

Defining a “Safe Code” Development Process

Stephen D. Morton
Applied Dynamics International

January 4, 2001

ABSTRACT

“Safe code” is characterized by its fitness for use in situations where software is in direct control of human lives or safety. As the use of embedded software in such situations rises, the need for strong safe code development processes in the embedded software industry increases correspondingly. Understanding what constitutes a safe code development process and how to implement such a process is an important element of the industry’s capability to fulfill this need. Herein are discussed such details, with an eye toward helping the industry draw its own roadmap. In the end, only informed choices can be the building blocks of a safe code development process.

INTRODUCTION

The world is getting “softer”. There is no mistaking the trend, either. Everywhere one turns a new or novel use of software is springing to life. Some software is overt, openly proclaiming “I am software! Hear me roar!”, but there is a whole world of covert software out there, hiding in plain sight.

Embedded software is all around us. It is in the programmable thermostat that automatically reduces household temperature at night. It lives in our cell phones and text-messaging pagers, quietly keeping us connected. It exists in the aircraft engines that power our planes, and in the avionics the pilots use to guide us to our destination. Software even controls our cars, from their electronic ignitions to modern cruise controls.

Thermostats and cell phones are important in our lives, but it is a rare occurrence when either device can actually inflict harm on us. Not so with aircraft or automotive software controllers, though. Clearly, the software in these applications must be held to a higher standard; A safety standard.

In truth, embedded software is becoming a major component in safety-related applications more than ever before. Safety-critical code once was confined to major systems, such as commercial and military aircraft. This is no longer true. Everyday applications, like the elevator in a modern hotel, are under software control. And such everyday applications are responsible more than ever before for the health or safety of human beings.

Correspondingly, there is a growing demand for “safe” code; Code that can be entrusted with precious cargo or responsibilities without worry that the program will cause irreparable harm. The aerospace industry has led the charge, under the dictates of the Federal Aviation Administration (FAA). But how can all embedded software reap the benefits of aerospace’s lessons learned?

The answer is a journey called ‘process’.

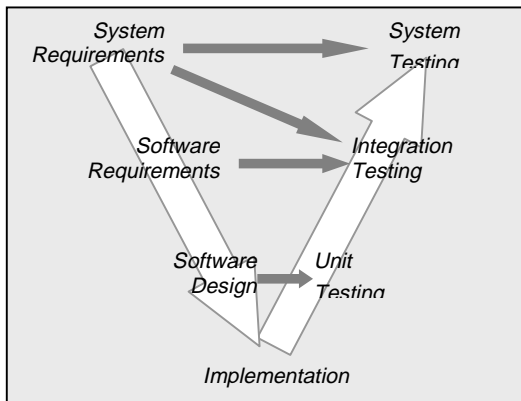
PROCESS DETERMINISM

Every programmer has a development process. Some don’t know this fact. Others prefer to ignore it. Proficient programmers in safety related arenas embrace sensibly strict processes as the vehicles that carry them down the software development turnpike. Only the true novice, never before having written software to solve a problem, doesn’t have at least a preferred *modus operandi*.

When safety is a guiding principle for software, the *method of operating* needs to be much more than a personal preference. The development process for safety related software must be a formally defined and rigorously enforced sequence of logical steps, each designed to mitigate the risks inherent in the application. The process must be written down, using language that is clear and easy to understand. It must also enunciate the goals to be achieved by following the process.

A VISUAL AID

Modern safe-code development processes are often described visually using the standard V process model, depicted below.



On the development side of the model, to the left of the implementation, each succeeding document, whether it be a requirements or design specification, is a derived artifact from those above it. For instance, the software requirements are distilled from the system requirements. The software design is then drawn from the software requirements. Finally, the implementation is created from the software design.

The implementation is the final artifact of the design process. It is also the base artifact influencing every step to its right, on the test side of the V.

Unit testing draws its test requirements from the detailed software design and applies them against the implemented code. Similarly, integration level testing draws its requirements from the software and system requirement specifications and applies them against logically related groups of units. Finally, system level tests are drawn from the system requirements and applied against the software (and often hardware) as a whole.

