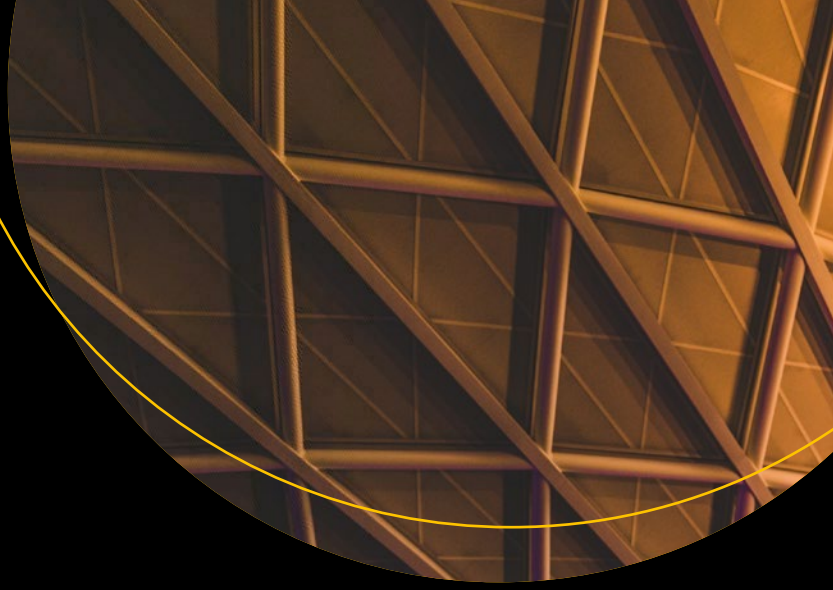

Patterns in the Machine

A Software Engineering Guide to Embedded Development

—

John T. Taylor
Wayne T. Taylor

Apress®

Patterns in the Machine

A Software Engineering Guide to Embedded Development

John T. Taylor
Wayne T. Taylor

Apress®

Patterns in the Machine: A Software Engineering Guide to Embedded Development

John T. Taylor
Covington, GA, USA

Wayne T. Taylor
Golden, CO, USA

ISBN-13 (pbk): 978-1-4842-6439-3
ISBN-13 (electronic): 978-1-4842-6440-9
https://doi.org/10.1007/978-1-4842-6440-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Callum Wale on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484264393. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To Sally, Bailey, Kelly, and Todd.
—J.T.

Table of Contents

About the Authors

John Taylor has been an embedded developer for over 29 years. He has worked as a firmware engineer, technical lead, system engineer, software architect, and software development manager for companies such as Ingersoll Rand, Carrier, Allen-Bradley, Hitachi Telecom, Emerson, and several start-up companies. He has developed firmware for products that include HVAC control systems, telecom SONET nodes, IoT devices, microcode for communication chips, and medical devices. He is the coauthor of five US patents and holds a bachelor degree in mathematics and computer science.

Wayne Taylor has been a technical writer for 25 years. He has worked with companies such as IBM, Novell, Compaq, HP, EMC, SanDisk, and Western Digital. He has documented compilers, LAN driver development, storage system deployment and maintenance, and dozens of system management APIs. He also has ten years of experience as a software development manager. He is the coauthor of two US patents and holds master's degrees in English and human factors.

About the Technical Reviewer

Jacob Beningo is an embedded software consultant with over 15 years of experience in microcontroller-based real-time embedded systems. After spending over ten years designing embedded systems for the automotive, defense, and space industries, Jacob founded the Beningo Embedded Group in 2009. Jacob has worked with clients in more than a dozen countries to dramatically transform their businesses by improving product quality, cost, and time to market. He has published more than 200 articles on embedded software development techniques and is a sought-after speaker and technical advisor. Jacob is an avid writer, trainer, consultant, and entrepreneur who transforms the complex into simple and understandable concepts that accelerate technological innovation.

