Roadmap for Coding Standards Implementation

2031 S. Myrtle Ave.
Monrovia, CA 91016
Phone: (888) 305-0041
Fax: (626) 305-9048
E-mail: info@parasoft.com
URL: www.parasoft.com

Abstract

There are a variety of error prevention practices available to software developers, among them code reviews, coding standards, unit testing, regression testing, and load and functionality testing. Each of these practices is perfectly suited to automation —software manufacturers committed to eradicating software errors would do well to create an automated error prevention system that addresses each of these practices as a separate stage within the software development lifecycle.

The focus of this white paper is coding standards (a.k.a. coding rules, or simply, rules). Coding standards are language-specific programming rules that greatly reduce the probability of introducing errors into your applications, regardless of which software development model (iterative, waterfall, eXtreme programming, etc.) is being used to create that application.

Coding standards are a vital and important error prevention tool, but they must be used regularly, even religiously, in order to be effective. This is especially true as development projects become more and more complex in an effort to meet consumer demand for better features and more functionality. Implementing and using coding standards early and enforcing them automatically throughout the development lifecycle results in many outstanding product benefits, including:

- Increasing product reliability.
- Increasing code portability and ease-of-maintenance.
- Increasing programmer productivity, both individually and across development teams.
- Reducing application time-to-market.
- Reducing development and support costs.
- Reducing developer learning curves for a specific programming language.
- Extending code-life
- Streamlining code reviews, making them more efficient and beneficial.
- Increasing the ease with which organizational certification is obtained.

Automated error prevention is an *evolution*, not a *revolution*, of your existing development process — automated coding standards make your development process better, more efficient, and more profitable.

Audience

The intended audience of this paper is software project managers and architects, who frequently have the cumbersome responsibility of selecting and implementing coding rules for development teams or organizations. Coding standard best practices for these roles are examined through a "Divide and Conquer" approach that includes implementing coding standards into the development lifecycle through the use of two important tools — source control repositories and static analysis filters; selecting rules based upon functionality, ownership, and severity level; assigning phase-in priorities to selected standards; and achieving full team buy-in and compliance when using standards.

Note: To learn more about the workflow and relationships between organizational roles in the Automated Error Prevention (AEP) methodology, see the Parasoft whitepaper "Understanding the Workflow in a Coding Standards Implementation."

Information about how the technology infrastructure and automation within Parasoft Enterprise Solutions support coding standards can be found in the Parasoft whitepaper "Configuring Your Automated Error Prevention Infrastructure."

Content

This paper is divided into the following sections:

1. **Introduction**: Outlines recent highly publicized software failures, and current process models that seek to prevent errors from arising during development.
2. **The Key To Success: Automated Infrastructures**: Discusses the benefits of automated coding standards enforcement.
3. **Implementing Coding Standards**: Examines best practices for implementing coding standards in an efficient manner that does not interrupt the software development lifecycle. These practices are examined in the following sections:
   A. Integrate coding standards into the development process.
   B. Introduce coding standards in a phased, "Divide and Conquer" approach. This includes functionality, ownership, severity levels, and phase-in priorities.
   C. Enforce development team participation.
4. **Conclusion**

Introduction

The long-standing debate over software quality flared in May 2002 when the Federal government disclosed a study that found that software bugs cost the United States economy nearly $60 billion dollars a year, an astronomical sum that suggests the exponential increase in software functionality in the last several decades has been devoid of true quality innovations. Conducted by the National Institute of Standards and Technology (NIST), the study notes that roughly 64% of this enormous cost is absorbed directly by the consumer, yet despite this tremendous burden there is no other consumer goods market where poor products enjoy such a vast popularity.[i]

Such complacency towards poor quality — by both the software manufacturer and software consumer — costs more than just money. In addition to the financial burden engendered by defects, bad software affects every part of our daily lives, disrupting productivity, posing health and safety hazards, and, in their most severe form, destroying both property and lives. The following list is a very small sampling of such events:

- Between June 1985 and January 1987, there were a half dozen serious incidents of radiation overdoses involving a computerized radiation therapy machine known as the Therac-25. These incidences resulted in at least three deaths. [ii]

