J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_1

1. Embedded Software Design Philosophy

Jacob Beningo[1]
(1) Linden, MI, USA

The first microprocessor, the Intel 4004, was delivered to Busicom Corp in March 1971 for use in their 141-PF calculators.[1] The 4004 flaunted a 4-bit bus running at 0.74 MHz with up to 4 KB of program memory and 640 bytes of RAM.[2] The 4004 kicked off a hardware revolution that brought more computing power to the world at lower costs and led to faster, more accessible software development. In time, fully integrated processors designed for real-time control in embedded systems were developed, including the microcontroller.

The first microcontroller, the Texas Instruments TMS1000,[3] became commercially available in 1974. The TMS1000 boasted a 4-bit bus running at 0.3 MHz with 1 KB x 8 bits program memory and 64 x 4 bits of RAM. The critical difference between a microprocessor and a microcontroller is that a microcontroller integrates ROM, RAM, and combined input/output (I/O) onto a single chip, whereas microprocessors tend to be multichip solutions. Thus, microcontrollers are intended for embedded systems with a dedicated function and are not general computing devices.[4] Examples include automobile controllers, home appliances, intelligent sensors, and satellite subsystems.

Early in embedded system history, the industry was dominated by a hardware-centric mindset. A product's differentiator wasn't the software but whether the system was built on the latest hardware. After all, the first microcontrollers were heavily resource-constrained and had a limited feature set. The more advanced the hardware, the more features one could pack into the product. Early on, there was only 4 KB of ROM for

software developers to work with, which doesn't seem like a lot by today's standards, but remember, that was 4 KB of hand-coded assembly language!

The software was viewed as a necessary evil; it's still sometimes considered an essential evil today, even though the hardware is relatively agnostic. Microcontrollers come in a wide variety of flavors with bus architectures including 8-bit, 16-bit, and 32-bit. One might be tempted to believe that 32-bit microcontrollers are the dominant variants in the industry, but a quick look on Digikey or Mouser's website will reveal that 8-bit and 16-bit parts are still just as popular.[5] Microcontroller clock speed varies widely from 8 MHz all the way up to 268 MHz in Arm Cortex-M4 parts. Although there are now offerings exceeding 1 GHz. Finding parts with sporting 512 KB program memory and 284 KB RAM is fairly common. In fact, there are now multicore microcontrollers with 1000 MHz system clocks, over 1024 KB of program memory, and 784 KB of RAM!

A key point to realize in modern embedded system development is that the software is the differentiator and the secret sauce. Microcontrollers can supply nearly as much processor power and memory as we want! The software will make or break a product and, if not designed and implemented correctly, can drive a company out of business. The hardware, while important, has taken the back seat for many, but not all, products.

Successfully designing, building, and deploying production embedded systems is challenging. Developers today need to understand several programming languages and how to implement design patterns and communication interfaces and apply digital signal processing, machine learning, and security, just to name a few. In addition, many aspects need to be considered, ranging from the software and hardware technologies to employ all the way through ramifications for securely producing and performing firmware updates.

As a consultant, I've been blessed with the opportunity to work with several dozen engineering teams worldwide and across various industries for over a decade. I've noticed a common trend between successful groups and those that are not. The successful design, implementation, and

deployment of an embedded system depend on the design philosophy the developer or team employs for their system (and their ability to execute and adhere to it).

> **Definition***A design philosophy* is a practical set of ideas about how to[6] design a system.

Design philosophies will vary based on the industry, company size, company culture, and even the biases and opinions of individual developers. However, a design philosophy can be considered the guiding principles for designing a system. Design philosophies should be practical and improve the chances of successfully developing the product. They should not handcuff the development team or add a burden on the developers. I often see design principles as best practices to overcome the primary challenges facing a team. To develop your design philosophy, you must first examine the embedded system industry's challenges and then the challenges you are facing in your industry and at your company.

> **Best Practice**Many discussions in this book are generalizations based on industry best practices and practical industry experience. Don't take it as gospel! Instead, carefully evaluate any recommendations, tips, tricks, and best practices. If they work and provide value, use them! If not, discard them. They will not work in every situation.

Challenges Facing Embedded Developers

Every developer and team faces inherent challenges when designing and building their system. Likewise, every industry will have unique challenges, varying even within the same company between the embedded software developers and the hardware designers. For example, some teams may be faced with designing an intuitive and easy-to-use graphical user interface. Another may face tough security challenges that drive nearly every design aspect. However, no matter the team or developer, there are several tried and true challenges that almost all teams face that contribute to the modern developers' design philosophy. These fundamental challenges include

- Cost (recurring and nonrecurring)
- Quality
- Time to market
- Scalability

- Etc.

The first three challenges are well known and taught to every project manager since at least the 1950s.[7] For example, Figure 1-1 shows the project management "iron triangle,"[8] which shows that teams balance quality, time, and cost for a given scope in every project.

