, we used to write functional specification but we don't do that anymore" means you forget to ride, not that you didn't need to do that step anymore. The Basic practices have been around for a long time. Their value contribution is widely recognized and documented in our software engineering literature. Their applicability is broad, regardless of product or process.

The Foundational practices are the rock in the soil that protects your efforts against harshness of nature, be it a redesign of your architecture or enhancements to sustain unforeseen growth. They need to be put down thoughtfully and will make the difference in the long haul, whether you build a ranch or a skyscraper. Their value add is significant and established by a few leaders in the industry. Unlike the Basics, they are probably not as well known and therefore need implementation help. While there may be no textbooks on them yet, there is plenty of documentation to dig up.

The Incremental practices provide specific advantages in special conditions. While they may not provide broad gains across the board of testing, they are more specialized. These are the right angle drills -- when you need it, there's nothing else that can get between narrow studs and drill a hole perfectly square. At the same time, if there was just one drill you were going to buy, it may not be your first choice. Not all practices are widely known or greatly documented. But they all possess the strength that are powerful when judiciously applied.

The next sections describe each of the practices and are grouped under Basics, Foundational, and Incremental.

2. The Basic Practices

- Functional Specifications
- Reviews and Inspection
- Formal entry and exit criteria
- Functional test - variations
- Multi-platform testing
- Internal Betas
- Automated test execution
- Beta programs
- 'Nightly' Builds

Functional Specifications

Functional specifications are a key part of many development processes and came into vogue with the development of the waterfall process. While it is a development

process aspect, it is critically necessary for software functional test. A functional specification often describes the external view of an object or a procedure indicating the options by which a service could be invoked. The testers use this to write down test cases from a black box testing perspective.

The advantage of having a functional specification is that the test generation activity could happen in parallel with the development of the code. This is ideal from several dimensions. Firstly, it gains parallelism in execution, removing a serious serialization bottleneck in the development process. By the time the software code is ready, the test cases are also ready to be run against the code. Secondly, it forces a degree of clarity from the perspective of a designer and an architect, so essential for the overall efficiencies of development. Thirdly, the functional specifications become documentation that can be shared with the customers to gain an additional perspective on what is being developed.

Reviews and Inspection

Software inspection, which was invented by Mike Fagan in the mid 70's at IBM, has grown to be recognized as one of the most efficient methods of debugging code. Today, 20 years later, there are several books written on software inspection, tools have been made available, and consulting organizations teach the practice of software inspection. It is argued that software inspection can easily provide a ten times gain in the process of debugging software. Not much needs to be said about this, since it is a fairly well-known and understood practice.

Formal Entry and Exit Criteria

The notion of a formal entry and exit criteria goes back to the evolution of the waterfall development processes and a model called ETVX, again an IBM invention. The idea is that every process step, be it inspection, functional test, or software design, has a precise entry and precise exit criteria. These are defined by the development process and are watched by management to gate the movement from one stage to another. It is arguable as to how precise any one of the criteria can be, and with the decrease of emphasis development, process entry and exit criteria went out of currency. However, this practice allows much more careful management of the software development process.

Functional Test - Variations

Most functional tests are written as black box tests working off a functional specification. The number of test cases that are generated usually are variations on the input space coupled with visiting the output conditions. A *variation* refers to a specific combination of input conditions to yield a specific output condition. Writing down functional tests involves writing different variations to cover as much of the state space as one deems necessary for a program. The best practice involves understanding how to write variations and gain coverage which is adequate enough to thoroughly test the

function. Given that there is no measure of coverage for functional tests, the practice of writing variations does involve an element of art. The practice has been in use in many locations within IBM and we need to consolidate our knowledge to teach new function testers the art and practice.

Multi-platform Testing

Many products today are designed to run on different platforms which creates the additional burden to both design and test the product. When code is ported from one platform to another, modifications are sometimes done for performance purposes. The net result is that testing on multiple platforms has become a necessity for most products. Therefore techniques to do this better, both in development and testing, are essential. This best practice should address all aspects of multi-platform development and testing.

