Managing Legacy Code

An Approach to Improving Older Code Bases

Coding Standards are filled with practical and good guidelines for catching and preventing bugs and making code more readable, maintainable and safe. PRQA’s QA C and QA C++ are recognised the world over as the most capable and accurate code analysis toolsets for these important activities.

But what if you need to start from an existing codebase with a long history of maintenance change and adaptation? Many developers have to work with portions of code that are almost hereditary in nature. While you will always want to pay close attention to the bugs that QA C and QA C++ expose, when you advance towards the prevention-oriented advice encapsulated in a coding standard you will find many non-compliances and be quickly flooded with many more “fix requests” than you can reasonably make in a single maintenance cycle.

What you often want to do is to selectively improve your code - to gradually reach full compliance, and to gradually eliminate dangerous and suspect coding practices. Of course this is not to overlook bad coding practices – there will always be critical bug-fix issues that ought to be immediately corrected. However, in a legacy environment there may be code fragments that represent potential danger but not actual danger in terms of how the code has been exercised in those legacy products. In such cases of code maintenance, you might want to remove all new coding non-compliances but not touch the legacy ones; in other words to at least not make the code worse.

PRQA has developed a technology to help this activity, termed Baselining. The basic premise is that for two versions of the same code base the set of coding issues – bugs or coding standard non-compliances – are compared and matched where they coincide. With some following technology, the net result is that you can view only the relevant issues that were introduced in your latest code changes. These are exactly the code corrections you should be able to make in the development cycle, prior to QA, Unit and Integration Testing.

To show its rich capabilities and its variety of applications, let’s explore this technology solution in more detail.

PRQA’s analysis technology, in QA C and QA C++, operates firstly on each Translation Unit (TU). Resulting from this analysis, diagnostics are created against locations in the source and in common header files. These diagnostics are usually mapped to coding rule violations, or may simply represent individual code bugs. Following on from this initial examination is a Cross-Module Analysis (CMA) phase, operating across the entire set of
TUs that form a linkable entity. This phase similarly generates diagnostics that catches many link-time errors and erroneous practices.

There are additional smarts in viewing this collection of diagnostic output. When merging output from each TU, it is necessary to de-duplicate diagnostics associated with common headers. Also, CMA diagnostics are merged with those from the TU analysis phase for each source or header file. Baselining technology must take account of all this merging, sorting and de-duplication in order to process a clean set of diagnostics.

Baseline analysis involves comparison of the set of diagnostics between two versions of a code base. Let’s call these the “new” and “old” code versions. Using a text-based differencing utility we can re-calibrate the “new” location of each diagnostic to where it would occur in the “old” version. This then allows us to temporarily remove, or suppress, it on the basis of its presence in the “old” version. Between two such versions of a code base, there will typically be entity name changes, different starting locations of functions or data, and different function lengths. Items might be added or removed. An added complexity is code altering behaviour of the pre-processor.

PRQA’s solution covers all these technical challenges. When reviewing diagnostic output through the baselining prism, prior diagnostics are automatically suppressed. The necessary text differencing operation occurs across each source and header file in the code base. If you don’t want to have to store the “old” code version, there is a capability to use the version control system (VCS) to manage the differencing operation. The baselining solution supports the most popular VCSs, with quick and easy adaptation of for other ones.

The popular Message Browser is a GUI component that adeptly represents the real-world mix of source/header files, providing full-functional viewing of active and baseline-hidden diagnostics alongside source, with summaries organised by file, message hierarchy, and analysis phase.

With baselining in mind, the Message Browser component has a new mode of operation: “Baseline Administration”. The user can browse the entire set of baseline diagnostics, and re-activate any chosen selection of diagnostics in order to make them reappear in any baselined analysis of a “new” version of source. There is easy navigation to select a single diagnostic or all those for a given message id, message group, or message level in the hierarchy. Save these changes to the baseline set, and this new adjusted baseline can be deployed for use by the entire team of developers, so that they always fix the important diagnostics and leave aside the residual unimportant ones.

Baselining is clearly a powerful feature for managing legacy code-bases. It permits focussed repair of the most important bugs and code correction work. It allows a priority set of diagnostics be passed to the development team for immediate fixing. Baselines can be reset in successive fix/analyse cycles for individuals or developer teams.
Finally, it also turns out that this technology has a couple of interesting applications, other than performing a comparison between two versions of source code. You can use it to test your application between host and target environments. If you change the analysis configuration to set tighter sizes of fundamental types and their byte alignments, you can model and check for newly arising conversion and pointer alignment issues. Another use-case might be to see what new diagnostics arise from a newer version of QAC or QAC++ before production deployment.

This highly refined form of differential analysis is a new and unique capability of our tools. Coupled with the precise diagnostic capabilities and strong language focus of the PRQA toolsets, the baselining solution gives you an additional foothold for advancing your code quality improvement efforts.

*** end ***