IPS

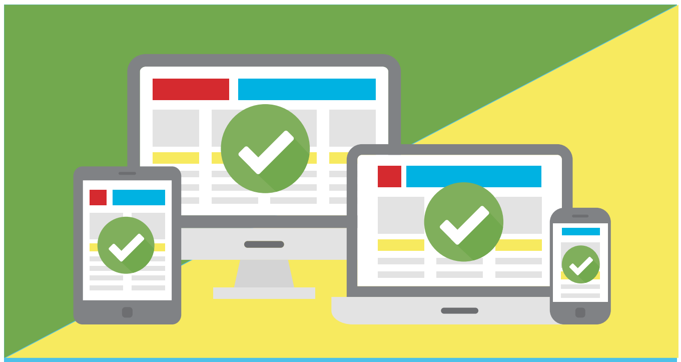
New York
Minneapolis

Turning Your Software Development Team into a World-Class Juggernaut

Michael Yackavage, *Senior Software Engineer*

September | 2017

Turning Your Software Development Team into a World-Class Juggernaut



Four Factors to Ensure High Quality Software

- **Automated Unit Testing:**
  As the first line of defense in software it removes the risk of having to back track.
- **Automated Static Code Analysis**
- **Coding Style Checker**
- **Code Review:**
  Gets rid of bad code and if followed the 3 steps properly there isn't much need here!

So. You're pulling together your software team for a project.

Maybe you've written your architecture description and design documentation, and you've had all the UML diagrams reviewed against the requirements. Perhaps you have simple prototype software, or have a legacy code base that needs to be carried forward against new requirements...or you are about to write the first line of code of a grand design.

Your development team gets underway, knocking off the bullet list items or agile stories. The capabilities start coming, but how can you ensure that the software being produced is high quality and capable of being maintained?

Consider applying these four factors to your development, even if your team has just two members or the project is only 10K lines of code. They provide value straight away, and add value to your code base long-term.

Four Steps to SW Coding Success

1. Automated Unit Testing

The earlier that bugs are found, the less expensive they are to fix.

Long ago the term "unit testing" referred to developer-driven one-off testing of code. The developer might informally write a stub of code to exercise a particularly complex piece of code and push some test cases through it, possibly changing conditions and recompiling it for each test they informally thought of. The stub would be discarded, or if kept its instructions would often be long-lost. QA would take the code for component testing, functional testing or system testing which are often done without visibility into the internal algorithms. If the code needed modification later, there would be little "unit testing" of the changes especially if the changes are done by someone new. There is rarely enough time or resource available for this type of "unit testing" when fixing bugs that have found their way to late stages or worse, out in the field.

More recently, engines have been developed to facilitate the maintenance and reusability of these unit tests. Developers can write them before they code the modules, and they can be run every time code is checked-into the code repository where they can reject the submission upon failure. They are simplest to apply to object-oriented languages where the testing can focus exclusively on the object, but they can be applied to more fundamental languages such as C as well.

Virtually all higher-level languages have automated software test capabilities readily available. Some are free to use, or relatively low cost. Most can be applied when the code is compiled, and they generally do not embed themselves in the production build itself so they do not bloat the code base.

The benefits of automated unit testing are many, but here are the ones I have found invaluable:

- Unit testing is the earliest and least-expensive opportunity to find software bugs in written code.

- White-box testing of edge cases.

No one knows where the edges of the algorithm are better than the author. Simple concepts such as where a counter or memory buffer overflows can be protected up-front, rather than trying to get all of these at a component or system level test.

- Facilitates refactoring

If a code maintainer has no way of quickly determining whether they have broken a piece of code, they may be inclined to change the code as little as possible.

Consider a scenario where the requirements on a software component have changed, and the original developer is no longer available. The component internal design may be using design patterns that were best suited for the original requirements, but are inefficient or inappropriate for the new requirements. Yet if someone maintaining the code has to wait until QA can validate a given change, the developer may shy away from such a large-scale algorithm change since they may be introducing problems that cannot be detected until much later (with higher cost). With good unit tests, the developer can make these changes with greater confidence that the changes will not destabilize the code.

- Shows example of how functions/methods/objects will be used

For software engineers attempting to use a module of code, a sample unit test can be an easy example of how the module is expected to be called.

2. Automated Static Code Analysis

What static code analysis can accomplish differs based on the source code language, but typically static code analysis is run on source code or object code, and examines the code for potential:

- Memory and other resource leaks. Was all of the heap space given back? Was a resource opened but never closed? Without static analysis these issues are often caught late, if at all.
- Potential security exposures: Might some conditions cause the code to overrun a buffer? Are deprecated methods used? Does the code use excessive (or incorrect) casting?
- Inefficiencies: Does the code build a string with excessive concatenation? A better algorithm can free the CPU for other work.
- Unused logic: A quality IDE can flag unused logic, true, but automated static code analysis can be used as a gatekeeper to keep unused logic from making it into your production software.
- Best practices: The code may be "correct", but how maintainable is it? Does the code contain a custom to String() method that might return null? Is there a loop with an unusual increment? Did one object open a resource and some other unrelated object close that resource? This may be what your design calls for, but in case that is not what you had intended a static code analyzer can bring these conditions to your attention.

Running static code analysis is like having a code reviewer sanity check the code before burning human capital on it, and often it can teach your intermediate team advanced nuances of your language better than articles or textbooks can.

Static code analysis cannot find all basic bugs by any means, but it does find a fair share. If the developers are able to run your selected tool(s) during development, there is no loss-of-face either.

