{Coders}
Kitchen

How to develop high-quality software

Improve quality through test-focused software development

Table of Contents

1 Overview

Every year, organizations commit themselves to key objectives. Oftentimes, this is achieved via metrics-based performance goals that may include quality goals and leveraging best practices to streamline their business process. Ultimately, measuring the impact these objectives have had on the organization involves some form of testing and reporting. Savvy employees know to ask for a list of what they will be judged on well before they are reviewed – yet when it comes to developing new software products, defining goals and objectives for testing is often over-looked.

When organizations are designing a physical product, much time is spent on designing the manufacturing process, and the automated testing of each completed item. No one would design a product that could not be manufactured or tested efficiently. The issues of manufacture and test are solved prior to the design being completed. A design is not useful if it cannot be manufactured economically and consistently.

What if the same approach were taken with the development of software? Suppose, as you started to design a software system, you thought about ways to make the software "manufacturing" and "testing" as efficient as possible. By doing this, you ensure a software product that has high quality, and a process to support continued high quality over the lifecycle of the product.

"Building technical systems involve a lot of hard work and specialized knowledge: languages and protocols, coding and debugging, testing and refactoring." – James Garrett

2 Building testable software

The idea that software should be designed with "testability" in mind may require something of a paradigm shift for your organization. During initial design and prototyping the focus is almost always on functionality and performance. While those items are important, they are useless if you produce a low-quality application with lots of bugs that is hard to maintain.

Some "soft" approaches you can take include: ensuring the completeness and correctness of requirements, architecting the application with software testing in mind, adopting an easy-to-understand coding style, and implementing a "real" peer review process.

3 Scrutinize software requirements

Ensuring that software requirements are complete will prevent many defects. Consider writing the specification for a "square root" function. In actuality, this is a simple function deep in some math library. But this "simple function" can be poorly implemented if the requirements are not complete. Consider this specification:

- The square_root() function shall return the square root of its input for all valid values.

A better specification would be:

- The square_root() function shall return the square root of its input for all valid values, and 0 for all invalid values. Valid inputs are positive 32-bit floating point numbers, zero and positive infinity. Invalid inputs are negative numbers, negative infinity, and NaN.

Based on this specification, an architect can easily build a set of low-level requirements, and test cases that will validate correct performance.

4 Separate control and data requirements

Designing systems with a clear separation between control and data processing simplifies the design, which improves testing. The chart below shows how this is broken out.

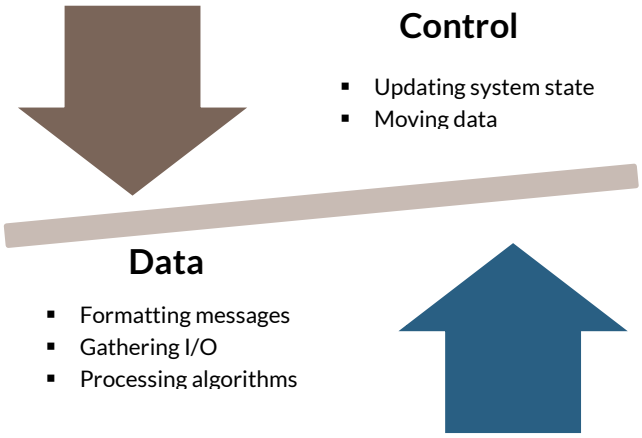

Figure 1: A clear separation between control and data processing helps the testing process

A clear separation allows designers and testers to focus on specific types of inputs, behaviors, and results. Additionally, a clean design increases software reuse, and decreases maintenance costs.

5 Pay attention to coding style and architecture

A good starting point for improving "testability" is to make sure that the code is easy to understand and designed to be flexible. In the early days of software development, developers who could make software "small and fast" were held in high regard. Computer systems had limited memory and CPU capacity, and engineers were constantly trying to cram more functionality into every application. Additionally, application code bases were much smaller and were often completely re-written over time as languages and hardware changed.

Things today are very different. While there are still size constraints and time-critical application components, overall application size and lifecycles have gotten dramatically larger and longer. Thus, there should be a premium on building code that is easy to understand and maintain. Fortunately, making code easy to understand dramatically improves "testability".

Bob Gray of consulting firm Virtual Solutions famously said that "Writing in C or C++ is like running a chain saw with all the safety guards removed" (cited in Byte (1998) Vol. 23, Nr 1-4. p. 70). Here are some "safety guards" for coding style that will improve testability.

