TIOBE Software BV
Website: www.tiobe.com
Email: info@tiobe.com
Phone: +31 (0)40 400 2800

TIOBE coding standard methodology

TIOBE has developed a step-by-step methodology to support companies to introduce coding standards into their organization. The methodology consists of 7 logical steps and is based on years of experience in the field. The independent nature of the steps allows organizations to take a sufficient break after each step to get familiar with the new situation. The 7 steps are shown in the figure below

Step #1: Adopt/Define Coding Standards

The first action to be taken is to define or adopt a coding standard. Usually a coding standard consists of a set of programming rules (e.g. “Do not compare two floating point numbers.”), naming conventions (e.g. “Classes should start with capital C.”), and layout specifications (e.g. “Indent 4 spaces.”).

It is strongly advised to adopt an existing coding standard instead of defining one yourself. The main advantage of this is that it saves a lot of effort. An extra reason for adaptation is that if you take a well-known coding standard there will probably be checking tools available that support this standard. It can even been put the other way around: purchase a code checking tool and declare (a selection of) the rules in it as your coding standard!

For C
“MISRA-C:2004 Guidelines for the use of the C language in critical systems” – MISRA
“ASML C Coding Standard” – ASML
“BARR C Coding Standard” – BARR Group

For C++
“C++ Core Guidelines” – GitHub
“Philips C++ Coding Standard” – Philips

For C#
“Microsoft C# Coding Conventions” – Microsoft
“Resharper Rule Set” – JetBrains
“Coding Standard: C#” – Philips Medical Systems

For Java
“Google Java Style Guide” – Google
“Android Open Source Project Java Code Style” – Android Open Source Project

For Python
“PEP8 – Style Guide for Python Code” – Python.org
“Google Python Style Guide” – Google

On the right, there is a list of most frequently used coding standards for mainstream industrial programming languages, that are used by our customers:

There might be good reasons to define your own coding standard. For instance, if a lot of existing code already adheres to undocumented but consistently used rules it is better to make these rules explicit in a proprietary standard.

It is convenient to have your coding standard available in a database or at least in hyperlinked HTML or XML instead of a plain document. This will help you to reuse coding standard info in later stages, e.g. jump from violations in the IDE to the corresponding description of the coding rule in the standard or generate a list of coding rules that must be used during code review because they are not checked automatically by a tool.

TIOBE Software BV
Website: www.tiobe.com
Email: info@tiobe.com
Phone: +31 (0)40 400 2800

Step #2: Select Code Checking Tool(s)

Code checking tools are software products that check compile-time whether a software program adheres to a certain set of rules of a coding standard. Without such code checkers, the enforcement of a coding standard in an organization is likely to fail. There are two main causes for this: (1) the number of rules in a coding standard is usually so large that nobody can remind them all (except the author) and (2) some context-sensitive rules that demand inspection of several files are very hard to check by human beings.

Although code checkers can reduce formal code review time considerably, it certainly does not mean that code reviews are not needed any more. Some coding rules cannot be checked at all (e.g. “Comments should be in English”) or only partly at compile-time (e.g. “Use delete, whenever you use new”). On average about 65% of a coding standard can be covered with code checking tools.

There are several characteristics that should be assessed during the selection process of a code checker. There are 6 issues that are most important in our opinion. Obvious requirements such as whether it is available for the used platform are not taken into account.

- Availability to define own rules. Experience has shown that only about 20% of a proprietary coding standard is available as built-in rule in a code checker. As a consequence, there should be a possibility to define own rules. This can be via an API (complex in use but powerful) or a graphical user interface (easy to use but limited).
- Integration in programming IDE. Most end users of code checkers are software engineers. Therefore a code checker should fit smoothly in the used programming environment. Preferably via a toolbar with an easy way of running the tool (one button push).
- Presence of command-line version. In order to be able to integrate the code checker in the software development process and/or in the programming IDE (if no plug-in is available) it should be command-line oriented. Command-line versions also allow for nightly batch runs of the code checker to collect quality data.
- High performance. Code checkers are slower than compilers by nature because they perform a more in-depth semantical analysis of the source code. However, performance is a key issue for end user acceptance of the tooling. Response times of more than 15 seconds per file are perceived as not workable.
- Low cost of ownership. Cost of ownership means purchase price and maintenance costs. Since code checkers are not used continuously by end users, floating licenses appear to be more cost effective in most cases. About 1 floating end user license is needed per 10 developers.
- Available support. Decent product support is an essential feature of a code checker. Good support does not only mean prompt and adequate response to user questions and problems, but also the frequency of new product release. “We will solve this in the next release” is not interesting if the average product release cycle is 1 year.
- False positive rate. If a code checker produces a lot of false positives (issuing a violation that appears to be a non-issue), users will get annoyed and stop using the code checker. If the false positive rate exceeds 5%, you better consider another code checker