- The scheduled 1993 opening of Denver International Airport was delayed more than sixteen months due to faulty baggage control systems. Software failure was the attributed reason. Faulty baggage handling continued to plague the airport years after it opened.[iii]

- On June 4, 1996, an Ariane 5 rocket launch ended in disaster when a software buffer overflow error occurred. This resulted in the loss of the rocket and its $500 million satellite. The loss was attributed to the use of inertial reference software on the Ariane 5 that had been designed for the smaller Ariane 4 launcher.[iv]

- In September of 1997, the United States Navy's new "smart-ship," the *U.S.S Yorktown*, was incapacitated for several hours by a computerized navigation system that froze when given incorrect data. When the problem could not be immediately correct the ship had be unceremoniously towed to port.[v]

- NASA lost the Mars Climate Orbiter in 1999 when the spacecraft entered the Martian atmosphere at the wrong angle. The loss was later attributed to the "failed translation of English units into metric units in a segment of ground-based, navigation-related mission software."[vi]

- NASA also lost the Mars Polar Lander/Deep Space 2 spacecraft in 1999 when the engines shutdown prematurely on final decent, leaving the craft and its two smaller probes to fall more than 100 feet to the Martian surface. The resulting impact most likely destroyed all three craft. Later reports indicated that faulty software sent bad signals to the landing legs, which when deployed, shut down the engines.[vii]

- In January 2003, more than 8000 patients recently discharged from Saint Mary's Mercy Medical Center in Grand Rapids, Michigan were incorrectly declared dead due to faulty insurance billing software. According to sources, a recent system upgrade of the software inadvertently deleted a zero in code that indicated the correct patient status.[viii]

Highly visible incidents such as these receive often embarrassing media attention, but there are literally hundreds more smaller incidences that occur on a daily basis that receive no attention at all. Cell phones, medical devices, smart cars with new interactive navigation systems, traffic lights, railroad crossings, municipal energy and power grids, and countless other products and large infrastructure facilities rely on software to make them work. Failure in these software systems has the potential to wreak havoc on a wide scale.

To counter poor software quality, many standardization and compliance organizations, such as ISO 9001 and SEI – CMM, have been created in an effort to bring the software development community into agreement on effective processes and procedures for creating usable, good quality software. CMM is popular within the software industry, as it directly addresses the five levels of process maturity that software manufacturers must fulfill in order to achieve development and quality process excellence.

CMM, unfortunately, is not an easy evolutionary process to follow, and very few companies have made it all the way to level 5, the highest level, where software is optimized through continuous improvement and defects are prevented before they reach the customer. Many companies reach the middle levels of the CMM and stagnate in their quality efforts because they have trouble implementing requirements in a standardized manner throughout their organization.[ix] In other words, they fail to automate an infrastructure that will ensure compliance to the organization's standard software processes.

The Key to Success: Automated Infrastructures

The most efficient automated infrastructure relies upon a combination of software that automatically validates the functional and structural integrity of other software, along with automation procedures that can be integrated into all stages of any software development process. Without automation companies are left to navigate the multiple levels of the CMM model using only standard quality management tools, which often do not consist of much more than a checklist and a written procedure of development and management activities. The potential for error generation (and regeneration) through manual methods puts at risk any benefits from standards compliance that might be otherwise enjoyed by both the company and the customer.

Implementing coding standards can be an over-whelming and difficult effort, one that can hinder the development process if a well-organized plan for deployment is not created. Having a defined set of coding standards in place for manual implementation is a good start. Such standards should address both organizational and individual project needs, and can even be supplemented by rules that address individual programmer needs.

However, it is not enough to simply legislate or require developers to use coding standards; an interactive and automated verification system is needed to ensure coding standard enforcement. The benefits of automated coding standards enforcement include:

- Streamlining source code control across multiple development teams or sites. Nightly code deposits and builds can be checked automatically by gatekeeper or filter program, such as Parasoft's CodeWizard™.
- Reducing potential for errors and programmer complacency through organization-wide compliance with all coding and development standards.
- Expediting organizational standards compliance for SEI-CMM, ISO, and other similar process models.
- Maximizing development team resources. New programmers can be put on important projects more quickly since standards compliance is automated.
- Reducing time spent on manual code reviews by both development and management team members.