Jacob has demonstrated his leadership in the embedded systems industry by consulting and working as a trusted advisor at companies such as General Motors, Intel, Infineon, and Renesas. He also speaks at and is involved in the embedded track selection committees at ARM Techcon, Embedded System Conferences, and Sensor Expo. Jacob holds bachelor's degrees in electrical engineering, physics, and mathematics from Central Michigan University and a master's degree in space systems engineering from the University of Michigan.

In his spare time, Jacob enjoys spending time with his family, reading, writing, and playing hockey and golf. When there are clear skies, he can often be found outside with his telescope, sipping a fine scotch while imaging the sky.

Acknowledgments

We'd like to thank Mike Moran for taking the time to read through early drafts of this book and for providing his usual insightful comments.

Preface

The mastermind behind this book is John. John is the one who has been working and developing code in the embedded systems space for 30 years, and this is his approach to developing software. *Patterns in the Machine*, or PIM, is his development process. If you worked with John, you'd see that his processes and his production code follow exactly what is prescribed in this book. While not all of his colleagues have been converted to his PIM approach, they can't argue with his success. John not only develops a prodigious amount of code, but he also keeps an amazing number of modules and unit tests and simulator bits current on his projects. And he does so by practicing what he preaches.

Consequently, in this book, when you see a phrase like "In my experience …" or "I worked on a project once …," it is usually John speaking. Occasionally, it's me editorializing or providing my own anecdote, but mostly it's John. We wrestled for a while about how to best alert readers to who was who, and, in the end, we decided we'd go with first-person narration. We never really were comfortable using "we." To us it sounded a little pretentious, pontificatory, and too much like the "royal we." So, when you read this book, know that "I" is John … mostly.

John and I have been involved with software development for a very long time. I wrote my first computer program in the basement of the Kiewit computer center at Dartmouth College when I was 10 years old. John was 7. Over the years, John and I have programmed in machine language, assembler, C, C++, Java, C#, Python, Perl, and so on. We've been involved in projects that range from firmware for very small hardware platforms to enterprise software for insanely large storage platforms. And while the specifics of our experience vary, we discovered over the course of writing this book that our ideas and conclusions about what constitutes smart development, what demonstrates elegance in design, and what is "the right way to do things" are surprisingly similar. When it comes to software development and programming languages, John and I are native speakers. And we speak with one voice.

—Wayne Taylor, Golden, Colorado, October 2020

CHAPTER 1

Introduction

This book is about how to be a genius—or, at least, how to design and implement software in a way that is pretty damn smart. This book is about how to build things like automated unit tests and functional simulators, which professionals in the embedded systems space hardly ever do because they feel there isn't enough time or there aren't enough resources in their programming environment or because there's never been hardware like theirs on the planet before. A lot of developers think it's unwise to write extensive code before the hardware is working, or they assume that their code can't be repurposed for a completely different hardware platform without massive rework. But that is simply not the case.

In this book, I'll show you how to apply some software engineering principles and best practices—or what I call patterns—to develop software in an efficient, sustainable manner. By applying these patterns in a deliberate way, you can develop software and firmware for embedded systems faster and with higher quality than ever before. To be clear, these patterns are not silver bullets. If, for example, your hardware platform requires you to "bit pack eight Boolean flags into a single byte," then these practices might be of limited use. Nevertheless, by implementing patterns, I think you'll find that the sum of the parts is greater than the whole. That is, the right effort applied in the right place can produce benefits far beyond what you might think.

In my experience, traditional embedded software projects tend to be monolithic applications that are optimized for their target hardware platforms. And this is understandable. Embedded projects have unique hardware characteristics, constrained resources (limited RAM, tiny amounts of Flash, no operating system support, etc.) and oftentimes require demanding real-time performance. On top of this, there are nearly always aggressive schedules and high expectations for the quality of the software. Consequently, the pressure to just get started, and to just meet the stated requirements at hand, is immense and only intensifies when, mid-project, software requirements change, hardware components become unavailable or go obsolete, and the time-to-market window gets shortened.