On the test side of the model, however, each succeeding level of test is predicated on the successful completion of the preceding levels. This can be visualized as a pyramid, with unit testing at the bottom and system testing at the top. The width of each step is indicative of the number of pieces to test (system testing has one, while unit testing has many), but inversely related to the scope of the test itself.

DEFINING SUCCESS

One of the key questions to be answered in selecting a process is the definition of “success”. Creating ‘perfect’ software is an impossible definition for success. After all, programmers write software, and I don’t know any ‘perfect’ programmers, including and especially myself!

Perhaps this definition can include a specific defect rate not to be exceeded, using a measurable metric as its gauge. Sometimes success is governed by regulatory certification, as in the aerospace industry. Depending on the application, success may even be best defined as drastically reducing any possible product liability.

The motivation for developing safe code will dictate the definition of process success. This definition needs to be drawn, therefore, by the software developers, managers, and customers in concert in accordance with the needs of the application under development.

PROCESS FOCUS

There are some key characteristics to safe code development processes that are universal, despite the variant definition of success. These characteristics include:

- Unambiguous and *testable* requirements & design.
- Requirements & design validation activities.
- Implementation verification activities.
- Mandatory feedback.

TESTABLE REQUIREMENTS

Ambiguity is the antithesis of testability. If the action required on the part of an embedded controller under any specific set of circumstances is not clearly and concisely defined in the requirements, then the action taken in the implementation of those requirements is at the discretion of the programmer.

As an example, consider that shifting the bits of a signed integer using the ‘>>’ operator is an implementation dependent characteristic of the C language¹. The

¹ MISRA C Rule 37 explicitly prohibits the use of bitwise operators on signed integers.

problem with this scenario is that the right-shift operator may, or may not, move the sign bit from its rightful place into the value places within the defined word length. It all depends on how the compiler was implemented. The C language specification does not tell how to handle it. Clearly, porting embedded code from one compiler to another in this situation involves more risk than is acceptable for a safety related application.

Requirement testability means that the specification must be comprehensive, covering every conceivable scenario, with fallback requirements that cover inconceivable scenarios. For a requirement that specifies turning on an indicator light when the fan speed reaches 10,000 rpm to be comprehensive, the requirement must also say what happens when the fan speed declines below the 10,000 rpm threshold. Does the indicator lamp stay illuminated when the sensor fails? All pertinent questions must be asked and answered to assure a comprehensive and testable requirement.

Testability also means that the requirements must be measurable. For example, It is not sufficient to state that the controller must exert control *in a timely manner* after a power-on reset occurs, but that it must take control *within 300 msec* in that situation. Without being a mind reader, it would be impossible to know what a “timely manner” is without an explicit value.

It is beyond the scope of this article to fully define requirement testability, but cognizance of this need is a critical element in creating an equitable safe code development process.

VALIDATION

Requirements and design validation is an important concept in ensuring requirements testability. Succinctly put, validation is the examination of a software requirement or design for self-contradiction, ambiguity, and the *ability to perform its assigned task*. While these first two factors are objective in nature, the ability to perform the assigned task is more subjective, and necessarily requires a higher degree of sophistication and experience in its evaluation.

To use a previously cited example, consider a jet aircraft engine control with a 300 msec time limit on power-on reset recovery. If the engine is projected to become unstable in one or more operating states 250 msec after a power-on reset, then the requirement is invalid. In fact, the specification would be considered self-contradicting.

Once again, a full explanation of software validation is well beyond the scope of a general overview such as is presented in this article, but understanding the need for requirements and design validation is a key concept when defining a process.

VERIFICATION

Implementation verification is vastly different from requirements and design validation, although the two tasks are often lumped together conceptually. Verification involves demonstrating that the application software successfully implements the software requirements and design, whereas validation is an evaluation of the requirements and design to be implemented.

Software testing is extremely important in verifying an implementation. Using a multi-tiered testing approach, including unit, integration, and system level testing, provides a comprehensive look at the structural and functional aspects of the software. Each level of test accomplishes distinct and valuable goals, and succeeding levels presume successful accomplishment of the preceding levels.