Most importantly, automating coding standards makes error prevention both universal across the organization and constant. Automation makes possible a "Test early, test often" approach to software development that makes your entire organization proactive to error eradication rather than reactive to error correction. Without automation, complete, company-wide acceptance of standards and procedures is impossible to achieve.

Implementing Coding Standards

With today's diverse development environments and practices, few would argue with the importance of good coding practices, yet many organizations are slow to adopt such practices, or they are still learning by trial and error. Although it is widely accepted that good coding practices are essential to development projects, implementing coding standards is usually an afterthought; often the idea is dropped altogether due to unrealistic project constraints or simply because the time consuming efforts involved in manual code reviews continually precludes the automation of coding standards.

To avoid the consequences of not implementing such standards, or of getting sidetracked once such standards are collected but not put into practice, organizations need to keep three fundamental concepts in mind when creating, implementing, and automating good coding practices:

1. Coding standards must be integrated into the development process.
2. Coding standards must be introduced in a phased, "Divide and Conquer" approach.
3. Participation by the development team is mandatory and must be enforced.

These concepts are described in detail in the following sections.

1. Integrate Coding Standards into the Development Process

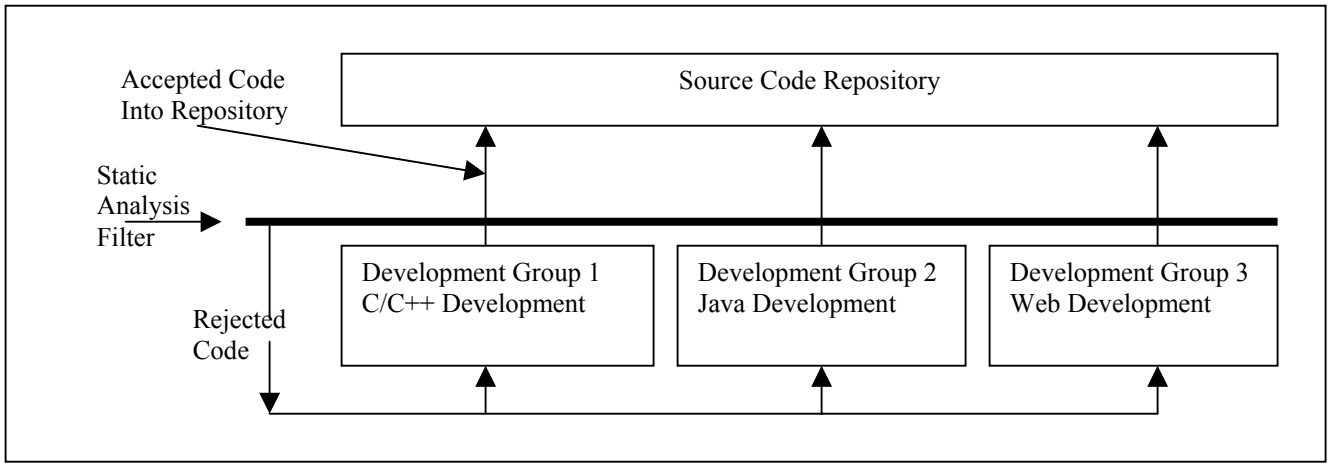
Coding standards must be integrated into the development process through the use of two important tools:

- A source code repository.
- A static analysis filter that verifies code before it is checked into the repository. This filter must have restricted access so that it is maintained and monitored only by the software architect and/or project manager.

The first of these tools, the source code repository, enjoys wide acceptance within the software development community, since a common storage area from which developers access their projects both streamlines development work and reduces the chances of losing or corrupting code. One of the benefits of a source control repository is that not only can you track the history of the code, but you can also improve efficiency by ensuring that revisions are not carelessly overwritten. The ability to revert back to archived versions also allows you to take risks with your revisions, and gives you the option of starting over again should it be necessary.