Internal Betas

The idea of a Beta is to release a product to a limited number of customers and get feedback to fix problems before a larger shipment. For larger companies, such as IBM, Microsoft and Oracle, many of their products are used internally, thus forming a good beta audience. Techniques to best conduct such an internal Beta test are essential for us to obtain good coverage and efficiently use internal resources. This best practice has everything to do with Beta programs though on a smaller scale to best leverage it and reduce cost and expense of an external Beta.

Automated Test Execution

The goal of automated test execution is that we minimize the amount of manual work involved in test execution and gain higher coverage with a larger number of test cases. The automated test execution has a significant impact on both the tools sets for test execution and also the way tests are designed. Integral to automated test environments is the test oracle that verifies current operation and logs failure with diagnosis information. This is a best practice fairly well understood in some segments of software testing and not in others. The best practice, therefore, needs to leverage what is known and then develop methods for areas where automation is not yet fully exploited.

Beta Programs

(see internal betas)

‘Nightly’ Builds

The concept of a nightly build has been in vogue for a long time. While every build is not necessarily done every day, the concept captures frequent builds from changes that are being promoted into the change control system. The advantage is firstly, that if a major regression occurs because of errors recently generated, they are captured quickly. Secondly, regression tests can be run in the background. Thirdly, the newer releases of software are available to developers and testers sooner.

3. Foundational

- User Scenarios
- Usability Testing
- In-process ODC feedback loops
- Multi-release ODC/Butterfly profiles
- Requirements for test planning
- Automated test generation

User Scenarios

As we integrate multiple software products and create end user applications that invoke one or a multiplicity of products, the task of testing the end user features gets complicated. One of the viable methods of testing is to develop user scenarios that exercise the functionality of the applications. We broadly call these User Scenarios. The advantage of the user scenario is that it tests the product in the ways that most likely reflect customer usage, imitating what Software Reliability Engineering has for long advocated under the concept of Operational Profile. A further advantage of using user scenarios is that one reduces the complexity of writing test cases by moving to testing scenarios than features of an application. However, the methodology of developing user scenarios and using enough of them to get adequate coverage at a functional level continues to be a difficult task. This best practice should capture methods of recording user scenarios and developing test cases based on them. In addition it could discuss potential diagnosis methods when specific failure scenarios occurs.

Usability Testing

For a large number of products, it is believed that the usability becomes the final arbiter of quality. This is true for a large number of desktop applications that gained market share through providing a good user experience. Usability testing needs to not only assess how usable a product is but also provide feedback on methods to improve the user experience and thereby gain a positive quality image. The best practice for usability testing should also have knowledge about advances in the area of Human Computer Interface

.

In-Process ODC Feedback Loops

Orthogonal defect classification is a measurement method that uses the defect stream to provide precise measurability into the product and the process. Given the measurement, a variety of analysis techniques have been developed to assist management and decision making on a range of software engineering activities. One of the uses of ODC has been the ability to close feedback loops in a software development process, which has traditionally been a difficult task. While ODC can be used for a variety of other software management methods, closing of feedback loops has been found over the past few years to be a much needed process improvement and cost control mechanism.

Multi-Release ODC/Butterfly

A key feature of the ODC measurement is the ability to look at multiple releases of a product and develop a profile of customer usage and its impact on warranty costs and overall development efficiencies. The technology of multi-release ODC/Butterfly analysis allows a product manager to make strategic development decisions so as to optimize development costs, time to market, and quality issues by recognizing customer trends, usage patterns, and product performance.

"Requirements" for Test Planning

One of the roles of software testing is to ensure that the product meets the requirements of the clientele. Capturing the requirements therefore becomes an essential part not only to help develop but to create test plans that can be used to gauge if the developed product is likely to meet customer needs. Often times in smaller development organizations, the task of requirements management falls prey to conjectures of what ought to be developed as opposed to what is needed in the market. Therefore, requirements management and its translation to produce test plans is an important step. This practice needs to be understood and executed with a holistic view to be successful.