TIOBE Software BV
Website: www.tiobe.com
Email: info@tiobe.com
Phone: +31 (0)40 400 2800

Apart from the requirements mentioned above, it is recommended to select a market leader's mainstream code checker. Current market leaders (as we encounter them at our customer sites) for the various programming languages are listed here.

For C	For C++	For C#	For Java	For Python
Coverity	Coverity	Resharper	PMD	Pylint
C++test	C++test	FxCop	CheckStyle	
PC-Lint	PC-Lint	Microsoft Code analyser	Jtest	
	CppCheck		SpotBugs	

Step #3: Customize Code Checking Tool(s)

The first obvious action of customizing a code checker concerns switching on all available rules that are part of the coding standard. This usually takes more effort than expected. Comparing say 250 built-in rules with 100 rules of a coding standard is a laborious task, especially because the meaning of the rules may differ in subtle ways.

Rules that can be checked automatically but are not provided as built-in rules can be implemented by means of a provided API or a graphical configuration tool. This is a specialized task that can be done best by engineers that are experienced in this field.

Companies that want to increase coverage of their coding standard even further can decide to write extra code checking tools of their own. This can be interesting in case there are no off-the-shelf code checkers available that allow the implementation of their complex, context dependent coding rules. But it can also be initiated because a non-standard dialect of a programming language is used (e.g. for a specific processor), or a substantial part of the code is written in a special, non-mainstream programming language or even in case the used platform is unusual. Unfortunately, most multinational companies are forced to develop add-on customized code checking tools for these reasons.

Despite the fact that proprietary tooling introduces a maintenance problem, there is also an advantage: more control. Think of direct hyperlinked connection between coding standard rules and rules defined in the tooling, think of applying different coding standards for different kinds of files (plain C++ vs. C++/CLI, for instance), think of introducing proprietary levels or categories, etc . In short, own tooling allows for adjusting it in such a way that it fits perfectly into the existing development processes instead of the other way around. For large organizations, it is often harder to change established processes than to invest in proprietary tooling..

TIOBE Software BV
Website: www.tiobe.com
Email: info@tiobe.com
Phone: +31 (0)40 400 2800

Step #4: Integration in Software Environment

As soon as your code checker has been configured according to the available coding standard, it is almost ready for use in daily practice. The only thing that remains to be done is integration in the software environment. This comes primarily down to:

- Providing a plug-in for the used programming environment.
- Alignment with the compilation process.
- Arrange up-to-date access to all parts of the code archive.

Plug-in in Programming Environment
A code checker should be available as a set of simple buttons within the used programming environment in order to ease its introduction in the organization. That's why code checker providers usually offer plug-ins for mainstream environments. Examples of popular programming environments are Eclipse (Java/C++) and Visual Studio (C++/C#).

If for some reason more than one code checker is in use -e.g. because two code checkers guarantee more coverage of the coding standards than one, or different languages are used within one programming environment- the outcomes of these checkers should be combined. This means that an appropriate plug-in must be provided that acts as an umbrella for the individual coding checkers.

Alignment Compilation Process
Static code checking is based on structural analysis of source code. If a file "includes", "imports", or "uses" another file, the code in that other file needs to be examined too because it could contain important information. But where to find these used files? Fortunately, information about the location of these other files should be present already because it is also needed for the compilation process.

In case a mainstream compilation process has been adopted such as Linux' make or the Microsoft project approach, the appropriate information will probably be provided by the code checker vendor. However, most large organizations deviate from standard compilation solutions (for good reasons by the way). This means that the compilation process must be simulated in some way to extract the location of the include files (such as the "-n option" for most make facilities) or the information must be gathered from some kind of log file after compilation.