If you do not currently have a source code control system in place you should begin your error prevention efforts by establishing one. There are a number of source control tools on the market today. If cost is a factor there are also open source tools available, such as GNU RCS and GNU CVS; you can use these tools as is, or customize them to your team's specific needs by writing or modifying shell scripts

As depicted in the following figure, developers check-in their source code into the repository upon completion of their daily tasks:



The most important feature in the above graphic is the static analysis filter, which serves as gatekeeper to the source code repository by analyzing and rejecting any code that has not passed a strict automated code review. Parasoft's CodeWizard should be used as such a filter — it acts as a first line of defense against errors by immediately rejecting code that does not conform to the accepted coding practices of either the development group or of the company as a whole. Code that has been deemed “clean” by the gatekeeper is much less problematic when it comes to more advanced testing procedures, such as unit testing, functional testing, and load testing; when all coding errors have been corrected only more complex errors remain, such as memory corruption, memory leaks, and so on. The overall testing effort is streamlined when coding errors are corrected early and often.

Together source code repositories and gatekeepers control automated nightly application builds. As such, they are not simply control code reservoirs but the first line of application testing. The minimal nightly build should extract all necessary code from the source code repository, compile that code, and build the application. The ideal nightly build should also run all available coding standards and report any rule violations that occur.

Such precautions ensure that all developers are following prescribed practices during the day and that any discrepancies are flagged before they corrupt further code additions. Most importantly, they minimize the overhead involved in assembling the various application pieces and keep unnecessary bug searches, and subsequent project delays that result from such searches, to a minimum. More detailed information concerning the integration of coding standards into the development process, including setting up source control filters, can be obtained by contacting Parasoft Professional Services at (888) 305-0041 or proserv@parasoft.com

2. Introduce Coding Standards in a Phased, "Divide and Conquer" Approach

Once the source code repository and static analysis filter are in place, it is time to populate the filter with coding standards. These standards should be introduced in a phased, incremental approach in order to not disrupt the development cycle. Points to consider when selecting appropriate coding rules are:

- Functionality
- Ownership
- Severity levels
- Phase-in priority

The following sections describe the process of breaking coding standards implementation into manageable units by completing each of these phases. The overall focus is determining your coding standards needs — what rules best fit your development style from an organizational standpoint and also for individual development teams, and what the priority level for these rules is with regard to your implementation timeline. When implementing the "Divide and Conquer" approach, it is assumed that an organization's development team manager or architect will have already asked some focused questions concerning an organization's commitment to error prevention and the role coding standards are to play in that commitment:

- How does our organization currently prevent coding errors from occurring (or reoccurring?).
- What is the best way for our entire organization to implement coding standards?
- Does our organization have a set of established coding standards?
    - If so, are they customizable?
    - Can these coding standards be referenced in a specification document?
    - Is the document fully institutionalized? That is, does management understand its importance and expect development to follow it?
- Do our developers participate in customizing coding standards so that they reflect actual standards and practices?

It is best to be as introspective as possible when introducing coding standards into a development team or organization. You must determine how your company confronts coding errors in order to determine the best method for eradicating those errors. For many companies this first step requires honesty and a great deal of patience. Companies that assume that they have some sort of error prevention methodology in place may be unpleasantly surprised to find that no such methodology exists, or that it exists in piecemeal fashion through a loose collection of individual developer rules.

Such unfounded assumptions point directly to a breakdown in communication between management and development, which is why it is necessary to begin this automated error prevention effort by addressing such basic issues as a team (both management and development working together). Having a list of coding standards for your developers to follow and actually providing them with a way to follow those standards are two very different things; asking detailed questions such as the ones listed above will simultaneously orient both management and development to the error prevention task at hand and provide an initial scope to any coding standards project.

If your company does not have a specification document that completely defines its approach to coding practices, we recommend using Parasoft's CodeWizard, which includes a well-defined, built-in rule set as a starting point for such a document. There are over 300 rules included in CodeWizard covering many different aspects of the C and C++ languages. You can customize this rule set by adding to it your own company rules or you can create your specification by taking from CodeWizard only those rules that you actually use in your development activities.

If your company or organization already has a coding standards document in place, then you should map your company's rules with the standard set included with CodeWizard. Mapping helps prevent duplication of rules, as well as pointing out which rules in your customized set need to be transferred into CodeWizard prior to using it as a static analysis filter. Parasoft can do this for you through its customized Professional Services program (proserv@parasoft.com).

Functionality

If your organization already has a basic set of rules that developers follow when coding, then the next step is to arrange those standards according to functionality.[x] Classifying your coding standards by functionality allows you the flexibility of identifying what areas within certain projects may need additional rule support. The metrics developed by this classification can be used to make determinations for resource allocation on a per-project basis.