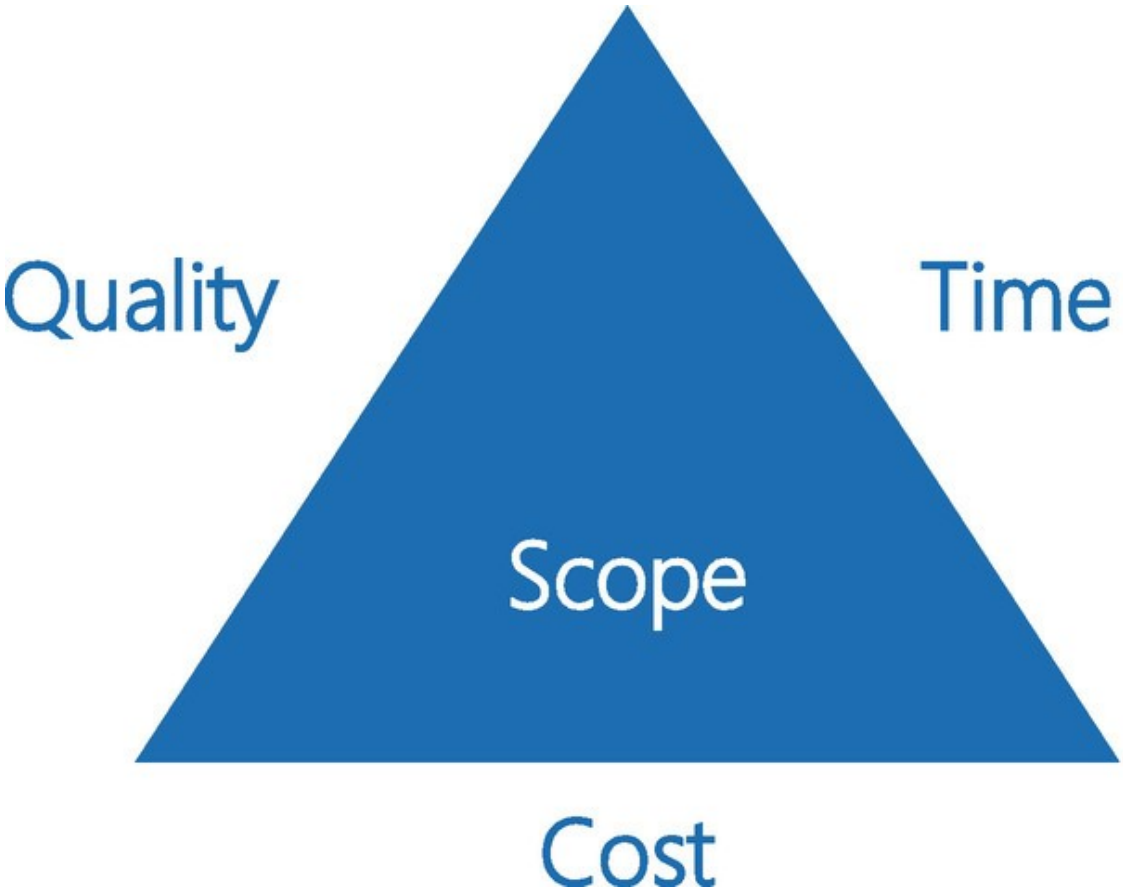

Figure 1-1 The project management iron triangle demonstrates that teams balance the demands of quality, delivery time, cost, and scope in any project

You may have heard project managers say that they can pick any two between quality, cost, and delivery time, and the third must suffer. However, what is often overlooked is that scope is just as important in the trade-off. If a project is going over budget or past the deadline, scope reduction can be used to bring them back into balance. So these properties are trading off against each other. Unfortunately, this model, in general, is NOT sufficient alone, but it at least visually provides teams with the primary business challenges that every team is grappling with. So, let's quickly look at some challenges and see how they can impact developers.

Every team that I encounter is designing their product under a limited

budget. I don't think I have ever met a team with an unlimited budget and was told to spend whatever it takes to get the job done. (If you work for one of those rare companies, my contact information is `jacob@beningo.com`.) For the most part, businesses are always trying to do more with smaller budgets. This makes sense since a business's core goal isn't to change the world (although they will tell you it is) but to maximize profits! Therefore, the smaller the development budget and the more efficient the engineers are, the higher the potential profit margins.

Many teams face problems with development costs because today's systems are not simple. Instead, they are complex, Internet-connected feature beasts with everything plus the kitchen sink thrown in. As feature sets increase, complexity increases, which requires more money up front to build out the development processes to ensure that quality and time-to-market needs are carefully balanced. The more complex the system becomes, the greater the chances that there will be defects in the system and integration issues.

Budgets can also affect software packages and tools, such as network stacks, compilers, middleware, flash tools, etc. Because it's "free," the big push for open source software has been a major, if not sometimes flawed, theme among teams for the last decade or more (we will discuss this in more detail later in the book). I often see teams unwilling to spend money on proper tools, all in an attempt to "save" money. Usually, it's a trade-off between budget and extra development time.

Product quality, in my mind, is perhaps the most overlooked challenge facing development teams. Product quality is simply whether the product does what it is supposed to when it is supposed to and on time. Unfortunately, today's systems are in a constant bug fix cycle. The real challenge, I believe, is building a product that has the right amount of quality to meet customer expectations and minimize complaints and hassle on their part. (It always drives me crazy when I need to do a hard reset on my smart TV.)

The time to market for a product can make or break a company. There is a careful dance that needs to be performed between delivering the product and

marketing the product. Market too soon, and the company will look foolish to its clients. Market too late, and the company may not generate sufficient cash flow to maintain its business. Delivering a product on time can be a tricky business. The project scope is often far more extensive and more ambitious than is reasonable for time and budget and can often result in

- Missed deadlines
- More debugging
- Integration woes

Lesson Learned Successful and experienced teams are not immune to delivering late, going over budget, and scope creep! Therefore, one must always be diligent in carefully managing their projects.

The first three challenges we have discussed have been known since antiquity. They are project management 101 concepts. Project managers are taught to balance these three challenges because they are often at odds with each other. For example, if I am in a delivery time crunch and have the budget and a project that can be parallelized, I may trade off the budget to get more developers to deliver the project quicker. In *The Mythical Man Month*, Fred Brooks clearly states that “adding people to a late project makes it later.”

Teams face many other challenges, such as developing a secure solution, meeting safety-critical development processes, and so forth. The last challenge that I believe is critical for teams, in general, is the ability to scale their solutions. Launching a product with a core feature set that forms the foundation for a product is often wise. The core feature set of the product can be considered a minimum viable product (MVP). The MVP helps get a product to market on a reasonable budget and timeline. The team can then scale the product by adding new features over time. The system needs to be designed to scale in the field quickly.

Scalability is not just a need for a product in the field; it can be a need for an entire product line. Systems today often have similar features or capabilities. The ability to have a core foundation with new features added to create a wide variety of products is necessary. It’s not uncommon for a single code base to act as the core for a family of several dozen products, all of which need to be maintained and grown over a decade. Managing product lines over time can be a dramatic

challenge for a development team to face, and a few challenges that are encountered include

- Tightly coupled code
- Vendor dependency
- Inflexible architecture

We should also not forget that scalability is essential because developers often work on underscoped projects! In addition, requirements change as the project is developed. Therefore, the scalable and flexible software architecture allows the developer to minimize rework when new stakeholders or requirements are passed down.

The challenges we have discussed are ones we all face, whether working for a Fortune 500 company or just building an experimental project in our garages. In general, developers start their projects a day late and a dollar short, which is a significant reason why so many products are buggy! Teams just don't have the time and money to do the job right,[9] and they feel they need to cut corners to try to meet the challenges they are facing, which only makes things worse!

A clearly defined design philosophy can help guide developers on how they design and build their systems, which can help alleviate some of the pain from these challenges. Let's now discuss the design philosophy I employ when designing software for my clients and projects that you can easily adopt or adapt for your own purposes.

7 Modern Design Philosophy Principles

As we discussed earlier, a design philosophy is a set of best practices meant to overcome the primary challenges a team faces. Of course, every team will have slightly different challenges and, therefore, a somewhat different set of design philosophies they follow. What is crucial is that you write down what your design principles are. If you write them down and keep them in front of you, you'll be more likely to follow them and less likely to abandon them when the going gets tough.