Analysis is nearly the equal of testing in value with respect to software verification, despite the fact that analysis is often an afterthought for the software developer. Regurgitation of the structural or functional details of an implementation can be instrumental in helping the software designer to verify that the software does what it is expected to do. More importantly, analytical input early in the process allows the software designer to evaluate the design before putting the software out to test or market. To quote the popular saying, “An ounce of prevention is worth a pound of cure”.

Verification, in terms of a well constructed safe code development process, should be a balance of analysis and testing. Analysis provides ongoing and immediate feedback

to the designer/developer that should agree with the requirement being implemented, as well as the designer's intent. Testing is used to provide formal proof that the implementation matches the requirement(s) and designer's intent, but does so much later in the development process.

FEEDBACK

Knowledge is power. Shared knowledge is increased, not diluted, power. Utilizing properly drawn, formalized reporting techniques throughout the development process ensures that validation and verification results are spread amongst all who have a stake in the software's development.

"Formalized" should not be confused with "formal". In some development contexts, a specific format for reporting issues and results is dictated by regulatory or legal mandate. This constitutes formal feedback. Formalized feedback, on the other hand, is the simple act of requiring some form of reporting under the process rules. The formality of required reports is a subject governed by the legal and regulatory aspects of the software's development, and must be spelled out in the process definition documentation.

Feedback should be prevalent throughout the development process. On the design side of the process, formal reviews or analytical reports may form the basis of the requisite feedback. On the test side, pass/failure reports from the executed tests and analytical byproducts of test design are common feedback methods. Whatever method is chosen for the process, it is important to ensure that feedback is both immediate and ongoing.

FOLLOW-THROUGH

No matter how rigorous the defined process, if the rules are not enforced, the process cannot ensure realization of its definition of success. Judicious selection of the process and sensibly rigorous enforcement of its rules are codependent factors in achieving success. Building realistic safeguards into the process is the only way to ensure adequate enforcement without creating an overload of minutiae for those involved with developing the application.

In most safe code development process, proper follow-through is ensured by using a system of gateways or milestones that are high-profile, drawing the attention of all levels in the development organization. If a particular milestone is not reached on schedule, then subsequent development activities are easily identified as at-risk. For instance, if the detailed design review (often called the Critical Design Review or CDR in aerospace processes) is not accomplished for two weeks after it was originally scheduled, then code development and all testing activities are known to be two weeks late in starting.

It should be apparent that follow-through is served in part by the feedback aspect of the development process. Mandatory sharing knowledge of status, risks, and success will accentuate the benefits of following the process, and thereby heighten the tendency to do so.

SETTLING ON A PROCESS

When a project is looking to establish its development process it is often necessary to make tradeoffs. Every benefit has a cost, but not every cost has a benefit. The optimum blend of rigidity and flexibility in a process is difficult to achieve, and is likely to involve an ongoing evaluation of the process under use, with modifications applied as necessary to achieve ultimate and ongoing success.

PARADIGM SELECTION

Perhaps the first decision in selecting a process is what paradigm to follow. The two major paradigms currently in favor have distinct advantages and disadvantages, and exceptional care is important to ensure that the most beneficial is selected according to the needs of the software under development.

The two paradigms, being model-based development and specification-based development, must be clearly understood before they can be decided between.

MODEL-BASED DEVELOPMENT

Model-based development involves the use of a simulation model as the requirements to be implemented. This means that the *behavioral characteristics* of the model are

used to define the required behavioral characteristics of the application. This distinction is important as a discriminator between the use of a graphical specification language (as discussed in the following section) and true model-based development. The automotive industry is actively embracing model based development, with encouraging results thus far.

Using a commercially available modeling package, a behavioral definition of the application is created as the golden rule for software development. The application's implementation is then written, replicating the behavioral and functional aspects of the modeled requirements. Finally, the implementation is measured against the model, with special emphasis on measuring quantifiable behavioral characteristics embodied in the model. This paradigm is sometimes labeled as the development of an "executable specification".