Automated Test Generation

Almost 30% of the testing task can be the writing of test cases. To first order of approximation, this is a completely manual exercise and a prime candidate for savings through automation. However, the technology for automation has not been advancing as rapidly as one would have hoped. While there are automated test generation tools they often produce too large a test set, defeating the gains from automation. On the other, there do exist a few techniques and tools that have been recognized as good methods for automatically generating test cases. The practice needs to understand which of these methods are successful and in what environments they are viable. There is a reasonable

amount of learning in the use of these tools or methodologies but they do pay off past the initial ramp up.

4. Incremental

- Teaming testers with developers
- Code coverage (SWS)
- Automated environment generator (Drake)
- Testing to help ship on demand
- State task diagram (Tucson)
- Memory resource failure simulation
- Statistical testing (Tucson)
- Semiformal methods (e.g. SDL)
- Check-in tests for code
- Minimizing regression test cases
- Instrumented versions for MTTF
- Benchmark trends
- Bug bounties

Teaming Testers with Developers

Its been recognized for a long time that the close coupling of testers with developers improves both the test cases and the code that is developed. An extreme case of this practice is Microsoft, where every developer is shadowed with a tester. Needless to say, one does not have to resort to such an extreme to gain the benefits of this teaming. This practice should, therefore, understand the kinds of teaming that are beneficial, and the environments in which they may be employed. The value of a best practice such as teaming should be therefore more than just concept. Instead it should include guidance on forming the right team while reporting the pitfalls and successes experienced.

Code Coverage

The concept of code coverage is based on a structural notion of the code. Code coverage implies a numerical metric that measures the elements of the code that have been exercised as a consequence of testing. There are a host of metrics: statements, branches, and data that are implied by the term code coverage. Today, there exist several tools that assist in this measurement and additionally provide guidance on covering elements not yet exercised. This is also an area that has had considerable academic play and has been an issue of debate for a couple of decades. The practice of code coverage should therefore carry information about the tools and the methods of how to employ code coverage and track results from the positive benefits experienced.

Automated Environment Generator

A fairly time-consuming task is the setting up a test environments to execute test cases. These tasks can take greater amounts of time as we have more operating systems, more versions, and code that runs on multiple platforms. The shear task of bringing up an environment and taking it down for a different set of test cases can dominate the calendar in system test. Tools that can automatically set up environments, run the test cases, record the results, and then automatically reconfigure to a new environment, have high value. The IBM Hursley Lab has developed a tool called DRAKE that does precisely. This best practice should therefore capture the issues, tools, and techniques that are associated with an environments set up, break down, and automatic running of test cases.

Testing to Help Ship on Demand

This is an idea from Microsoft where they look at the testing process as one that enables late changes and accommodates market pressures. It changes the role of testing to one of providing excellent regression ability and working in late changes that still do not break the product or the ship schedule. This really amounts to a philosophical view of testing, placing it in a different role yielding new ramifications for the entire development process. We cite this as a best practice to recognize that there may be areas where such a conceptual framework necessitate a very reactive testing practice. The practice ought to identify how to work this concept into organizations and products in specific markets. It may have applicability in the E-Commerce world, where there is far greater customer interaction and competitive pressure.

State Task Diagram

This practice captures the functional operations of an application or a module in the form of a state transition diagram. The advantages of doing so allows one to create test cases automatically or create coverage metrics that are closer to the functional decomposition of the application. There are a fair number of tools that allow for capturing Markov models which may be useful for this practice. The difficulties have usually been in extracting the functional view of a product which may not exist in any computable or documented form and producing the state transition diagram. One of the automated test generation tools called Test Master from Teradyne
actually uses state task diagrams for the generation of functional test. This practice has possibly more than one application and the keepers of the practice need to capture the tools, the methods, and its uses.

Memory Resource Failure Simulation

This practice addresses a particular software bug, namely the loss of memory because of poor heap management or the lack of garbage collection. It is a fairly serious problem for many C programs in Unix applications. It also exists on other platforms and

languages. There are commercial tools available to help simulate memory failure and check for memory leaks. The practice should be generic and develop methods and techniques for use on different platforms and language environments.