I believe every team should follow seven modern design philosophy principles to maximize their chances for success. These include

- Principle #1 – Data Dictates Design
- Principle #2 – There Is No Hardware (Only Data)
- Principle #3 – KISS the Software
- Principle #4 – Practical, Not Perfect
- Principle #5 – Scalable and Configurable
- Principle #6 – Design Quality in from the Start
- Principle #7 – Security Is King

Let's examine each principle in detail.

Best Practice Create a list of your design principles, print them out, and keep them in a visible location. Then, as you design your software, ensure that you adhere to your principles!

Principle #1 – Data Dictates Design

The foundation of every embedded system and every design is data. Think carefully about that statement for a moment. As developers, we are often taught to focus on creating objects, defining attributes, and being feature-focused. As a result, we often get hung up on use cases, processes, and deadlines, quickly causing us to lose sight that we're designing systems that generate, transfer, process, store, and act upon data.

Embedded systems are made up of a wealth of data. Analog and digital sensors are constantly producing a data stream. That data may be transferred, filtered, stored, or used to actuate a system output. Communication interfaces may be used to generate system telemetry or accept commands that change the state or operational behavior of the system. In addition, the communication interfaces may themselves provide incoming and outgoing data that feed a primary processing system.

The core idea behind this principle is that embedded software design is all about the data. Anything else focused on will lead to bloated code, an inefficient design, and the opportunity to go over budget and deliver late. So instead, developers can create an efficient design by carefully identify-

ing data assets in the system, the size of the data, and the frequency of the operations performed with the data.

When we focus on the data, it allows us to generate a data flow diagram that shows

- Where data is produced
- Where data is transferred and moved around the application
- Where data is stored (i.e., RAM, Flash, EEPROM, etc.)
- Who consumes/receives the data
- Where data is processed

Focusing on the data in this way allows us to observe what we have in the system and how it moves through the system from input to output, enabling us to break down our application into system tasks. A tool that can be very useful with this principle is to develop a data flow diagram similar to the one shown in Figure 1-2.

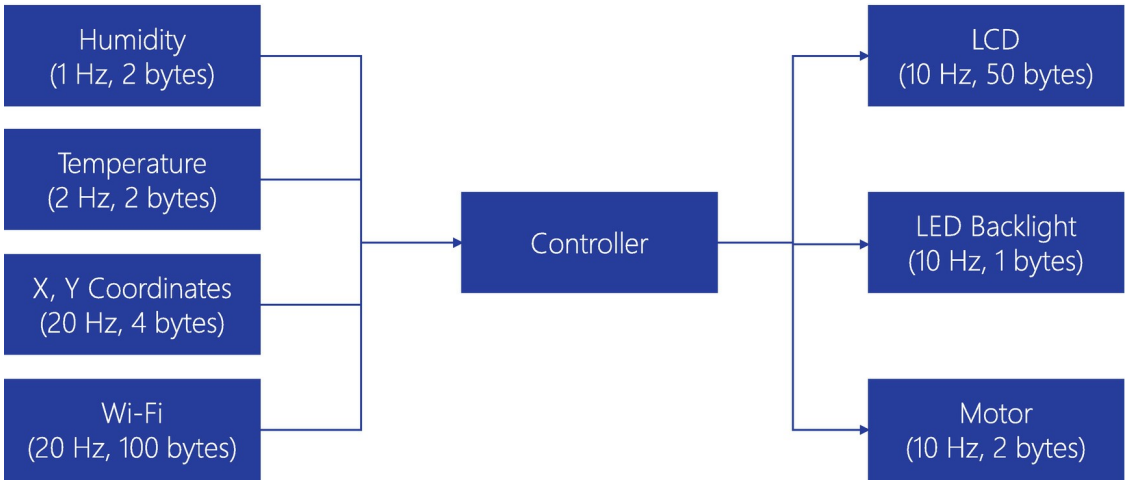

Figure 1-2 A simple data flow diagram can help developers identify the data assets in their system and how they move through the application

Principle #2 – There Is No Hardware (Only Data)

There is no hardware is a challenging principle for embedded developers to swallow. Most embedded software developers want to run out and buy a development board as quickly as possible and start writing low-level drivers ASAP. Our minds are engrained that embedded software is all about the hardware, and without the hardware, we are downriver without a paddle. This couldn't be further from the truth and puts the cart be-

fore the horse, leading to disaster.

I often see teams buying development boards before creating their design! I can't tell you how many times I've sat in on a project kickoff meeting where teams on day one try to identify and purchase a development board so they can get started writing code right away. How can you purchase and start working on hardware until the design details have been thought through and clear requirements for the hardware have been established?

Embedded software engineers revel in writing code that interacts with hardware! Our lives are where the application meets the hardware and where the bits and bytes meet the real world. However, as we saw in principle #1, the data dictates the design, not the hardware. Therefore, the microcontroller one uses should not dramatically impact how the software is designed. It may influence how data is generated and transferred or how the design is partitioned, but not how the software system is designed.

Everything about our design and how our system behaves is data driven. Developers must accept that there is no hardware, only data, and build successful business logic for the software! Therefore, your design at its core must be hardware agnostic and only rely on your data assets! A general abstraction for this idea can be seen in Figure 1-3.

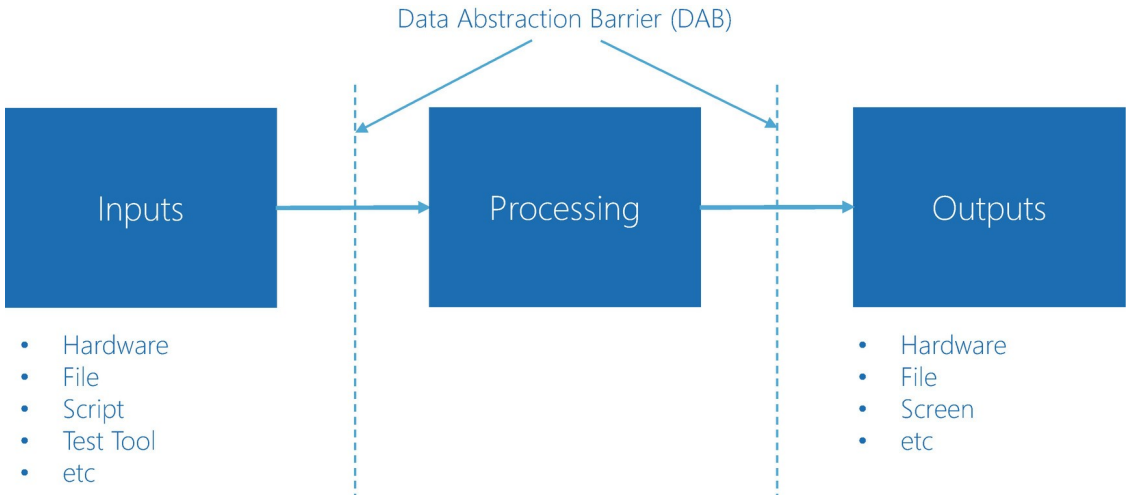

Figure 1-3 Application design should view hardware as agnostic and focus on the system data's inputs, processes, and outputs

This idea, at first, can be like Neo from *The Matrix*, having to accept that there is no spoon! For us developers, there is no hardware! Only data! Data acts as an input; it's then processed and used to produce system outputs (remember, it might also be stored). Before you do anything drastic with that pitchfork in your hand, let me briefly explain the benefits of thinking about design in this way.