J. T. Taylor and W. T. Taylor, *Patterns in the Machine*,
https://doi.org/10.1007/978-1-4842-6440-9_1

But referring to "traditional" embedded software projects may be the wrong word to use. Embedded software isn't developed the way it is because of tradition; rather, it is often developed this way out of a sense of desperation. As one manager I worked with put it: the process is like "building a railroad bridge over a gorge in front of a moving train while the bridge is burning down behind it." This rush to get things done, then, leads to software that is fragile and that tends to collapse if there are requirement changes or feature extensions. It also leads to software that is challenging to test, especially before fully functioning hardware and fully integrated software are available. But by following the patterns in this book, these patterns in the machine (PIM), if you will, you can create software or firmware that actually embraces change and maximizes testability. PIM does not lament the fact that change is a constant; rather, it embraces it and focuses on highly decoupled designs that can accommodate changes without sacrificing quality.

A NOTE ABOUT TEACHING PIM

If I were to teach a class on *Patterns in the Machine*, the syllabus would look something like this:

- Week 1—Hand out a board and supply the class with requirements for an application to be built on it. Tell them that a working application will be required at the end of week 5.
- Weeks 2–5—Lecture and demonstrate how to design and develop with a PIM methodology.
- Week 6—Hand out a different board and add some new requirements and change some existing requirements. A modified version of the application they just completed will be required at the end of week 8.
- Weeks 7–8—Lecture and demonstrate how to adapt the first application to the new hardware and requirements.
- Week 9—Hand out a new board, add some new requirements, and change the requirements one last time. A working application will be required to be submitted at the end of week 10 as the final exam.

It should be obvious to most of the students that unless they design their software with an eye toward accommodating the changes that will come later in the semester, they will not be successful in week 10. Unfortunately, in the "real world," project managers and development managers don't tell the team "Six weeks from now the hardware will change, and we'll add some new requirements." More often than not, they say the very opposite: "The hardware and requirements are frozen. We promise." But almost without fail, the changes come. More than anything else, it was this fact of life that led me to develop and implement the principles of PIM. It was the only way I could survive.

As an exercise, then, ask yourself this about your current project: "If in a few weeks I were to get new hardware and new software requirements, but my original deadline does not change, could my current design and implementation allow me to proceed in a reasonable, sustainable manner? Or would I be frantically working overtime to refactor my code?"

Patterns in the Machine

PIM is an amalgamation of design methodologies, best practices, software architectures, and continuous integration principles which, when applied to the embedded development space, deliver projects faster and with higher quality. As an example of faster, consider that

- PIM allows developers to start writing and testing actual, meaningful production code without hardware.
- PIM allows you to start testing early and often. Finding bugs at the beginning of the development cycle—especially bugs related to design flaws—greatly reduces the overall development time.
- PIM yields reusable code, which means there is less code to write on subsequent projects.

As an example of higher quality, consider that

- PIM emphasizes unit tests that inherently make modules more testable. One of the outcomes of this testing focus is that PIM achieves many of the quality benefits of Test-driven development (TDD). And while PIM does not embrace all TDD practices, PIM is fully compatible with it.

- PIM facilitates the ability to create a functional simulator that allows for all phases of testing to start early (i.e., before the hardware is available). Obviously, this yields greater test time, but it also enables downstream tasks like developing user documentation and training materials to start much earlier.
- PIM provides for true reuse. That is, PIM allows you to reuse source code files without modification or cloning, so there is no loss of quality or functionality in reused modules.

Other benefits to consider are

- PIM has an extendable code base. That is, accommodating new features and requirements is easier because of the inherent decoupling of the code from hardware.
- PIM allows many developers to work efficiently on the same application because the decoupled code base translates into developers not competing for access to the same files in the software configuration management (SCM) system.
- PIM is portable; when properly architected, over 90% of the source code is compiler and hardware independent.
- PIM is an agnostic development process. That is, it can be used in Agile, TDD, waterfall, and so on.

What Is Software Engineering?

Whereas there are no readily agreed-upon canonical definitions of what software engineering is, here are some interesting definitions:

> *[Software engineering is] the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.*
>
> —IEEE Standard Glossary of Software Engineering Terminology, IEEE std 610.12-1990, 1990.