Another commonly used approach is to integrate code checking into the compilation process. Instead of applying the compiler, the code checker is invoked. In this way duplication of information is avoided. Besides this, it is sometimes easy to accomplish. The disadvantage to this approach is that it becomes impossible to check individual files, such as a separate header file, since checking is bound to a compilation target.

Access to the Code Archive
This might seem a trivial constraint. For large code archives with complex, multi-site configuration management systems with many parallel branches this is certainly not the case. The complexity of this problem may even increase if one makes heavy use of symbolic links, mapped network drives, and/or a mixture of file systems (e.g. using Linux and Windows in parallel).

Step #5: Set up Quality Database

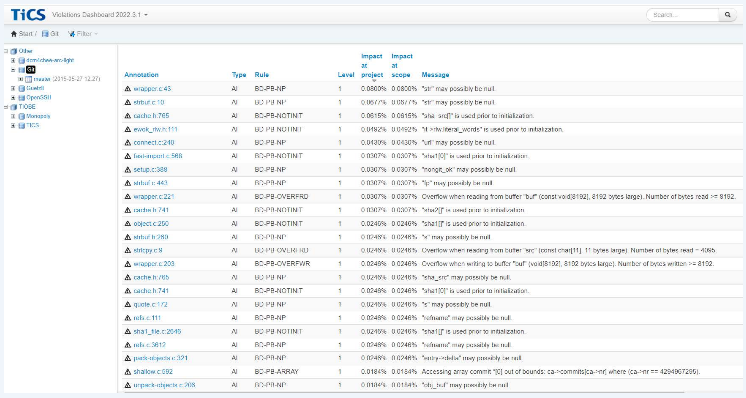
As long as code is checked by individual developers all calculated quality data is volatile. This means that it is not stored and thus cannot be retrieved at a later moment in time. In this way, it is impossible for organizations to get a clear overview of the current status of violations in the code archive. A “quality database” that contains all violations helps organizations to make structural improvements with respect to the coding standard through time.

The quality database runs on a “quality server” and collects new violation data by determining what files have been changed in the archive since its last check. Only deltas are checked because it is most of the time impossible to check all available files within 24 hours. A code archive of 10,000 files and an average check time of 30 seconds per file results into a throughput time of about 3.5 days for the entire archive. It is important to note that only accepted (sometimes called consolidated) files should be checked by the quality server. An intermediate file of an individual developer that has been saved but not checked in and that has not yet been accepted by the system integrator should not be taken into account.

What information should be collected in a quality database? If we are talking about large code archives, a lot of data will be generated. In order to keep control over the size of the database, one should be careful about what to store. Since large organizations tend to branch (complete archive copies that remain relatively unmodified) and merge their archives lots of time, file based data collection appears to be far too time and space-consuming in practice. Instead of taking files as the leading principle, which might seem to be the most natural entity at first sight, we propose a combination of file name and its checksum. This means that only if a file changes, a new entity will be created. The major advantage of this approach is that in this way copies of a file in different branches are checked and stored only once.

The following basic data is stored for file name/checksum entities:

- Complete file path (canonical);
- Lines of code;
- All violations (rule ID, line number) per file.