First, a data-centric and hardware-agnostic software system will naturally abstract itself to be more portable. It is not designed for a specific hardware platform, making the code more "mobile." Second, the software will more readily integrate with test harnesses because the interfaces are designed to not be hardware dependent! This makes it much easier to connect them to a test harness. Third, the abstractions make it much easier to simulate the application! Developers can run the application on a PC, target hardware, or any convenient computing machine even if the hardware designers haven't yet produced prototypes. (And let's be honest, even if you have a prototype, you can't always trust that it will work as intended.) Fourth, suppose we decouple the application from hardware. In that case, we can rapidly develop our application by removing tedious compile, program, and debug cycles by doing all the debugging in a fast and hosted (PC) environment! I'm just warming up on the benefits, but I think you get the point.

Look again at Figure 1-3. You can see how this principle comes into practice. We design tasks, classes, modules, and so forth to decouple from the hardware. The hardware in the system is either producing data, processing that data, or using the data to produce an output. We create data abstraction barriers (DABs), which can be abstractions, APIs, or other constructs that allow us to operate on the data without direct access to the data. A DAB allows us to decouple the hardware and then open up the ability to use generators, simulators, test harnesses, and other tools that can dramatically improve our design and how we build systems.

Principle #3 – KISS the Software

Keep It Simple and Smart (KISS)! There is a general tendency toward so-

phisticated and overly complex software systems. The problem is that developers try to be too clever to demonstrate their expertise. As everyone tries to show off, complexity increases and code readability decreases. The result is a design and system that is hard to understand, maintain, and even get working right.

The KISS software principle reminds us that we should be trying to design and write the simplest and least complex software possible. Minimizing complexity has several advantages, such as

- More accessible for new developers to get up to speed
- Decrease in bugs and time spent debugging
- More maintainable system(s)

It's easy to see that the more complex a design is, the harder it will be to translate that design vision into code that performs as designed.

I often have told my clients and attendees at my workshops that they should be designing and writing their software so that an entry-level engineer can understand and implement it. This can mean avoiding design patterns that are difficult to understand, breaking the system up into smaller and more manageable pieces, or even using a smaller subset of a programming language that is well understood.

Keep the software design simple and smart! Avoid being clever and make the design as clear as possible. Brian W. Kernighan stated, "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

DefinitionA design pattern is a reusable solution to a commonly occurring problem in software design.[10]

Principle #4 – Practical, Not Perfect

There are many reasons why developers are writing embedded software. Still, the primary purpose for many reading this book is professional engi-

neers developing a product, an entrepreneur starting a business, or those perhaps interested in building a do-it-yourself (DIY) project and looking to improve their skills. In general, the purpose is to create a product that creates value for the user and generates profit for a business.

Developers and entrepreneurs often lose sight of the purpose of developing their software. The excitement or the overarching vision starts to cloud their judgment, and they begin to fall into perfectionism. Perfectionism is "a flawless state where everything is exactly right. But, it can also be the act of making something perfect."[11] There are two categories of perfectionism that we need to discuss:

- "One system to rule them all."
- "It still needs tweaking."

In "one system to rule them all," the team has become solely focused on creating a single product that can offer themselves and their users every possible feature imaginable. The product is a Swiss Army knife on steroids and, once completed, will not need to be modified or rewritten for centuries to come. Of course, it's perfectly acceptable to have a glorious product vision. Still, an excellent way to go out of business is to build the final vision first, rather than slowly working up to that final version. After all, Apple didn't wait until they had version X of their iPhone to launch their product; they found the minimal set of features that would be a viable product and launched that! Over time, they added features and tweaked hardware and software, evolving their product into what it is today.

Early in my career, I worked at a start-up that was creating revolutionary in-dashboard products. Unfortunately, the company never shipped a product in its five years of existence. Every time we had something that was close to shipping, the owner would change the requirements or have some new hardware feature that is a must-have before the product could be shipped. As I write this, it's nearly 15 years later, and the "must-have" features still cannot be found in similar products today!

The second perfectionist category is the "it still needs tweaking" group. This group of engineers is never quite happy with the product. For exam-

ple, sensor readings are within 1% and meet the specification, but we should be able to get within 0.5%. That code module is working but needs to be rewritten to refactor, add more documentation, etc. (Sometimes, that is a fair statement.) Yes, features A and B are working, but we should include C and D before we even consider launching. (And once C and D are complete, there's E and F, then G and H, then ….)

Best Practice The 80/20 rule states, "80% of software features can be developed in 20% of the time."[12] Focus on that 80% and cut out the rest!

It's helpful to mention right now that there is no such thing as a perfect system. If you were to start writing the ideal Hello World program right now, I could check in with you when I retire in 30 years, at age 70, and find that you almost have it, just a few more tweaks to go. Developers need to lose the baggage that everything needs to be perfect. Anyone judging or condemning that the code isn't perfect has a self-esteem issue, and I would argue that any developer trying to write perfect code also has one.

Systems need to be practical, functional, and capable of recovering from faults and errors. The goal isn't to burn a company's cash reserves to build a perfect system. It's to make a quality product with a minimal feature set that will provide customers with the value they need and generate a profit for the company. The faster this can be done, the better it is for everyone!

Principle #5 – Scalable and Configurable

Since our goal is not to build "one system to rule them all" or build the perfect system, we want to set a goal to at least make something scalable and configurable. We create a product starting with our minimum feature set, deploy it, and then continue to scale the system with new features and make it adaptable through configuration. Software needs to be scalable and configurable for a product to survive the rapidly evolving business conditions developers often find themselves in.

A scalable design can be easily adapted and grown. For example, as cus-

tomers use a product, they may request new features and capabilities for the software that will need to be added. Therefore, a good design will be scalable, so developers can quickly and easily add new features to the software without tearing it apart and rewriting large pieces from scratch.