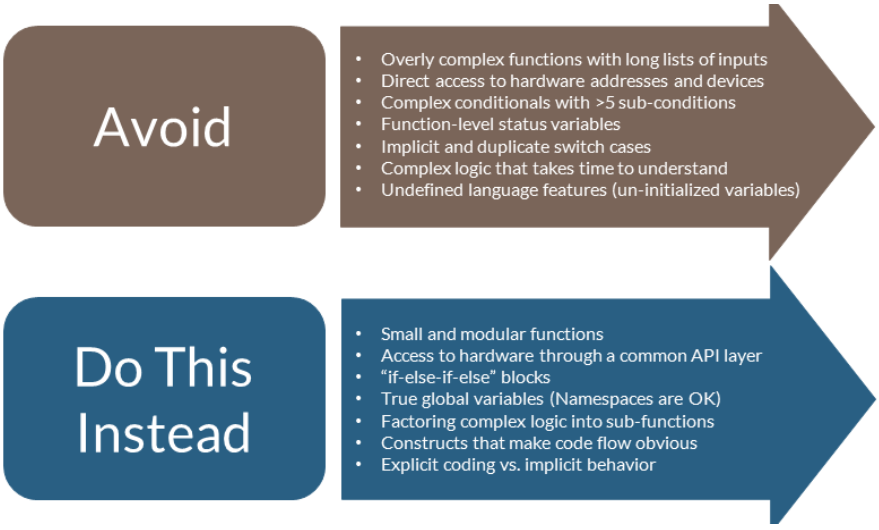

Figure 2: Follow these quick tips to help you write easily testable code

6 Implement a meaningful peer-review process

If you ask a developer what constitutes well designed code, you will probably get a different answer from each one. And you should! Implementing meaningful peer review is a great way to find out whether your code really is easy for others to understand or just you. Additionally, your team has a lot of "group knowledge", and peer review will ensure that you leverage this knowledge.

7 Best practices for improving software quality

The best method for creating a quality culture is to establish work-flows and metrics for each team member which addresses an individual part of the quality challenge. For example, if the developers are able to commit changes to the code base without any restrictions, or "gates", you will soon have a really buggy code base. On the other hand, if there are too many "gates", you may never release a new version.

What you need is a practical and repeatable workflow. For example, prior to being allowed to commit a code change into configuration management, developers might have the following obligations.

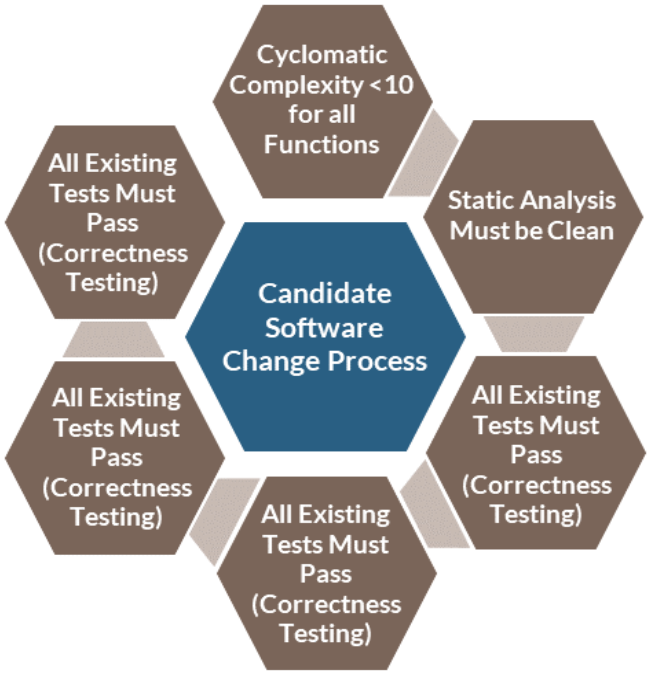

Figure 3: Candidate software change process

If this sort of process is new to your organization, or you have a 20-year-old code base with an inadequate test infrastructure, you will have to create a workflow that is appropriate for those circumstances. No one would suggest building test cases to achieve 100% code coverage for an already deployed application, but wouldn't it make sense to impose a requirement that all "new" code must have correctness and completeness testing performed? One thing is clear: adopting an attitude of "our code base is horrible, we can never make it better" will never improve quality.

As you transition to a *quality culture* you might get some push-back from the developers who consider the new "gates" time consuming and "additional work". Mentoring and training can smooth the transition. Creating a group of senior team members to lead the transition and leveraging vendor consultants to help with best practices are two low cost methods to speed adoption. Real buy-in will happen when the developers see their work move through QA and get deployed to customers more quickly.

It is important for the team to see tangible results, so publishing metrics as bugs are found and fixed at each stage of the process will reinforce the internal quality improvement message. Over time, the bug count should move "left", as shown in the following example.