The following table offers an example of an initial functional classification document. Keep in mind that this table does not constitute a complete representation of every functional category with the C and C++ coding languages.

Category	Rule
Shifting from C to C++	Prefer iostream.h to stdio.h
	Use new and delete instead of malloc and free
Memory Management	Use the same form in corresponding calls to new and delete
	Call delete on pointer members in destructors
	Check the return value of new
	Adhere to convention when writing new
	Avoid hiding the global new
	Write delete if you write new
Classes and Functions: Design and Declaration	Differentiate among member functions, global functions, and friend functions
	Avoid data members in the public interface
	Pass and return objects by reference instead of by value
Constructors, Destructors, and Assignment Operators	Define a copy constructor and assignment operator for classes with dynamically allocated memory
	Prefer initialization to assignment in constructors
	List members in an initialization list in the order in which they are declared
	Make destructors virtual in base classes
	Have operator= return a reference to this*
	Assign to all data members in operator=
	Check for assignment to self in operator=

This table is a simple list of rules frequently used by a given development team. That there are only two rules included in the "Shifting from C to C++" category is not a problem if such a shift has not been undertaken, or if it is not in the immediate future for that team. However, should a wholesale shift from C to C++ become necessary, then having rules mapped by category will enable this team to see at a glance that additional rules will be needed in order to make this transition as smooth as possible.

A functionality review should be performed frequently, approximately every six months, so that the rules specification document does not become obsolete. For example, should your organization choose to transition to 64-bit architectures, then an entirely new category of rules would be needed, which could be called "32-bit to 64-bit Porting Rules." This category could include such rules as those listed in the following table, among others of this type:

Rule	Description
3264bit_32BitMultiplication	Do not assign the value 32-bit multiplication to type long
3264bit_CastPointerToUINT	Do not cast pointer to UINT type
3264bit_IntPointerCast	Do not cast pointer to int type
3264bit_IntToLongPointerCast	Incompatible cast
3264bit_LongDouble	Possible truncation
3264bit_LongToDoubleCast	Possible truncation
3264bit_Union	Avoid union if it is possible
3264bit_UsingLong	Avoid using long

Ownership

Once your rules are classified by functionality, then you should divide them into the three ownership groups listed in the following table:

Group	Description
Organization	These coding standards represent those rules that everyone must follow, regardless of project or development group. These rules are language-specific standard; any violation of these standards results in unpredictable application behavior. Examples of organization rules include "Define a copy constructor in assignment operator for classes with dynamically allocated memory" or "Assign to all data members in operator =".
Project	These coding standards represent those rules that a particular project or development team must follow, in addition to the organizational rules. If used frequently enough project rules could become organizational rules, but generally these rules are deviations to the organizational rules that a project team must follow in order to adhere to customer application requirements. Examples of project rules include "Have operator= return a reference to *this" or "Do not use i64 or L suffixes directly."
Individual	These coding standards represent those rules that individual programmers define for personal use. The types of coding standards that fall into this classification depend upon the individual developers skill level, as these rules are usually built to serve as corrective anecdotes for frequently made coding errors. Examples of individual rules include basics such as "Do not cast pointers to non-pointers" or "Provide only one return statement in a function."

By placing your coding standards into these categories, you have a much-needed view of how and where your coding standards are being utilized throughout your organization. It is not enough to say that you have coding standards documented; you must avoid ambiguity as to what these standards are, what the functional purpose is, and where these coding standards are being utilized throughout your organization. The ownership demarcations provide, at a glance, confirmation of where your coding standards are being employed.

Severity Levels

After your coding standards have been divided by ownership categories, you must determine the severity level of each standard. This allows you to prioritize your coding standards and deploy them in order of importance. The five severity level categories are based upon the likelihood of bugs being introduced into your application if a particular rule is violated or skipped.