A configurable design allows the developer and/or the customer to change the system's behavior. There are two types of configurations: build time and runtime. Build time configuration allows developers to configure the software to include or exclude software features. This can be used to customize the software for a specific customer, or it can be used when a common code base is used for multiple products. In addition, the configuration can be used to specify the product and the target hardware.

Runtime configurations are usually changeable by the end customer. The customer can change settings to adjust how the features work to best fit their end needs. These changes are often stored in nonvolatile memory like EEPROM. Therefore, changing a setting does not require that the software be recompiled and deployed.

Designing a scalable and configurable software system is critical for companies that want to maximize code reuse and leverage their software assets. Often, this principle requires a little bit of extra up-front work, but the time and money saved can be tremendous on the back end.

Principle #6 – Design Quality in from the Start

Traditionally, in my experience, embedded software developers don't have the greatest track record for developing quality systems. I've worked with over 50 teams globally so far in my career, and maybe five of them had adequate processes that allowed them to build quality into their product from the start. (My sampling could be biased toward teams that know they needed help to improve their development processes.) I think most teams perform spot checks of their software and simply cross their fingers that everything will work okay. (On occasion, I'm just as guilty as every other embedded developer, so this is not a judgmental statement, I'm just calling it how I see it.) This is no way to design and build modern

embedded software.

When we discuss designing quality into software, keep in mind at this point that quality is a "loaded" term. We will define it and discuss what we mean by quality later in the book. For now, consider that building quality into a system requires teams to

- Perform code reviews
- Develop a Design Failure Mode and Effects Analysis
- Have a consistent static analysis, code analytics process defined
- Consistently test their software (preferably in an automated fashion)

It's not uncommon for teams to consider testing to be critical to code quality. It certainly is important, but it's not the only component. Testing is probably not the foundation for quality, but it could be considered the roof. If you don't test, you'll get rained on, and everything will get ruined. A good testing process will ensure

- Software modules meet functional requirements.
- New features don't break old code.
- Developers understand their test coverage.
- Tests, including regression testing, are automated.

The design principles we discussed all support the idea that testing should be tightly integrated into the development process. Therefore, the design will ensure that the resultant architecture supports testing. Building the software will then provide interfaces where test cases can simultaneously be developed, as shown in Figure 1-4.

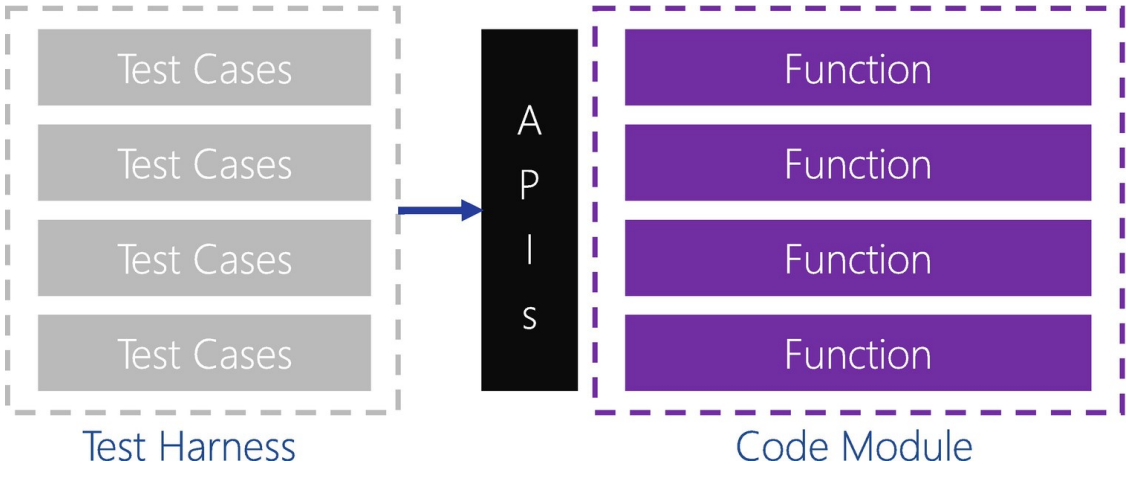

Figure 1-4 Embedded software can be designed to easily interface to a test harness that can automatically

execute test cases by interfacing to a module's APIs

Testing has several important purposes for developers. First, it ensures that we produce a higher-quality product by detecting defects. It doesn't guarantee quality, but can improve it. Second, testing can find bugs and software not behaving as intended. Next, testing can exercise every line and conditional statement in a function and every function in a module. Third, writing tests while developing software makes it more likely to approach 100% code coverage. Finally, testing can confirm that new code added to the system has not compromised software already in the system.

Principle #7 – Security Is King

The IoT has been taking over nearly every business and industry. Practically everything is in the process of being connected to the Internet: vehicles, home appliances, and even human beings. Even the simple mousetrap is now an Internet-connected device.[13] (Don't believe me, check out the footnote link!) All this connectivity leads to a crucial point and a principle that many developers can no longer ignore: security is king.

Many developers cannot design and build an embedded system without considering security and privacy concerns. For example, a device connected to the Internet needs security! It may not need to be Fort Knox, but it needs to protect company and user data and ward off common attacks so that the device does not become a pawn in nefarious activity.

Best Practice Security needs to be designed into the system, **NOT** bolted onto the end. Security must come first, then the rest of the design follows.

This brings me to an important point; you cannot add security to a system after being designed! A secure system requires that the security threats be identified up front to set security requirements that direct the software design. To perform that analysis, designers need to identify their data assets and how they may be attacked. This is an interesting point because, if you recall principle #1, data dictates design! To secure a system, we need to identify our data assets. Once we understand our data, then we can properly design a system by

- Performing a threat model security analysis (TMSA)
- Defining our objectives
- Architecting secure software
- Building the software
- Certifying the software

We will talk more about this process in later chapters.

If you design a completely disconnected system and security is not a concern, then principle #7 easily drops off the list. However, you can think up a new seventh principle that makes more sense for the system you are working on and add it.

Remember, these seven principles are designed to guide designers to overcome the common challenges that are being faced. These may not apply across the board, and I highly encourage you to give each some thought and develop your design principles to guide your development efforts. However, as we will see throughout the book, these principles can go a long way in creating a well-defined, modern embedded software design and development process.

ResourceYou can download a printable copy of the seven modern design principles at `https://bit.ly/3OufvfO`.

