Compliments of IBM

IBM Limited Edition

Systems Engineering FOR DUMMIES®

Learn:

- What systems engineering is
- How systems engineering can help you develop smart, connected products
- How to expedite time-to-market, ensure business agility, and deliver high-quality smart products
- How to cut costs

Cathleen Shamieh

Systems Engineering FOR DUMMIES®

IBM LIMITED EDITION

Cathleen Shamieh

Wiley Publishing, Inc.

Systems Engineering For Dummies®, IBM Limited Edition

Published by
Wiley Publishing, Inc.
111 River Street
Hoboken, NJ 07030-5774
www.wiley.com

Published by Wiley Publishing, Inc., Indianapolis, Indiana

ISBN: 978-1-118-10001-1

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Publisher's Acknowledgments

We're proud of this book and of the people who worked on it. For details on how to create a custom *For Dummies* book for your business or organization, contact `info@dummies.biz`. For details on licensing the *For Dummies* brand for products or services, contact `BrandedRights&Licenses@Wiley.com`.

Some of the people who helped bring this book to market include the following:

Acquisitions, Editorial, and Media Development

Project Editor: Carrie A. Burchfield

Editorial Manager: Rev Mengle

Sr. Acquisitions Editor: Katie Feltman

Business Development Representative: Sue Blessing

Custom Publishing Project Specialist: Michael Sullivan

Composition Services

Sr. Project Coordinator: Kristie Rees

Layout and Graphics: Melanee Habig

Proofreader: Jessica Kramer

Publishing and Editorial for Technology Dummies

Richard Swadley, Vice President and Executive Group Publisher

Andy Cummings, Vice President and Publisher

Mary Bednarek, Executive Director, Acquisitions

Mary C. Corder, Editorial Director

Publishing and Editorial for Consumer Dummies

Diane Graves Steele, Vice President and Publisher, Consumer Dummies

Composition Services

Debbie Stailey, Director of Composition Services

Business Development

Lisa Coleman, Director, New Market and Brand Development

Contents at a Glance

Introduction 1

Chapter 1: Generating Smarter Products 3

Chapter 2: Taming the Tiger with Systems Engineering 13

Chapter 3: Revolutionizing Requirements. 23

Chapter 4: Getting Abstract with System Modeling . . .35

Chapter 5: Ensuring Tip-Top Quality 43

Chapter 6: Enabling Large Teams to Collaborate and Manage Changes 51

Chapter 7: Ten Ways to Win with Systems Engineering 61

Introduction

Smart products are everywhere. They track your packages, control traffic lights, fly aircraft, and guide you to your destination. They're at the heart of the systems and services you use every day — from smartphones to smart cars, to medical systems, and to aerospace and defense systems.

Intelligent, instrumented, and interconnected products are revolutionizing the way we interact with each other and perform everyday tasks. Through a combination of electronics, software, sensors, and other hardware, we have the technology to create multifunctional customized products. And with our unlimited imaginations, we have the potential to define hundreds of innovative, value-driven, and personalized systems and services.

The real challenge in creating smart products is one of organization: how can we effectively and efficiently integrate a complex combination of technologies to create an intelligent "system of systems" that fulfills its promises and lives up to its potential? The solution lies with systems engineering.

Systems Engineering For Dummies, IBM Limited Edition, explains what systems engineering is and how it can help you harness the complexity inherent in developing smart, connected products. If you're looking for ways to expedite time-to-market, ensure business agility, and deliver high-quality smart products while cutting costs, *Systems Engineering For Dummies,* IBM Limited Edition, is the book for you.

How This Book Is Organized

The seven chapters in this book are geared to help you understand the problem that systems engineering aims to solve and all the steps involved in solving it. Here's an overview of what's inside:

- Chapter 1 explains what smart products are and why they warrant a new approach to systems development.
- Chapter 2 provides a high-level overview of systems engineering.
- Chapter 3 lays out the critical role requirements play throughout the systems development cycle.
- Chapter 4 shows you how models can enhance your understanding of system structure and behavior.
- Chapter 5 explains how to make sure you build the right system (validation) and build the system right (verification).
- Chapter 6 suggests ways to enhance collaboration among development teams.
- Chapter 7 provides a look at some real-world companies that have incorporated systems engineering into their core business practices.