It should be understood that the application's implementation may be structurally different from its model. In fact, the implementation's construction need not resemble that of the model at all, provided that the behavioral and functional characteristics of the model are properly replicated. The implementation's structural characteristics are governed by the software design, which may be created external to or incorporated within the model.

SPECIFICATION-BASED DEVELOPMENT

Specification-based development entails the usage of written requirements (in a specific human or graphical language) to specify the requirements to be implemented. This paradigm is more universally accepted in the aerospace industry, although model-based development is gaining in popularity.

Specification-based development is characterized by the "shalls", "shoulds", and "wills" of its language. Interestingly, specification-based paradigm usage appears to correlate with the need for regulatory certification of the application under development.

One relatively new invention for specification-based development is the use of graphical software requirement/design languages. These languages allow the

graphical depiction of both the software requirements and the design that implements them. Control-dependent requirements can be depicted in a flowchart, giving explicit statement ordering control to the designer. Computation-dependent requirements can be depicted using signal flow diagrams, where high-level constructs such as filters, integrators, and derivatives are often incorporated as a single "block". Using a graphical specification language can be a clear and concise way to document the commitment of software requirements into the application's design. To paraphrase the old saw, "A picture is worth a thousand lines of code".

Use of a graphical software requirement and design language can blur the line between model-based and specification-based development. Perhaps the future may hold the birth of a hybrid development paradigm, encapsulating the benefits of both base paradigms.

RIGOR VERSUS FLEXIBILITY

Process rigor emphasizes safety. If the rules are rigid enough to ensure no deviation, then the goals of the process will always be achieved, at least in theory.

On the other hand, flexibility allows for innovation. It also enhances follow-through, which is a key to the success of the process. Excessive flexibility, however, renders a valid process ineffective, easily evaded and invalidated.

Striking a balance between rigor and flexibility is therefore important in defining a safe code development process. The process must be enforced, but provisions for implementing acceptable deviations must also be included. How much enforcement is too much is a subjective decision, dependent on the character of the development organization as well as that of the individual developers.

THE DEVELOPMENT ENVIRONMENT

Does the application have to be certified under some specific regulatory guideline? How direct is the software's ability to cause human harm? Is there a formal standard that must be followed? Will following a certain standard enhance the product's image, becoming a "selling point"?

These are all germane aspects of the application's development environment, at least when contemplating the selection of a development process. It should be obvious that regulatory requirements must be accounted for in the selection of a safe code development process. Similarly, the need to mitigate the risk of human harm is a de facto requirement to be considered in defining a process. It is less obvious that marketing considerations also play an important part in the selection of a process.

The point to be understood here is that software development process selection can be instrumental in enhancing the market for the application. If the process is not dictated by a regulatory standard, then marketing considerations may be the deciding factor in process selection. Responsibility for considering this aspect lies with project management.

PROCESS IMPLEMENTATION

Once a process paradigm has been selected and the process is defined, process implementation enters the arena. In some ways process definition is the easy part. Getting software developers to sign onto a new process can be a staggering effort. Equally difficult is the task of putting in place the infrastructure to support the new process.

The availability of commercial or proprietary tools to support the process is a key consideration. Impending tool releases may be considered as well, although the immaturity of a new tool may be a drawback. In either case, automating aspects of the process should be a serious consideration when defining a process.

Equally serious is the need to consider the lessons learned by those who have trod similar development paths before. It is very true that those who ignore the mistakes of the past are doomed to repeat them. When defining a safe code process, investigate and consider the best practices of projects and companies who produce similar, even competing, products.

BUILDING THE INFRASTRUCTURE

Infrastructure requirements must be considered when selecting a process. Nothing is gained if the optimum process is

selected while lacking the ability to support its needs. COTS (commercial, off the shelf) tools or products that can be used as a basis for the implementation of that process are probably the most cost effective method of building an infrastructure. Tool development, after all, is expensive, and incurs ongoing costs in maintenance and upgrades that can be prohibitive.