> *[Software engineering is] the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines.*
>
> —Fritz Bauer. "Software Engineering." Information Processing. 71: 530–538.

> *Software engineering should be known as "The Doomed Discipline," doomed because it cannot even approach its goal since its goal is self-contradictory. Software engineering, of course, presents itself as another worthy cause, but that is eyewash: if you carefully read its literature and analyse what its devotees actually do, you will discover that software engineering has accepted as its charter "How to program if you cannot."*
>
> —Edsger W. Dijkstra.
> www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html

To put it simply: PIM requires you to do software engineering. And for the purposes of this book, the IEEE definition of software engineering will suffice. Unfortunately, in my experience, software engineering best practices require a level of discipline from developers (and principal stakeholders) that is, more often than not, sacrificed for the sake of tactical concerns.

Software Engineering Best Practices

Software engineering best practices can be broken down into two categories:

- Tactical—Designing and constructing individual components or modules
- Strategic—Specifying how individual components work together, how they can be tested, and how they can be architected in a way that accommodates changes in requirements or the addition of new features

In my experience, tactical best practices are routinely incorporated into projects. Strategic best practices, however, are typically a function of the tribal knowledge of an organization. As a result, they vary widely between groups and departments. Additionally, the strategic best practices that do exist are usually narrowly focused to

meet past needs or present concerns. This differentiation between tactical and strategic is important because without a disciplined approach and commitment to strategic best practices, these are the first things dropped when "crunch time" arrives. While this may seem logical or even expedient, it is a net negative to the project's overall timeline and quality because it's the strategic best practices that maintain the "big picture" and integrity of the software. While tactical missteps typically have immediate consequences, strategic missteps typically aren't manifested until late in the project life cycle when they are expensive (in terms of time and effort) to fix. And, in many cases, the problems are never fixed as development teams often elect to take on "technical debt" by patching things together harum-scarum just to get the software out the door.

Here are some examples of tactical software engineering best practices:

- Design patterns
- Encapsulation
- Structured programming
- Object-oriented programming
- File organization
- Naming conventions
- Dependency management

Here are some examples of strategic software engineering best practices:

- Design patterns
- File organization
- Naming conventions
- Dependency management
- Automated unit testing
- Software architecture

Note that the two lists overlap. The reason is because most aspects of software development have both tactical and strategic characteristics. For example, let's examine naming conventions. These conventions are usually defined in the project's coding standards document. Typically, these conventions address things like case, underscores,

nouns, verbs, Hungarian notation, scope of variables, and so on—all of which can be considered tactical. However, an example of a strategic naming convention would be specifying a requirement that the use of C++ namespaces (or name prefixing in C) be incorporated to prevent future naming collisions.

Another example would be requiring the use of specific design patterns. For example, applying the "observer pattern" to a module in isolation that implements a callback would be considered tactical. However, it would be considered strategic to require that the same observer pattern be applied consistently across the entire data model so that change notifications are always generated for any changes anywhere.

Chapter 2 expands on these core concepts for PIM and explains the tactical and strategic considerations for each concept.

What PIM Is Not

Patterns in the Machine is not an introduction to, nor a beginner's guide for, embedded software development. In fact, it covers very few details about tactical topics for embedded development. This book is about how to use some basic planning, architecture, and design to build highly decoupled embedded applications and then how to exploit that design and implementation to get your project done faster and with higher quality.

While the following list is not comprehensive, here are some topics that will *not* be covered in this book:

- Multi-threading programming
- Real-time scheduling
- Differences between an MCU and a CPU
- How to work with hardware peripherals (ADC, SPI, I2C, UART, timers, input capture, etc.)
- Hardware design
- Reading schematics
- Interrupt handling
- Math (floating point vs. integer vs. hexpoint, etc.)
- Low-power designs

- Cross compilers
- Optimizing for space and real-time performance
- Safety-critical applications
- IoT applications
- Watchdogs
- Networking

What You'll Need to Know

The target audience for PIM are developers who have worked on at least one embedded project and have experience with either C or C++. For example, this may be

- Software developers or firmware developers that have mid-level or higher experience.
- Technical leads
- Software architects
- Development managers

Additionally, it will be helpful if you can read and follow code written in C and C++. While this is not a strict requirement, all the sample code that is provided with this book is written in C and C++. While in many instances I do provide detailed explanations of the algorithms, sometimes it is just more effective to provide a snippet of code.