Icons Used In This Book

As you read this book, you'll notice several eye-catching icons designed to highlight special information.

This icon alerts you to key concepts you might want to commit to memory.

This icon appears next to actionable suggestions that are meant to make your life a lot easier.

These are the opposite of tips. They're suggestions that, if ignored, are bound to make your life a lot harder.

I realize that you don't necessarily need to know everything I do, so this icon tells you a little more detail than is absolutely necessary, so you can skip it if you like.

Chapter 1

Generating Smarter Products

In This Chapter

- Dealing with the demand for intelligent, interconnected systems
- Recognizing challenges on the road to success
- Shifting gears to encompass a broader landscape

Picture this: As you back down your driveway, your car sends a signal to your home to arm the alarm system and close the garage door. Your cell phone automatically synchronizes with your car's voice command system, and the built-in global positioning system (GPS) analyzes live traffic patterns and suggests a time-saving alternate route to work. Your car announces it's due for an oil change, checks your schedule on your smartphone, suggests possible appointment times, and offers to initiate a call to your favorite service station. Your car can even inform you of potential flooding conditions expected during your commute home.

Is it possible that a vehicle that was once known as a "horseless carriage" could be so incredibly smart and helpful? Absolutely!

In this chapter, you discover what makes smart products tick, how they collaborate with each other, and why you should adopt new business processes for developing them.

What's So Smart about Smart Products?

Smart products are all the rage nowadays. It's hard to imagine life before programmable kitchen appliances, interactive video games, and multitasking cell phones that record videos, print pictures, surf the web, and play music. Aircraft can find their way while avoiding collisions, and we count on intelligent drones and other precision defense systems to keep us safe.

Just what is it that makes these and other inanimate objects so capable of performing such amazing feats?

Delivering value with smart systems

Intelligent, software-driven systems are popping up all over the ecosystem:

- **Healthcare:** Custom software provides reliable, secure access to complex medical images and reports via mobile devices, enabling medical professionals to review patient information on the go and expedite emergency diagnoses.
- **Utilities:** Bi-directional communication between energy suppliers and consumers over a "smart grid" facilitate intelligent control of energy usage. For instance, washing machines can be turned on by the grid when power is least expensive, and selected appliances can be turned off during peak usage intervals. Smart cars are another example in this area.
- **Intelligent appliances:** Connected home appliances provide status updates on energy usage through intuitive user interfaces. Remote control of these appliances enables consumers to achieve desired comfort levels while reducing energy consumption.
- **Entertainment:** Smart TVs provide wireless Internet access, enabling consumers to rent movies, shop, check the weather, set up customized news pages, and download apps.
- **Automotive:** Smart collision avoidance systems integrate differential GPS technology, wireless communication, and in-vehicle graphical displays to alert drivers to nearby vehicles and even take emergency evasive action.

Blending technological ingredients

Today's smarter products and services are the result of the convergence of manufacturing, electronics, and information technologies. Quick-thinking manufacturers realized that they could take advantage of the tremendous advances in microelectronics, software, mechanical devices, sensors, and actuators to create products that would wow their customers — and wallop their competitors. So they grabbed a little bit o' this and a little bit o' that, mixed it all together (with some help from their engineering friends), and — *voilà*— cooked up products that even George Jetson would find innovative!

Smart products come in all shapes and sizes, but generally speaking, they're characterized by these three adjectives:

- **Instrumented:** Smart products sport devices, such as cameras; motion detectors; position sensors; wireless receivers; sound, heat, and light humidity; and magnetic fields, which constantly monitor their own operation and sniff out the neighborhood around them. By establishing context in this way, smart products can adapt to their environment in real time.
- **Interconnected:** When two or more products interact with each other and share information, they can deliver value that extends beyond the capabilities of each individual product. Connect them to the Internet and back-office or other IT systems, and the sky's the limit!
- **Intelligent:** Using sensory data, historical trends, and user profile information, well-designed products can make predictions, optimize outputs, and customize the user experience.