Even still, bringing in "expert" tools from outside your organization suffers from the same drawbacks as bringing in a human expert. Namely, the NIH, or "Not Invented Here", syndrome is likely to rear its ugly head. Salesmanship on the part of the process definers, getting the developers to buy into the process objectives and rules, is the key to ameliorating this problem.

A similar consideration is the speed of change inherent in implementing the new process. Contrary to the popular saying, familiarity breeds not contempt, but comfort. Moving from an ad hoc or inferior process to a superior one should be done in stages, changing as much as is acceptable at any one point in time, but no more than can be chewed and swallowed in one bite. What is acceptable will be defined by the people involved in following the process, and may come down to a judgement call by the implementers of the process.

HOW DOES SAFETY FIT IN?

Wait a minute! I thought we were discussing the development of safe code! Isn't that the subject at hand?

Yes, as a matter of fact, it is. Safe code development means selecting product safety as the definition of "process success". Accordingly, the process steps and guidelines selected for implementation must be molded to support ultimate success. In doing so, one must always ask (*and answer*) the eternal question – "Is it safe"?

This may seem axiomatic, but it is an often overlooked aspect of adopting a new technology or development approach. No matter the source of the new idea, and no matter the efficacy of the new idea, the idea must be positively evaluated with respect to its safety before it can be implemented.

One tool that can mitigate the risk of implementing a new idea or process is the utilization of commonly accepted standards

or language features. There are numerous standards accepted for use in safety related software development, including the MISRA C standard and the SPARK Ada standard. Type-checking and other aspects of tools should also be evaluated with respect to the safety requirement of the final product. The possible evaluations that can be used to enhance process safety include:

- Ada versus C for source code language.
- MISRA C² versus ANSI C.
- SPARK Ada³ versus Ada 83 or Ada 95.
- The RTCA/DO-178B guidelines for software certification⁴.
- Company internal standards.

CHANGING THE MIND SET

Process redefinition can be considered the mid-life crisis of software development. Telling a software developer that they need to change the way they pursue their chosen profession can be a traumatic experience for even the most experienced programmer.

As the definer of a new software development process, you should be aware that the universe is squarely centered on your own desk. You must become the loudest cheerleader for the new process. You must also be willing to shoulder the responsibility of making the new process work, including smoothing out the difficulties that arise from tool problems and personal objections from developers.

Starting from the center of the development universe, being your own desk, travel in increasingly larger concentric circles, implementing and promoting the new process as you go. Your enthusiasm for and belief in the safe code development process will be infectious, spreading amongst those you come in contact with in direct relationship to your own fervor.

² The MISRA C standard is available from the Motor Industry Software Reliability Association ISBN 0-9524156-9-0.

³ SPARK Ada refers to the Spade Ada kernel.

⁴ The FAA has selected RTCA/DO-178B as its current standard for aerospace certification of systems including software as a component.

Along the way, ask and answer the eternal question – Is it safe? Help developers to get into the same frame of mind, always seeking application safety in order to heighten the prospect of process success. Is it safe? Would it be safe? Is safety compromised by this idea? Don't be afraid to ask the question, and then provide the answer. Therein lies the key.

Finally, seek out and use the best safety-critical tools available on the market. Such tools are designed to ask the eternal question for you, and to disallow unsafe practices. They will become your allies in promoting and implementing your new safe code development process.

SUMMARY & CONCLUSIONS

Safe code development is a process dependent sport. The end goal of safe code development is the fielding of software driven systems that inherently promote human safety over all other aspects. Safe code development processes err on the side of caution, when error is necessary, but attempt to anticipate the situation that could force such errors.

Learning from history and pioneers of safe code development will be a lasting key to finding a successful safe code development process for your application. No one is an island, and everyone can learn from the pitfalls of the past. The tools developed to counter past problems elsewhere can be instrumental in solving looming issues in your own arena, provided that the problems can be anticipated while constructing the process.

Finally, be your own cheerleader. Support your process after you have judiciously selected it. Smile while you enforce the rules, but remain flexible where appropriate. In short, be smart. After all, demonstrating intelligence is the surest route to lasting success ever known to man.