The following table represents the severity level rankings deployed by Parasoft's CodeWizard. Order of severity importance is numerical, that is, a Severe Violation (#1) always results in an error. Each category thereafter represents rules violations that have a progressively lower probability of resulting in an error, down to the Informational category of rules (#5):

Rank	Name	Description
1	Severe Violation	This category of errors represents a severe violation of a rule or rules. This category has the greatest chance of resulting in a bug.
2	Violation	This category represents a violation of a rule or rules and may result in an error or unpredictable behavior.
3	Possible Severe Violation	This category represents possible severe violations of a rule or rules and will most likely result in an error or unpredictable behavior.
4	Possible Violation	This category represents a possible violation of a rule or rules and could result in an error or unpredictable behavior.
5	Informational	For information purposes only. This category of errors has the least chance of resulting in a bug.

When introducing coding standards into your development process you should implement rules incrementally based upon this order of severity. Implement those rules that have the greatest probability of preventing an error and are of extreme importance first, then implement those rules that have a lower probability of preventing an error and are less critical. You can implement rules based on single categories (one category at a time, from 1 to 5) or group them together for a faster implementation (categories 1 - 2 first, then 3 - 4, and finally 5, or 1 - 2, then 3 - 5).

As your team or organization becomes comfortable with the first set of standards and avoids violating those rules, you can enforce the next category. When developers are comfortable with that set, they can enforce the next set, and so on. No

matter what stage of enforcement developers are in, all appropriate coding standards should be passed before code is checked into the source code repository. This prevents error-prone code from spawning errors and becoming increasingly error-prone and difficult to manage/develop, and it forces developers to face and fix problems before they continue to introduce the same problem into new code.

Phase-In Priority

Introducing your coding standards in stages creates an environment that is conducive to good coding practices, allowing development teams to begin using important or critical rules first, then less important rules, and so on. This approach also allows development teams to ramp up their coding standard initiatives in a gradual manner without creating unreasonable workload expectations. Coding standards phase-in priorities should be defined as shown in the following table:

Level	Description	Phase-In
Must Have Rules	These are high priority rules that must be implemented first in order to successfully complete a project or application. Examples include standard C and C++ language constructs ("Do not call delete on non-pointers") and rules that result in severe or unpredictable functional behavior when violated.	30 Days
Should Have Rules	These are medium priority rules that can be implemented after the severe "must have" rules have been chosen and automated. These are standard rules that result in erratic code behavior when violated.	60 Days
Nice To Have Rules	These are the low priority rules that can be implemented after the first two categories have been successfully automated. These represent rules that may not result in erratic functionality or code behavior when violated but that nevertheless result in poor coding constructs and reduced code maintainability.	90 Days

The gradual, phased approach of the 30-60-90 rule makes coding standards enforcement much more manageable; by tackling the most important or needed rules first, you make the application immediately viable. Once the application has become stable through the enforcement of the Must Have rules, you can concentrate on perfecting the code with the remaining categories.

3. Enforce Mandatory Development Team Participation

Full participation in coding standard enforcement by the development team is crucial — upon this fact hinges the entire automated coding standards enforcement effort.

Many software manufacturers assemble sets of coding standards because they recognize the need to adhere to good coding practices. However, assembling a set of coding standards and actually putting it into practice are two separate efforts. The old saying that "you can lead a horse to water but you can't make him drink" is easily applied to software developers and their use of coding standards — even the most meticulous set of coding rules is useless if not regularly used within and among development teams.

Changing this mindset is a dual-pronged task. First, project managers and architects must take the initiative in setting up an environment for good coding practices. The subjects touched upon in this paper are a good foundation for project managers and architects to use when institutionalizing automated standards. Second, complete participation in automating coding standards is a full team effort. Project managers and architects can facilitate coding standard implementation, but all team members must be involved in determining which errors most often enter their code and why. This is very important. For the entire team to buy-into standards and the automation effort, then standards and the eradication of coding errors must be a group goal. The entire development group or team must design, create, and enforce coding standards that prevent inaccuracies from occurring and reoccurring. At the very least, this must be done for all Must Have coding standards that a team deems are crucial to a given project.

If teams follow this strategy for every error that they detect, they will quickly build a set of coding standards that can prevent their teams' most common errors from ever entering their code. Development teams must be required to follow these prescribed practices throughout the day and before checking their work into the source code control system.

