



Strong Engineering LLC

Recommendations for Software Process Improvement

- Duane Strong
- Strong Engineering LLC
- duanes@strongenging.com



Introduction

As a consultant, I get to see how many different companies develop software. Something I found interesting was how similar many of these companies were in terms of weaknesses in their software process. This presentation is a collection of recommendations I propose time and time again. These are just the most basic recommendations.

This presentation will propose a number of recommendations to improve the quality and time to market of software components. These recommendations are well known in the software engineering discipline and have been documented by a number of case studies that show improvement in the software process in terms of fewer bugs, faster time to market, and a higher level of satisfaction by the end users of the software components.

Organizational and Coding

The recommendations are split into two groups; organizational, and coding.

The organizational recommendations are more about how the department is run.

The coding requirements are more about what the department produces.

In each section the recommendations are in the order of their importance.

Organizational-Invest in Process Improvement

Implementing these recommendations and keeping them current will require a large amount of someone's time.

Establish an official position (software manager, software lead, architect, etc.) and task that person with the responsibility of implementing these recommendations. This will take time away from production coding by that person especially at first, and always by some percent of their time.

Realize that this is an investment no different than a capital investment that will pay back in the long term. Realize that by not doing so you will be creating long term problems that directly affect the quality of your product and result in higher cost overall.

Organizational-Collect, organize, and centralize information

Establish an internal web site, cloud service or source control that contains all the documentation pertaining to the software department and the products it produces.

A team member should never have to ask someone to dig up a document from his or her personal workstation.

Consider also keeping these documents under source control.

Consider using a Wiki web server extension for collaborative documentation.

Consider using markdown syntax and keeping the documentation along side the code it speaks to.

Establish a standard 'look' for documents using an html template or standard Word template. Document all software standards and development tool setups.

Organizational-Establish coding conventions

Document conventions for how code should be written. Enforce adherence to the conventions by team members.

This is the time to re-evaluate past practices and learn from those who have done this before.

There are many excellent books on this topic, some are listed in the References at the end of this presentation.

Organizational-Improve the specification process

Many specifications I see are of the 'stream of consciousness' type because they are an attempt to record the thought process of the engineer after they have produced an artifact. Specifications need to be written before the action of writing the software has begun. Yes even in agile! It is not possible to schedule a software effort without some type of specification up front.

Specifications need not be overly verbose and should fit into an overall methodology. Some companies like Big Upfront Design where specs are huge and unchanging and the process is a waterfall model. Others like Agile or Extreme processes where the spec is under constant change and many small iterations of software are produced.

Organizational-Improve the specification process

In agile environments the specification can be represented by the stories in the backlog. Even then those stories need to be traceable back to a requirements specifications.

A requirements specification usually originates via inputs from a marketing department or by the Product Owner and can be quite succinct.

These inputs need to be organized by the engineering department to ensure that the requirements are technically feasible.

Organizational-Document the schedule

A schedule for the software should be generated and documented. A simple spreadsheet is all that is required for this, not Microsoft Project.

Tasks should be identified from the specification and broken down into a granularity no larger than a numbers of days (or even hours.) See Painless Software Schedules in the reference section.

Ideally once the tasks are sufficiently broken down the size of a story becomes the “work unit” and the schedule can be estimated just from the count of stories, without the need for spreadsheets or other means.

Organizational-Share code in source control

Most of the time code is shared (if at all) via a 'cut and paste' method where multiple divergent copies are present in a number of source control projects.

These should all be pulled from the same source file in source control and used directly in all projects. Source control should represent the Intellectual Property Library of the company and should be leveraged into as many products as possible.

This may require some refactoring of the source code to allow for better configuration in each platform where it is used. The sources should be packaged into multiple folders where each folder represents a stand-alone highly cohesive component with minimal coupling to other components.

Organizational-Implement code reviews

Before any code is released (or even merged to a main branch) it should be reviewed by a selection of team members.

Bugs caught at this phase have been shown to be exponentially less costly than at later stages.

A formal review process should be established where the author must present the source and comments/suggestions are recorded and action items assigned. Consider using an online review system like GitHub or GitLab.

Organizational-Use Issue/Bug tracking software

There are many excellent products on the market for tracking issues and bugs. Resist the temptation to use a spreadsheet!

Keep all discussions, data, photos, charts etc. concerning the bug or issue in the bug or issue tracker. Do not have email, Slack, Teams or whatever conversations that are not part of the bug or issue tracker.

This data is extremely valuable to the company as time goes on and can be “mined” for all types of information.

Organizational-Implement unit testing

Require developers to test their own code first by writing unit tests.

These catch many bugs before they get to system integration and SQA where they cost much more to fix.

Consider using a unit test framework similar to JUnit for Java.

Some organizations drive development from the unit tests in a test driven development environment.

Having a robust set of unit tests makes regression testing of a new release much easier and less risky.

Organizational-Automatic documentation generator

Documentation beyond a high level software specification can be done by inserting special comments into the source.

These 'Java Doc' comments can be extracted into nice looking PDF or HTML documents automatically by tools such as Doxygen
www.doxygen.org

Organizational-Establish automatic nightly builds

A system that automatically builds all products each night or at each merge catches problems introduced when modifying shared code.

Getting this information in a timely fashion is important to keep shipping product's code bases intact while implementing new products and features.

A system that can archive each build is very useful for testing as it can pinpoint when a problem emerges.

Coding-No warnings on the build

Establish a goal to have no warnings on the build.

Too many warnings and the important (crash inducing) ones are missed among the noise.

While it may not be possible to eliminate them totally, this should be the goal. No more than a page full for an entire build should be tolerated.

A warning from the compiler is your friend. It is trying to tell you something important.

Coding-Reduce `#ifdef` and `#define`

Establish a goal to reduce or eliminate `#ifdef` and `#define`.

Platform configuration via `#ifdef` results in spaghetti code that is difficult to maintain. Try to drive configuration from the makefile and use different include paths to pick up variation in implementations.

The C preprocessor has many documented problems that cause many types of errors.

C++ offers facilities that can lessen the dependence on the preprocessor. Refactoring using configuration classes, enumerations, and inline functions can eliminate `#ifdefs` and `#defines`.

Coding-Eliminate global variables

Globals are a source of bugs and usually indicate system architecture that has not been thought through.

These can be eliminated through the use of the singleton pattern and other techniques.

Coding-Reduce class sizes

Many times the code base has some monster sized classes.

These are too complex to be effectively reused in other environments.

Consider splitting them up into classes that are combined into an overall class via composition.

Coding-Reduce source file sizes

Source file sizes should not exceed a few tens of kilobytes.

Files that exceed this indicate that they need to be broken up into smaller classes or simply fewer classes per file.

References

The Joel Test: 12 Steps to Better Code

<http://www.joelonsoftware.com/articles/fog0000000043.html>

Painless Software Schedules

<http://www.joelonsoftware.com/articles/fog0000000245.html>

Rapid Development, McConnel, Microsoft Press, 1996

<https://www.amazon.com/gp/product/1556159005>

Code Complete 2nd edition, McConnel, Microsoft Press, 2004

<https://www.amazon.com/gp/product/0735619670>

C++ Coding Standards, Sutter and Alexandrescu, Addison Wesley, 2005

<https://www.amazon.com/Coding-Standards-Rules-Guidelines-Practices/dp/0321113586/>

Q & A