Harnessing the Design Yin-Yang

So far, we have discussed how critical it is for developers to understand the challenges they are trying to solve with their design and how those challenges drive the principles we use to design and build software. Success, though, doesn't just depend on a good design. Instead, the design is intertwined in a Yin-Yang relationship with the processes used to build the software.

This relationship between design and processes can be seen easily in Figure 1-5. We've already established that core challenges lead to design solutions. The design and challenges direct the processes, which in turn provide the likelihood of success. This relationship can also be traversed in reverse. We can identify what is

needed for success, which directs the processes we use, which can lead our design, telling us what challenges we are solving.

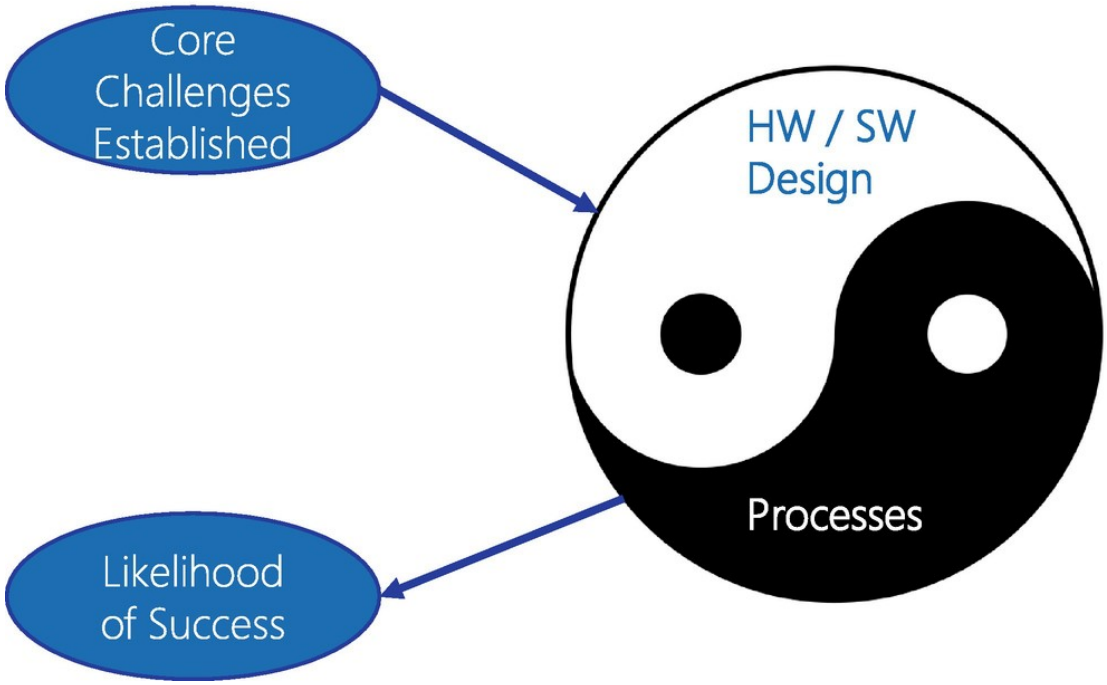

Figure 1-5 The software design and build processes balance each other to determine the project's likelihood of success and to establish the challenges that are solved by design

For example, if I am developing a proof-of-concept system, I will most likely constrain my challenge domain to only the immediate challenges I want to solve. My design will focus on that constrained domain, which will likely constrain the processes I use and dramatically improve my chances for success. On the other hand, in a proof of concept, I'm trying to go as fast as possible to prove if something is possible or not, so we end up using the minimal design and processes necessary to get there.

> **Tip**A proof of concept, a.k.a. a prototype, often morphs into a deliverable. That should never happen. Use fast tools that can't be embedded like Excel, Visual Basic, Python, etc., when possible.[14]

If, on the other hand, I want to develop a production system, my challenge domain will likely be much more significant. Therefore, my design will need to solve more challenges, resulting in more software processes. Those processes should then improve our chances of successfully producing a solution that solves our problem domain.

This brings me to an important point: all design and development cycles

will not be alike! Some teams can successfully deliver their products using minimal design and processes. This is because they have a smaller problem domain. For example, other teams, maybe ones working in the safety-critical system, have a much larger problem domain, resulting in more development processes to ensure success. The key is to identify your challenges correctly to balance your design and development processes to maximize the chances of success!

> **Best Practice**Periodically review design and development processes! Bloated processes will decrease the chances for success just as much as inadequate processes!

Traditional Embedded Software Development

Traditional embedded software development is a loaded term if I have ever heard one. It could mean anything depending on a developer's background and industry. However, for our purposes, traditional is the average development processes embedded teams have used over the last decade or two to develop software. This is based on what I saw in the industry when I first started to write embedded software in the late 1990s up until today.

Typically, the average software build process looked like Figure 1-6. The process involved hand-coding most software, starting with the low-level drivers and working up to the application code. The operating system and the middleware may have been open source, but teams often developed and maintained this code in-house. Teams generally liked to have complete control over every aspect of the software.

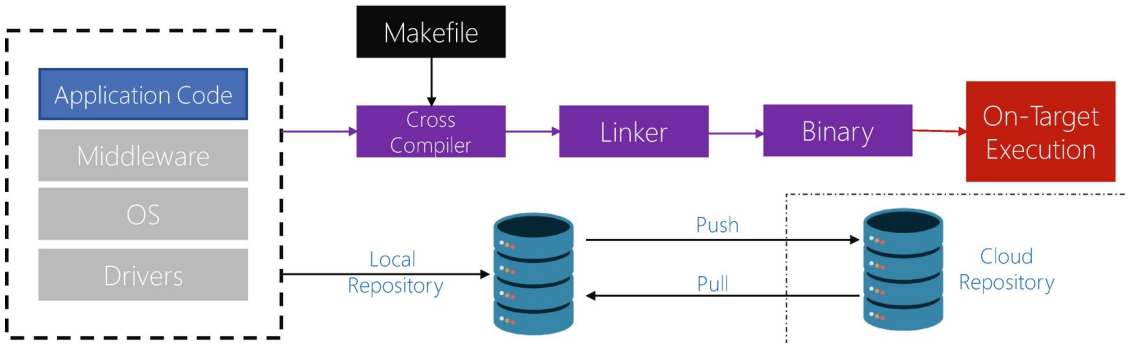

Figure 1-6 Traditional embedded software development leverages a hand-coded software stack that was only cross-compiled for on-target execution

Another critical point in the traditional embedded software process is that on-target development is heavily relied upon. From early in the development cycle, teams start development at the lowest firmware levels and then work their way up to the application code. As a result, the application code tends to be tightly coupled to the hardware, making it more difficult to port or reuse application modules. However, that is more an effect of the implementation. In general, the process has nothing wrong with it. The modern approach is more flexible and can dramatically decrease costs, improve quality, and improve the chance to meet deadlines.