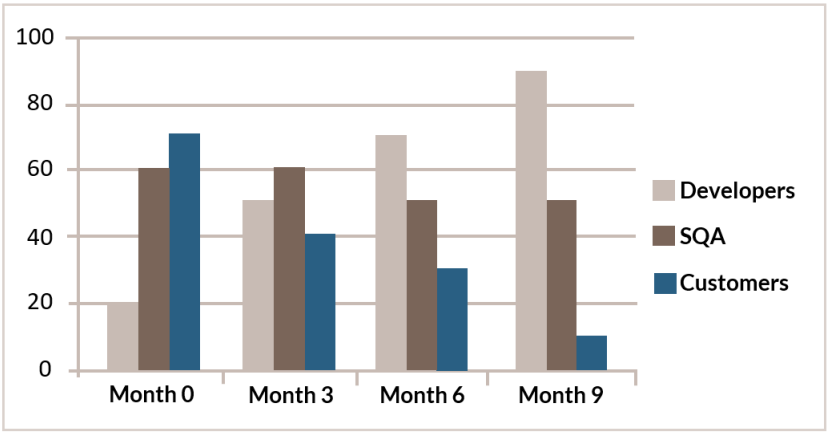

Figure 4: Fixing bugs before they are shipped is the most cost-effective way to deliver quality applications

As fewer bugs get to QA, the developers will have more time to work on new features, and QA will have more time for deeper testing — both of which will result in fewer bugs reaching the customer.

8 Implementing test automation

For software testing to improve application quality, it has to be thorough. It also must be easy and fast, so that any developer can run any test, at any time, on any version of the code.

While every organization has developed a software build system that allows for unattended incremental application building, most have not implemented a repeatable incremental testing infrastructure. Too often, testing is performed periodically with manual processes required, rather than constantly and incrementally with complete automation.

In many groups, each developer implements their own testing methodology, and there is no organization-wide platform for test automation. Because of this, there will be a significant lag between when a bug is introduced and when it is found.

The longer the lag time, the harder it is the find the cause. Ideally, each change to the software will be run against all tests that are affected by that change prior to the change being implemented.

So how do you determine what tests to run when the code changes? A change-based testing infrastructure will be able to automatically determine the sub-set of all tests that are touched by a change, and run only those tests. The key for this process to work is making the following steps fast and automatic.

1. Identify sub-set of tests affected by a code change
2. Run those tests against the code change
3. Report on any test failures and update code coverage

This approach enables a coding and testing cycle of minutes, rather than days.

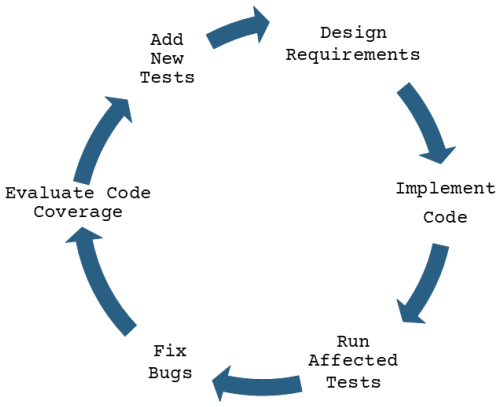

Figure 5: Implementing a continuous test process speeds time-to-market and ensures quality applications are delivered

9 Development tools are needed

If you are interested in creating a quality culture the following should be in your software toolbox.

- Static Analysis Tools: find patterns in the source code that commonly result in bugs
- Dynamic Testing Tools: black-box and white-box testing to ensure application correctness
- Coverage Analysis Tools: measures the thoroughness of all test activities
- Test Automation Tools: allows any team members to run any test on any code change

10 Business advantages

Improving quality and testing completeness are great goals, but in today's hyper-competitive economy there must be valid business reasons for change. Here are some obvious ones:

For improving software testability:

- Lower Development Costs
  - > Lower-cost junior developers can be used for test design and implementation
  - > Fewer tests are impacted by each code change
- Lower Life Cycle Costs
  - > Increased software re-use
  - > Decreased maintenance cost

For improving software quality:

- Fewer bugs go to integration, which shortens integration time
- Shorter integration time means faster release cycles
- Fewer bugs go to customers, leading to happier customers
- Happier customers lead to increased revenue and brand loyalty

11 Summary

It's hard work building high quality software. Creating a design approach that is mindful of testing at every stage of development enables your team to deliver applications faster, with quality "baked in" to the product. The earlier you start planning your testing campaign, the easier it is to handle. Humility plays a role, too – the fact is that coding does matter, but not as much as your developers might think. Ultimately, establishing a test-focused process and investing in good tools for software testing is what enables organizations to solve the software quality challenge.

Imprint

Vector Informatik GmbH
Ingersheimer Str. 24
70499 Stuttgart
Germany

Phone: +49 711-80670-0
E-mail: info@vector.com

www.coderskitchen.com