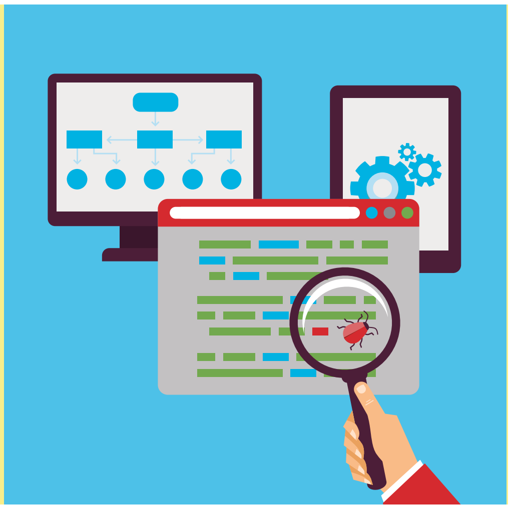
One head's up though - if the tools are first applied to an existing code base late in the project, the call outs can be overwhelming. If this is your only option to apply these tools, be ready to have your team go through a large list of callouts to diagnose them and don't lose heart. The callouts can be prioritized and even just the high priority ones can still save your team a great deal of effort over trying to analyze the equivalent complicated QA bugs even later in the project timeline.

Software Quality Assurance will save you money (and potentially your business)

The Department of Commerce's National Institute of Standards and Technology found that software bugs cost the US economy $59.5 billion annually.

The report, commissioned by Tricentis, identifies 548 recorded software fails affecting some 4.4 billion people and US$1.1 trillion in assets. It highlights year-on-year software fail statistics and trends across finance, retail, services (e.g. internet, telecom), government transportation, and entertainment.

Reportedly, accumulated time lost due to software failures was 315 years, 6 months, 2 weeks, 6 days, 16 hours and 26 minutes.



3. Coding Style Checker

Most experienced software engineers have their own writing style. Maybe you like to left-justify all of your curly-braces, and others like to have the open curly-brace on the right of the line that opens the scope. Maybe you like to indent each line four spaces and your teammate indents three.

There are a few coding "standards" out there for each language, and it's worthwhile for you to pick one and stick with it. There are two benefits for doing so:

- The code will be easier to read and maintain for new engineers in the future, if it all reads similarly.
- A common code style facilitates more valuable and efficient code reviews (spoiler alert, the fourth tip).

Few software engineers want to be the ones that enforce the coding standards, and those that do enjoy it are not well liked. Fortunately there are a few tools out there that will automatically check code against style rules, and similarly to the automated unit testing they can reject the code at compile time or at repository commit time so there is no time wasted on debate. Many allow you to configure their style rules, so you can change that one nuance that your team just can't get behind (eg: "I would use the google standard if they just didn't indent that second line eight spaces"). Some IDE's will auto-adjust code to your standard directly - which you might already know if you've sent sample code to a vendor, and received it back with a one line change but otherwise completely reformatted leaving you searching for the change.

Once a developer becomes used to the style, these call-outs are trivial to adjust when they are called-out and do not cost much additional effort.

Catch bugs early with automated unit testing & static code analysis:

It is much more difficult to address a SW bug after release, rather than catching it with software testing.

The cost of a bug goes up based on how far down the SDLC (Software Development Life Cycle) the bug is found. When a bug is found in production the code needs to go back to the beginning of the SDLC so the agile development cycle can restart.

The Systems Sciences Institute at IBM has reported that "the cost to fix an error found after product release was four to five times as much as one uncovered during design, and up to 100 times more than one identified in the maintenance phase." This doesn't even include the frustration and embarrassment.

As you can see, finding a bug later in the life cycle of the software costs exponentially more. Collaborating with the customer through Agile software testing will greatly help reduce development against buggy requirements.

4. Code Review

If you have little time and have skipped any of the previous steps, consider them first as they will provide more return for your effort. However, if you have applied the three earlier concepts, peer review of code will efficient and time well-spent.

Code review is probably the oldest concept on this list, and when it first came about it was probably the most unpleasant part of software development. Engineers would sit together in a room with paper printouts of source code, and read through it together, mostly calling out valueless coding style comments. There was a published average rate of bugs per 24-line page of code, and the code review would be forced to continue, possibly over several sessions until that number of bugs were called-out. Someone would write up the findings, and management would get a false sense of security about the correctness of the code. The statistics would be applied, of course reinforcing the average rate of bugs found since the method was a self-fulfilling prophecy and documents were prepared showing how much money was saved by this method. Rinse, lather, repeat.

In reality however, unless you had superstars on your review team the more obvious call-outs would burn much of the energy and effort long before the more heinous bugs could be uncovered.

Code review has come a long way since then. A code review with these three other tools in place leaves more time to concentrate on the business algorithms in use, the requirements, and more complex concepts in the victim code.

"Software is not quality software until it passes testing."
Danny Aponte, *IPS Senior Director, Software Engineering*

This kind of code review also has benefits for the team, in that they will learn from each other and they all will have some ideas on how the parts of the system that they are not coding themselves are working.

With development groups not being co-located these days, some great tools have arisen that facilitate asynchronous code review. Participants can review the code when convenient for them, they can put changes and original code side-by-side and post comments directly in context that other team members can reply to. Approvals can be tracked and "gate-keepers" can be configured to require a certain approval level before allowing the change to be committed to a master branch.

Conclusion

Adopting these four concepts can allow the energy of your team to be channeled into greater things, and allow them to become the software juggernaut you know they can be.