Once coding standards are implemented, and the automated infrastructure is set up to enforce them, then a short trial period can be used to determine the impact of those standards. Six months is a standard evaluation time. After six months the amount that errors have been reduced can be measured, and reports for upper management drawn up. If, at the end of this period, further rules need to be added or current rules amended, then the "Divide and Conquer" approach outlined in this paper can be restarted to fully evaluate where the process can be improved. Developers not using the system or new team members can be trained at this time, and experienced team members can give whatever feedback they feel will make the system even better. Project managers and architects must remember to initiate process improvements and inspections

periodically, getting full team participation in reviewing and updating the automated error prevention process in order to get the maximum benefit for their entire software development lifecycle.

Whatever your development processes — C/C++, Java, Database or Web development, Web Services — automated error prevention can be applied. If such a strategy is not used, errors will grow exponentially, affecting all areas of your production. Remember, automated error prevention is an *evolution*, not a *revolution*, of your existing development process.

Conclusion

Many software companies make a fatal mistake when implementing coding standards by greatly oversimplifying the task; many development shops erroneously believe that simply adopting some form of naming conventions — such as "begin class names with an upper case letter" — is sufficient. All too often a rule of this type is simply written into a house "style sheet" that each developer is expected to follow. Software companies that leave coding standards at this "honor system" level — where developers police themselves when it comes to following coding rules — have not even begun to address the implementation of such standards.

Other companies make the process much to burdensome on managers and developers alike by asking for too much too soon. Hundreds of rules are chosen and thrown at developers, who must figure out a way to use them all or risk never checking in their code. This burden quickly precludes any actual coding progress within and across development teams, and the process of using coding rules is quickly abandoned.

Regardless of which approach is taken, when such efforts turn sour (and they will turn sour, from both development and management neglect) their unfortunate response is to assume that coding standards simply do not work. However, it is their process of implementing coding standards that failed, not the standards themselves.

Companies that adopt a "Divide and Conquer" approach to coding standards implementation are much more successful in adopting good coding practices across their entire organization. The "Divide and Conquer" approach breaks down the enormous task of implementing coding standards into manageable units. These units are completed and adopted in successive stages, so that coding standards implementation and automation is introduced gradually rather than in one massive effort. This phased approach permits a great amount of leeway when considering the length and breadth of an organization's coding standards needs. Rules can be tested on different projects and adopted as is, altered, or dropped altogether if necessary.

Notes

[i] "The Economic Impacts of Inadequate Infrastructure for Software Testing." Washington D.C., National Institute of Standards and Technology (NIST), 2002. Report available online at http://www.nist.gov.

[ii] Eby, Mark. "Therac-25: The Treatment That Killed." Online report available at http://www.uoguelph.ca/~meby/.

[iii] Dempsey, Paul Stephen. "Airport Woes A Wakeup Call." Denver Business Journal, January 8, 1999. Online report available at http://www.bizjournals.com. This article contains examples of the failure of major technological systems that in recent years delayed or hampered the opening of several international airports.

[iv] "ARIANE 5: Flight 501 Failure — Report by the Inquiry Board." Professor J.L. Lions, Chairman. Paris, July 19, 1996. Online report available at http://java.sun.com/people/jag/Ariane5.html.

[v] Slabodkin, Gregory. "Software Glitches Leave Navy Smart Ship Dead in the Water." Government Computer News, July 13, 1998. Online report available at http://www.gcn.com.

[vi] Isbell, Douglas and Don Savage. "Mars Climate Orbiter Failure Board Releases Report . . ." NASA press release, November 10, 1999. Available online at http://mars.jpl.nasa.gov.

[vii] Stenger, Richard. "NASA Report: Software, Tight Budget Doomed Mars Lander." CNN Online, March 28, 2000. Article available online at http://www.cnn.com.

[viii] "Hospital Computer Makes Fatal Error," CNN Online. January 8, 2003. Article available online at http://www.cnn.com.

[ix] The middle tier in the CMM process model, level 3, is the "defined" level of software development, in which management and engineering activities "are documented, standardized, and integrated into a common software process for the entire organization. At this level all projects use an approved, tailored version of the organization's standard software process for developing and maintaining software."

[x] Classification by functionality is only necessary if you are using your own predefined coding standards. If you have elected to use the recommended coding standards built into Parasoft's CodeWizard you will not need to perform this step as classification by functionality is already built into CodeWizard.