Statistical Testing

The concept of statistical testing was invented by the late Harlan Mills (IBM Fellow who invented Clean Room software engineering). The central idea is to use software testing as a means to assess the reliability of software as opposed to a debugging mechanism. This is quite contrary to the popular use of software testing as a debugging method. Therefore one needs to recognize that the goals and motivations of statistical testing are different fundamentally. There are many arguments as to why this might indeed be a very valid approach. The theory of this is buried in the concepts of Clean Room software engineering and are worthy of a separate discussion. Statistical testing needs to exercise the software along an operational profile and then measure interfailure times that are then used to estimate its reliability. A good development process should yield an increasing mean time between failure every time a bug is fixed. This then becomes the release criteria and the conditions to stop software testing.

Semi-Formal Methods

The origin of formal methods in software engineering dates a couple of decades. Over the years it has made considerable progress in some specific areas such as protocol implementation. The key concept of a formal method is that it would allow for a verification of the program as opposed to testing and debugging. The verification methods are varied, some of which are theorem provers, while some of them simulation against which assertions can be validated. The vision of formal methods has always been that if the specification of software is succinctly captured it could lead to automatic generation of code, requiring minimal testing.

In the practice there has been much debate on the viability of semi-formal methods and to a large extent the industry ignores it. However, one must recognize a very key contribution from IBM's Hursley Lab. This is where the kernel of CICS was implemented after a formal specification written in Z. A semi-formal method is one where the specifications captured may be in state transition diagrams or tables that can then be used for even test generation. IBM's Zurich Research Lab has done some work in this area and very successfully used this for protocol implementations. The best practice in semi-formal methods ought to capture our experience and also guide places where such applications may be viable.

Check-in Tests for Code

The idea of a check-in test is to couple an automatic test program (usually a regression test) with the change control system. Microsoft has been known to employ

such a system very well. This allows for an automatic test run on recently changed code so that the chances of the code breaking the build are minimized. In fact, Microsoft's change control system and build are supposedly set up such that unless the code passes the test, it does not get promoted into the next build.

Minimizing Regression Test Cases

In organizations that have a legacy of development and of products that have matured over many releases, it is not uncommon to find regression test buckets that are huge. The negative consequence of such large test buckets is that they take long to execute. At the same time, it is often unclear as to which of these test cases are duplicative providing little additional value. There are several methods to minimize the regression tests. One of the methods looks at the code coverage produced, and distill test cases to a minimal set. One must note that this method, though attractive, does confuse a structural metric with a functional test. Never the less, it is a way to implement the minimization.

Instrumented Versions for MTTF

An opportunity that a beta program provides is that one get a large sample of users to test the product. If the product is instrumented so that failures are recorded and returned to the vendor, they would yield an excellent source to measure the mean time between failure of the software. There are several uses for this metric. Firstly, it can be used as a gauge to enhance the product's quality in a manner that would be meaningful to a user. Secondly, it allows us to measure the mean time between failure of the same product under different customer profiles or user sets. Thirdly, it can be enhanced to additionally capture first failure data that could benefit the diagnosis and problem determination. Microsoft has claimed that they are able to do at least the first two through instrumented versions that they ship in their betas.

Benchmark Trends

Benchmarking is a broad concept that applies to many disciplines in different areas. In the world of software testing, we could interpret this to mean the techniques and the performance of testing methods as experienced by other software developers. Today there is not an avenue to regularly exchange such information and compare benchmarks. This best practice could be initiated by benchmarking across IBM Labs and then advance the practice to include a larger pool with competitors and customers.

Bug Bounties

We have heard that bug bounties were used in Microsoft and we know that they have been used in IBM during the 10X days. Bug bounties refers to our initiatives that charge the organization with a focus on detecting software bugs. At times providing

rewards too. The experience states that such effort tend to identify a larger than usual number of bugs. Clearly additional resource is necessary to fix the bugs. But the net result is a higher quality product.