CHAPTER 2

Core Concepts

This chapter introduces the core concepts of PIM and explains why they matter. For each concept discussed here, there is a corresponding chapter in the book that provides a more detailed discussion of the material.

Software Architecture

Just like the term software engineering, the terms "software architecture" and "software detailed design" do not have concise definitions. On many embedded project teams, there is no distinction—or at least not one that the developers can articulate—between the two. The tendency, then, is to define architecture and detailed design together. This works up to a point, but teams tend to focus on the detailed design, and the architecture essentially becomes the output of that detailed design. This leads to an architecture that is rigid in terms of dependencies and oftentimes inconsistent with itself.

The problem with code designed without an architecture document arises when you try to add new features that don't quite match up with the original detailed design or when you encounter a scenario where you're trying to shoehorn a "missed feature" into the design. For example, I worked on one project where the team designed the HTTP request engine to use synchronous and asynchronous inter-thread communication (ITC) to send requests to the external cell modem driver. Later in the project, we added a watchdog sub-system that would monitor the system for locked up threads, but we found that the watchdog would intermittently trip on the thread running the HTTP engine. The root cause turned out to be that, given a specific set of preconditions related to cellular network failures, the synchronous ITC calls from the HTTP request engine would block for minutes at a time. Nothing in the original design proscribed when synchronous ITC could (or could not) be used. Because we did not have a written software architecture, there was nothing to guide or constrain the design of this feature. The developer of the HTTP engine just threw something together that reflected his minimal understanding of cell modem behavior. Ultimately, we had to leave the watchdog sub-system out of the final product.

J. T. Taylor and W. T. Taylor, *Patterns in the Machine*,
https://doi.org/10.1007/978-1-4842-6440-9_2

You always want to have a detail-agnostic software architecture that the detailed design must conform to. It's the difference between driving a car on a paved road with guard rails and driving through an open field. Yes, the paved road has constraints on what and when and how vehicles and people can travel on it, whereas the open field has none; but getting from point A to point B is a lot faster and safer on the paved road as opposed to crossing an unbounded open field where nothing prevents you from colliding with other vehicles or local wildlife.

Software architecture best practices are strategic in nature. Define your project's software architecture first. Keep it separate from the software detailed design. There is an implied waterfall process here, but it's a good thing. Organically derived software architecture is the path to the dark side; or, without the moral overtones, it is often a quick path to "bit rot." Up-front architecture—separated from design—allows for just-in-time design, which is what you want in a development process like Agile. For example, if your software architecture defines the interface between the core business logic and the user interface as model points, then any work you do on the UI stories is completely decoupled from the business logic stories and vice versa. Only the model point instances need to be defined up front. (A more detailed discussion of model points is provided in Chapter 9.)

Automated Unit Testing

Unit tests are your friends; automated unit tests are your BFFs. Why? Because unit tests are an effective and repeatable way for developers to demonstrate that their code actually works. Manual testing may seem quicker in the moment because there is no test code to write, but it is rarely repeatable. This may not seem like a big deal until you have to make a change or have to fix a bug that requires regression testing. Additionally, without unit tests, it can be difficult to quantify actual test coverage.

In my experience, the time spent writing unit tests has always been net positive over the entire development cycle. Automated unit tests are even better because the execution of the tests can be incorporated into the project's continuous integration effort, yielding continual regression testing with code coverage metrics.