Modern Embedded Software Development

Modern embedded software development can appear complicated, but it offers flexibility and many benefits over the traditional approach. Figure 1-7 provides a brief overview of the modern embedded software build system. There are several essential differences from the traditional approach that is worth discussing.

First, notice that we still have our traditional software stack of drivers, OS, middleware, and application code. The software stack is no longer constrained to only hand-coded modules developed in-house. Most software stacks, starting with low-level drivers through the middleware, can be automatically generated through a platform configuration tool. Platform configuration tools are often provided by the microcontroller vendor and provide development teams with an accelerated path to get them up and running on hardware faster.

While these configuration tools might first appear to violate the principle that there is no hardware only data, at some point we need to provide hooks into the hardware. We maintain our principle by leveraging the platform code behind abstraction layers, APIs. The abstractions allow our application to remain decoupled from the hardware and allow us to use mocks and other mechanisms to test and simulate our application code.

It's important to note that vendor-provided platform configuration tools are often designed for reuse and configuration, not for execution efficiency. They can be a bit more processor cycle hungry, but it's a trade-off between development

speed and code execution speed. (Don't forget principle #4! Practical, not perfect! We may do it better by hand, but is there a tangible, value-added benefit?)

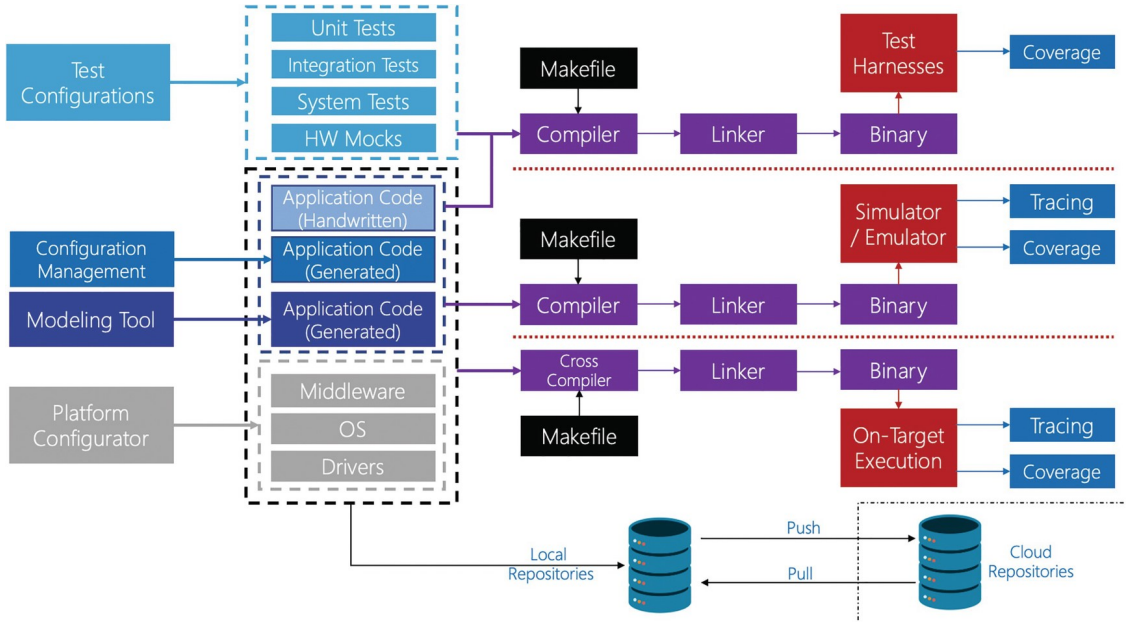

Figure 1-7 Modern embedded software applies a flexible approach that mixes modeling and configuration tools with hand-coded development. As a result, the code is easily tested and can be compiled for on-target or simulated execution

Next, the application code is no longer just hand-coded, there are several options to automate code generation through modeling and configuration management. This is a massive advantage because it allows the application business logic to be nearly completely fleshed out and tested in a simulated environment long before the product hardware is available to developers. Business logic can be developed, tested, and adjusted very quickly. The code can then be rapidly autogenerated and deployed to the target for on-target testing. Again, the code generated by a modeling tool is generally bad from a readability standpoint. However, if the modeling tool is used to maintain the business logic, there isn't any harm unless the generated code is grossly inefficient and doesn't meet the systems requirements.

Configuration management is a critical component that is often mismanaged by development teams. It is common to have a core product base that is modified and configured differently based on the mission or customer needs. Embedded software teams can explore general computer science best practices and use hosted configuration files like YAML, XML,

etc., to configure and generate flexible components within the application. We'll talk more about this later in the chapter on configuration management.

Another improvement over the traditional approach is that testing is now intricately built into the development process. Embedded software always required testing, but I found that the traditional method used manual test cases almost exclusively which is slow and error prone. Modern techniques leverage a test harness that allows automated unit and regression testing throughout development. This makes it much easier to continuously test the software rather than spot-checking here and there while crossing fingers that everything works as expected. We will be extensively discussing how to automate tests and leverage continuous integration and continuous deployment in the Agile, DevOps, and Processes part of the book.

Finally, perhaps most importantly, the software's output has three separate paths. The first is the traditional path where developers can deploy the compiled code to their target system to run tests and traces and understand test coverage. Developers can also perform on-target debugging if necessary. The second, and more exciting approach, is for developers to run their software in a simulator or emulator instead. The advantage of simulation is that developers can debug and test much faster than on the target. The reason simulation is faster is that it removes the constant cross-compile, flash, and debug process. It can also dramatically remove developers' need to develop and provide prototype hardware, saving cost and time. Finally, the software can also be run within a test harness to verify each unit behaves as expected and that the integration of modules works as expected.

Modern software development is well beyond just the basic build processes we have discussed. If you look again closely at Figure 1-4, you'll notice on the lower right that there is a cloud repository that has been sectioned off from the rest of the diagram. The cloud repository serves a great purpose in software development today. The cloud repository allows developers to implement a continuous integration and continuous deployment (CI/CD) process.

CI/CD is a set of practices that enable developers to frequently test and deploy small changes to their application code.[15] CI/CD is a set of best practices that, when leveraged correctly, can result in highly robust software that is easy to deploy to the devices regularly as new features are released. A general overview of the process for embedded developers can be seen in Figure 1-8.