Annotation	Type	Rule	Level	Impact at project	Impact at scope	Message
wrapper.c:43	AI	BD-PB-NP	1	0.0800%	0.0800%	"str" may possibly be null.
strbuf.c:10	AI	BD-PB-NP	1	0.0677%	0.0677%	"str" may possibly be null.
cache.h:765	AI	BD-PB-NOTINIT	1	0.0615%	0.0615%	"sha_src[]" is used prior to initialization.
ewok_rlw.h:111	AI	BD-PB-NOTINIT	1	0.0492%	0.0492%	"it->rlw.literal_words" is used prior to initialization.
connect.c:240	AI	BD-PB-NP	1	0.0430%	0.0430%	"url" may possibly be null.
fast-import.c:568	AI	BD-PB-NOTINIT	1	0.0307%	0.0307%	"sha1[0]" is used prior to initialization.
setup.c:388	AI	BD-PB-NP	1	0.0307%	0.0307%	"nongit_ok" may possibly be null.
strbuf.c:443	AI	BD-PB-NP	1	0.0307%	0.0307%	"fp" may possibly be null.
wrapper.c:221	AI	BD-PB-OVERFRD	1	0.0307%	0.0307%	Overflow when reading from buffer "buf" (const void[8192], 8192 bytes large). Number of bytes read >= 8192.
cache.h:741	AI	BD-PB-NOTINIT	1	0.0307%	0.0307%	"sha2[]" is used prior to initialization.
object.c:250	AI	BD-PB-NOTINIT	1	0.0246%	0.0246%	"sha1[]" is used prior to initialization.
strbuf.h:260	AI	BD-PB-NP	1	0.0246%	0.0246%	"s" may possibly be null.
strlcpy.c:9	AI	BD-PB-OVERFRD	1	0.0246%	0.0246%	Overflow when reading from buffer "src" (const char[11], 11 bytes large). Number of bytes read = 4095.
wrapper.c:203	AI	BD-PB-OVERFWR	1	0.0246%	0.0246%	Overflow when writing to buffer "buf" (void[8192], 8192 bytes large). Number of bytes written >= 8192.
cache.h:765	AI	BD-PB-NP	1	0.0246%	0.0246%	"sha_src" may possibly be null.
cache.h:741	AI	BD-PB-NOTINIT	1	0.0246%	0.0246%	"sha1[0]" is used prior to initialization.
quote.c:172	AI	BD-PB-NP	1	0.0246%	0.0246%	"s" may possibly be null.
refs.c:111	AI	BD-PB-NP	1	0.0246%	0.0246%	"refname" may possibly be null.
sha1_file.c:2646	AI	BD-PB-NOTINIT	1	0.0246%	0.0246%	"sha1[]" is used prior to initialization.
refs.c:3612	AI	BD-PB-NP	1	0.0246%	0.0246%	"refname" may possibly be null.
pack-objects.c:321	AI	BD-PB-NP	1	0.0246%	0.0246%	"entry->delta" may possibly be null.
shallow.c:592	AI	BD-PB-ARRAY	1	0.0184%	0.0184%	Accessing array commit *[0] out of bounds: ca->commits[ca->nr] where (ca->nr == 4294967295).
unpack-objects.c:206	AI	BD-PB-NP	1	0.0184%	0.0184%	"obj_buf" may possibly be null.

On top of a quality database a web application can be built that shows all kinds of quality data. Historical comparisons, comparing different modules, absolutely or relative (i.e. per lines of code) views, organizational (comparing departments) or hierarchical (comparing directories), the sky is the limit. An example of such a web application is TIOBE’s TiCS framework as shown to the left.

TIOBE Software BV
Website: www.tiobe.com
Email: info@tiobe.com
Phone: +31 (0)40 400 2800

Step #6: Define Quality Targets

Having a coding standard in place and being able to check it automatically with code checkers is a major achievement. But if you don't put a little bit of force on being compliant to the standards, hardly anything will happen. For this you have to set quality targets. Experience has shown that relative targets work much better than absolute targets. Stating that every changed file should be coding standard violation free (absolute target), is not the way to go. This is because it implies that also all kind of violations need to be fixed that are already in for years. There is a risk that if you fix these you introduce new bugs, especially if you are not the one who introduced them in the first place. So it is better to define the quality target as "no new violations are allowed" (relative target). In this context, "new" means introduced since the last official baseline or even since last check-in. There is also a psychological effect to this: engineers are more willing to fix coding standard violations they have introduced themselves than cleaning up the mess of others. The only downside to this is that you need to have a facility to compare the changes to the right baseline. This might be challenging in an environment where engineers work on many branches in parallel.

In order to keep focus, it might help to restrict the quality target to only the most severe coding standard rules. So the quality target might be "no new violations are allowed of severity levels 1 to 3".

Step #7: Enforce Compliance

The last step in this process is enforcing compliance to the quality targets that have been set. This is achieved by using quality gates. Quality gates can be enabled during various stages of the software development cycle and in various ways. This all depends on the culture and size of the company. In small teams the communication lines are short and sending an email if some quality target is not met is sufficient. In larger organizations with larger turn over of engineers and a tendency to apply more formal methods, quality gates can be made more strict, e.g. not allowing to deliver code to the archive if the quality gates are not met. These kind of stricter quality gates are very effective and we have experienced very good results. There are some important preconditions before such quality gates are enforced. The number of false positives of the quality gated coding standard rules should be very close to 0% and there should be a very fast waiver mechanism (arbitration) in case something is wrong with the tooling to avoid unnecessary blocking of engineers when they want to deliver their code.