Unfortunately, writing unit tests—and especially automated unit tests—is not ingrained in the culture of embedded system development. I have no definitive explanation as to why this is, only empirical evidence that unit testing is not mainstream in the embedded world. My hypothesis is that because embedded development is

tightly coupled to hardware and, consequently, bleeding-edge development, test frameworks are not readily available on many target hardware platforms. As a result, it is easy to rationalize that writing unit tests is not practical. Nevertheless, in my experience, there are no technical constraints that prevent automated unit testing from becoming the norm for embedded development. PIM's approach to unit testing is a subset of Test-driven development (TDD) in that it only requires three things:

- That you build a unit test for each module
- That you test sooner rather than later
- That you build your tests incrementally

There are two principal ways to perform automated unit tests for embedded systems. The first is to have an automated platform that can simulate the system's environment and interact with the software while it is running on its target hardware. There are many advantages to this approach, but it is costly in terms of resources, money, and time. In many ways, developing this test platform is an entire software project of its own. The second approach is to have the automated unit tests run as terminal (or console) applications on a computer. These tests return pass/fail. The obvious advantage here is that there is no simulation infrastructure to build, and there are many tools available to assist and augment the automated unit tests. The disadvantage to this approach is that it requires that the software be developed in a way that allows it to be executed both with the test computer's operating system and with the target hardware.

The PIM approach to automated unit testing is to decouple the software under development from the platform (i.e., the hardware, the OS, the compiler, etc.) so that computer-based automated testing is practical. While not all software can be abstracted away from the platform, in my experience, over 90% of an embedded application can be decoupled from the target platform with minimal extra effort. Whether it is a project on an 8-bit microcontroller or a CPU running a process-based operating system, after the source code is decoupled from the target platform and compiler, there is no downside to having computer-based automated unit testing. Of course, decoupling the software from the target platform can be tricky. But in most cases, with some up-front planning—and the discipline to follow the plan—it is a straightforward process. Furthermore, decoupling the software from the target platform also creates other benefits like being able to create a functional simulator.

To summarize, then, requiring unit tests and automated unit tests is a strategic best practice. The construction of the unit tests and test frameworks are the tactical best practices.

Functional Simulator

Just like changing requirements are a fact of life when developing software applications, "hardware is always late" is a truism for embedded projects. I have worked on numerous projects where software development begins before any hardware engineers or resources were assigned to the project, so, by definition, the hardware was already late. This creates the challenge of trying to write and test production-quality code without target hardware and without incurring a large amount of technical debt. This is where the advantages of having a functional simulator come in.

The goal of a functional simulator is to execute the production source code on a platform that is not the target platform. The simulator should provide the majority of the functionality (but not necessarily the real-time performance) of the application. In most cases, this hardware platform is a personal computer running Windows or Linux.

I first started incorporating a functional simulator in an embedded project 20 years ago as a direct result of target hardware not being available. And even after the hardware became available, the functional simulator was still used as the principal development platform. In fact, the only developer testing done on the actual target hardware were hardware-specific tests of real-time features. This was due to the fact that developing, and then executing, code on a PC was simply easier and Faster than on the target hardware where you had to cross-compile, program the Flash in the target microcontroller, and then debug the result on the target hardware. While the tools available for many target hardware platforms have improved greatly over the last two decades, developing code on a functional simulator is still easier and faster than using actual hardware.

While including a functional simulator in the project development cycle does require additional effort and planning, the complexity of that effort will vary by the nature of the project and target platforms. By starting with a minimal simulator and only then extending its capabilities on a case-by-case basis, the extra effort is minimized. The point to be emphasized here is that the effort to create a minimal functional simulator is close to free because the design work and planning that go into creating automated unit tests are 80% of the effort that is required to build the simulator.

When I first started building functional simulators, I had to convince management that constructing a functional simulator would be a net positive effort for the project. Today, I just pitch the concept of automated unit testing to management which is an easy sell. And lately I don't even have to pitch anything because management has

already bought into automated unit testing. But after automated unit testing is part of the development process, constructing a minimal functional simulator becomes an uncontested line item in the schedule because the effort is small enough to be "lost in the noise."

The decision to include a functional simulator on a project is a strategic best practice. The use of the functional simulator as a substitute for the target hardware platform is a tactical best practice.