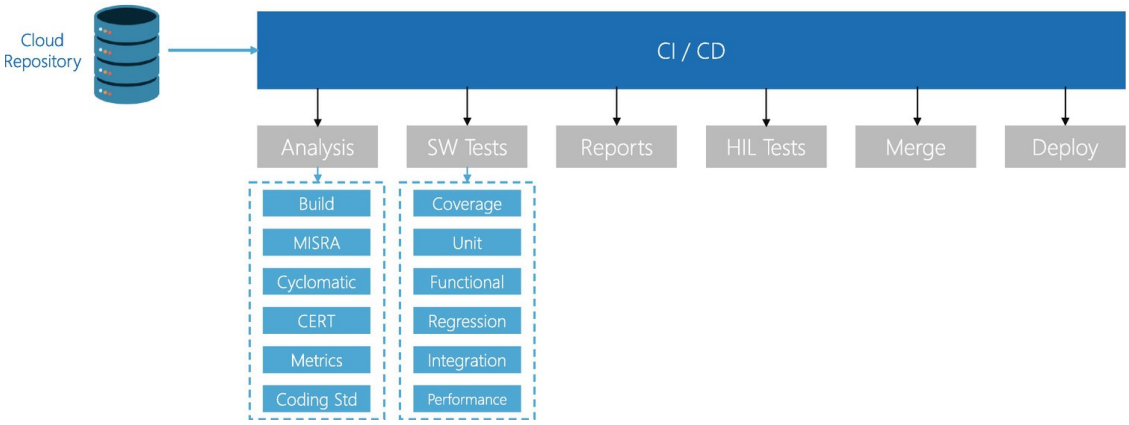

Figure 1-8 Modern development processes utilize a CI/CD server with various instances to continuously analyze and test the code base

Every team will have slightly different needs, which allows them to customize their CI/CD process. We will discuss CI/CD in more detail later in the book. For now, it's helpful to understand that it is a great way to automate activities such as

- Software metric analysis
- Software testing
- Reporting activities
- Hardware in-loop testing
- Merge requests
- Software deployment

Getting an initial CI/CD system set up can be challenging at first, but the benefits can be fantastic!

The Age of Modeling, Simulation, and Off-Chip Development

Simulation and off-chip development are by no means something new to embedded software developers. It's been around at least since the 1990s. However, at least 90% of the teams I talk to or interact with have not

taken advantage of it! By default, embedded software developers feel like they need to work on the target hardware to develop their applications. In some cases, this is true, but a lot can be done off-chip to accelerate development.

At its core, embedded software handles data that interacts directly with the hardware. But, of course, PC applications also interact with hardware. Still, the hardware is so far removed from the hardware with software layers and abstractions that the developer has no clue what is happening with the hardware. This makes embedded software unique because developers still need to understand the hardware and interact with it directly.

However, embedded software is still just about data, as I've tried to clarify in the design principles we've discussed. Therefore, application development for processing and handling data does not necessarily have to be developed on target. But, on the other hand, developing it on target is probably the least efficient way to develop the software! This is where modeling and simulation tools come into the picture.

A simulation is "a functioning representation of a system or process utilizing another system."[16] Embedded software development has approached a point where it is straightforward for developers to run their application code either in an emulator or natively in a PC environment and observe how the software behaves without the embedded target present. Simulation provides developers with several benefits, such as

- Reducing debugging time by eliminating the need to program a target
- Ability to examine and explore how their application behaves under various conditions without having to create them in hardware artificially
- Decoupling software development from hardware development

Modeling is very closely tied to simulation. In many cases, developers can use a modeling tool to create a software representation that can run simulations optimally. Simulations exercise the application code in a virtual environment, allowing teams to collect runtime data about their applica-

tion and system before the hardware is available. Today, there is a big push within embedded communities to move to model and simulation tools to decouple the application from the hardware and speed up development with the hope that it will also decrease costs and time to market. At a minimum, it does improve understanding of the system and can dramatically improve software maintenance.

Final Thoughts

The challenges that we face as teams and developers will dictate the principles and processes we use to solve those challenges. The challenges that embedded software teams face aren't static. Over the last several decades, they've evolved as systems have become more complex and feature rich. Success requires that we not just define the principles that will help us conquer our challenges, but that we continuously improve how we design and build our systems. We can continue to use traditional ideas and processes, but if you don't evolve with the times, it will become more and more difficult to keep up.

Action Items

To put this chapter's concepts into action, here are a few activities the reader can perform to start applying design philosophies:

- First, make a list of the design challenges that you and your team are facing.
- From your list of challenges, what are the top three big ones that require immediate action?
- Next, review the seven modern design philosophy principles and evaluate which ones can help you solve your design problems.
- Identify any additional design philosophies that will guide your designs.
- Identify the behaviors that are decreasing your chances for success. Then, what can you do over the next week, month, or quarter to mitigate these behaviors?
- Review your design and development processes. Are there any areas where you need to add more processes? Any areas where they need to be lightened up?

- Does your development more closely resemble traditional or modern development? What new modern technique can you focus on and add to your toolbox?
- Evaluate whether your entire team, from developers to executives, are on the same page for following best practices. Then, work on getting buy-in and agreement to safeguard using best practices even when the heat is on.

These are just a few ideas to go a little bit further. Carve out time in your schedule each week to apply these action items. Even minor adjustments over a year can result in dramatic changes!

Footnotes

1 https://spectrum.ieee.org/tech-history/silicon-revolution/chip-hall-of-fame-intel-4004-microprocessor

2 https://en.wikichip.org/wiki/intel/mcs-4/4004#:~:text=The%204004%20was%20a%204,and%20640%20bytes%20of%20RAM

3 https://en.wikipedia.org/wiki/Texas_Instruments_TMS1000

4 High-end, modern microcontrollers are looking more and more like general computing devices ...

5 www.embedded.com/why-wont-the-8-bit-microcontroller-die/

6 www.merriam-webster.com/dictionary/philosophy

7 Atkinson, Roger (December 1999). “Project management: cost, time and quality, two best guesses and a phenomenon, its time to accept other success criteria.” International Journal of Project Management. 17 (6): 337–342. doi:10.1016/S0263-7863(98)00069-6.

8 www.pmi.org/learning/library/beyond-iron-triangle-year-zero-6381

9 I believe this is a perception issue, not a reality.

10 https://en.wikipedia.org/wiki/Software_design_pattern

11 www.vocabulary.com/dictionary/perfection

12 https://en.wikipedia.org/wiki/Pareto_principle

13 www.zdnet.com/article/victor-aims-to-use-iot-to-build-a-better-mousetrap/

14 A suggestion provided by Jack Ganssle.

15 www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html

16 www.merriam-webster.com/dictionary/simulation