Continuous Integration

In PIM, the decision to include continuous integration (CI) on a project is a strategic best practice. As the concept of CI has been around since the 1990s, many of you have already accomplished the tactical objective of creating an automated build system for your embedded projects. Nevertheless, it is still important to articulate and define the strategic rules that govern the creation and ongoing maintenance of the CI for your project.

Martin Fowler provides this succinct definition of CI:

> *Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.*
>
> —Martin Fowler (1 May 2006). "Continuous Integration." martinfowler.com. Retrieved 9 January 2014.

In the context of PIM, that yields the following strategic objectives:

- Build all work that is checked into the software configuration management (SCM) system. This should be done *before*—that is, separate and apart from—the work of merging all of the checked-in work to a mainline or otherwise stable branch of the code.
- Use the same build server for compiling daily check-ins as well as for creating formal builds from stable branches in the SCM repository.

- Execute the automated unit testing from the build server, and have the build fail if one or more automated unit tests fail.
- Build everything all the time

Setting up a CI process is nontrivial. Make sure that you include stories or tasks in your schedule to get the build server and CI process up and running. Also, periodic maintenance and support for the CI process should be included in your schedule as well.

So, how costly is it to add CI to a project? The short answer is: it depends. Here are some considerations that can complicate the CI process:

- Your experience with automation tools—There are numerous commercial and open source tools for automating builds and executing unit tests. These tools require a certain level of expertise to properly configure and use them.
- Your SCM tools—There are two primary issues here:
    1) Defining a branching or workflow strategy that explicitly incorporates CI. There needs to be steps in the workflow that prevent merging source code changes to stable branches until the CI server has successfully built and verified the changes.
    2) Defining the source code and repository organization such that it integrates with the automation tools (e.g., job construction in Jenkins is simpler when there is only one SCM repository involved as opposed to many).
- Your host build tools and environments—All of the tools used to build an embedded project must be installed on the build server (or a slave server). This also includes having compiler and tool licenses for the build server or build servers. Having build tools that execute on different operating systems further complicates the build server configuration and job construction.
- Your build engine or make files—The project's build process needs to support building the released application as well as building the automated unit tests. Depending on a project's constraints and requirements, the time and effort to define and implement this can vary greatly.

- The maintenance of the build server—Whether your build server is a physical machine or a virtual machine, you need to follow IT best practices in maintaining and backing up the platforms. The automation tools themselves will need a certain amount of maintenance as you add or update existing automated jobs.
- The build times—The amount of code that is built, and the number of unit tests that are executed, increases over time. In a perfect world, the build and test cycle for CI would be seconds. In my experience, however, for embedded projects, the reality is that the build times are minutes to hours. A general rule of thumb for build server hardware is that you can never have too much disk space, too much RAM, or too many cores because, inevitably, reducing CI build times becomes an issue.

As you can see, CI is not a simple or free addition to a project. So why do it? Going back to Martin Fowler's definition of CI, the reason to do it is to detect integration errors as quickly as possible. On the surface, this may not sound like a huge win, but CI is a significant net positive when it comes to maintaining stable branches in your SCM. For me, the best argument for detecting integration errors as quickly as possible is to avoid the pain I experienced living through broken builds that were required to be stable at all times and getting bogged down in "merge hell." I have worked with really big companies that dreaded pulling formal releases together because the build process was anything but integrated, and it had to account for a myriad of dependencies on other projects. In these environments, it could often take a week to release the project, and, in the end, I had the nagging sense that what we finally pulled together was very fragile.

This brings us to a final strategic objective for CI: build everything all the time. It is not uncommon for a project to be defined something like this: (1) release the project on ABC hardware; (2) a month later, release the software on XYZ hardware; (3) a month later, release localized versions of the software on both hardware platforms. In cases like this, it is important not to postpone the building of the XYZ project because "it hasn't really started" or to put off building the localized versions because "we haven't even sent the text off for translation." Rather, from the beginning, you should establish jobs in your CI process where all the artifacts for all the platforms and languages get built. Even if originally the jobs for the anticipated hardware start out as a lot of "stubbed" code and the localized versions only have one or two words translated, it is a net win to have these jobs building artifacts with every automated build.