Weaving it all together with software

There's no question that the single most important driver of the smart product revolution is the phenomenal growth of processing power. As the brains behind every smart product, embedded microprocessors run algorithms, analyze data, perform heavy-duty number crunching, and control all the mechanical and electronic components of a smart product.

In order to get the job done, microprocessors run thousands — sometimes millions — of lines of software code. For instance, today's top-of-the-line cars contain dozens of microprocessors that maybe run 100 million lines of code — for the express purpose of delivering 250 to 300 functions to drivers and passengers.

As semiconductor manufacturers keep churning out more and more powerful microprocessors, the opportunity for software developers to create new-fangled whiz-bang functionality is skyrocketing. That's a good thing — because much of the hardware that differentiated products is quickly becoming commoditized. More and more often it's software, rather than electronics, that makes a product stand out and determines which products win market share.

The inherent flexibility of software offers loads of opportunity to develop additional features and functions, enabling manufacturers to upgrade their products to meet customer expectations of novelty. For instance, today's leading MP3 players do much more than just play music; they host music libraries, stream video, support messaging, provide games, and run third-party applications. The best products can be easily updated to add functionality so they keep pace with changes in the market.

Going viral with open standards

By 2013, there will be a mind-boggling 1.2 billion connected consumer electronic devices sitting in 800 million homes that have broadband access. If you have some good ideas about offering value-added services to customers via the Internet, you're probably salivating at these numbers.

Imagine if all electronic devices were created in a vacuum, with each manufacturer developing its own way to connect to the Internet and communicate with the outside world. There may be a lot of chatter on the lines, but none of it would be useful. What a waste of an opportunity!

Not to worry! Some really smart people with great visions of the future came up with the idea to create *open standards,* or agreed-on ways of sending messages between devices and service providers. The more companies that choose to use the communications protocols defined by open standards,

the more opportunity there is for everyone. The population is quickly evolving toward an "Internet of things" — a global ecosystem of smart, connected products and services.

Catering to Customers

A sizable chunk of the world's population consists of experienced users of smart products.

In 2010 alone, roughly 40 million personal navigation devices were sold throughout the world, and in 2008, over 55 million people snatched up iPods.

If you're a frequent flyer, you don't fret if the weather turns ugly during a cross-country flight because you know the airplane you're flying on is equipped with smarts that ensure your safe passage. Savvy users are well aware of the capabilities of today's smart products — and are driving demand for even smarter ones.

Today's customers are demanding reliable, real-time, interactive products — and they want them now! And if you're lucky (and smart) enough to deliver on time, don't rest yet, because before you know it, your customers will already be expecting an upgrade!

By incorporating personalization and integration capabilities into your designs, you create products that cater to the user's individual needs and adapt to the user's environment. Customers want products that promise to make their lives easier and more enjoyable, yet each of them has a unique way of getting things done, and a distinct set of values, pet peeves, and idiosyncrasies. Customers want smart products as easy to customize as downloading the latest smartphone app!

Realizing that Smart Products are not Islands

You may realize that it's no longer enough to create a really cool, feature-rich, stand-alone product. Today's smart products must know enough to interact with other smart products.

For example, take a look at a modern car (see Figure 1-1). A typical top-of-the-line car contains a whole slew of sophisticated software-intensive subsystems designed to make driving fun while improving safety and optimizing fuel efficiency: anti-lock brakes, collision avoidance, comfort controls, security, and much more. Designing and building a smart car — which is really a "system of systems" — involves the complex integration of mechanical, electrical, and electronic components, not to mention the proper execution of about, oh, say, 100 million lines of code.

To complicate things even more, smart cars also interact with several other systems external to the car itself. Location-based services, vehicle diagnostic systems, and hybrid/electrical recharging systems are just a few of many systems a car may interact with in a larger automotive ecosystem.

If it sounds complicated, it is! But it's also incredibly useful. For instance, if your car's security system is engineered to interact with emergency response centers, it can deliver accident details to first responders based on data collected from sensors within the car. Critical details, such as force of impact, can assist a 911 operator in determining the most appropriate rescue resources to dispatch.

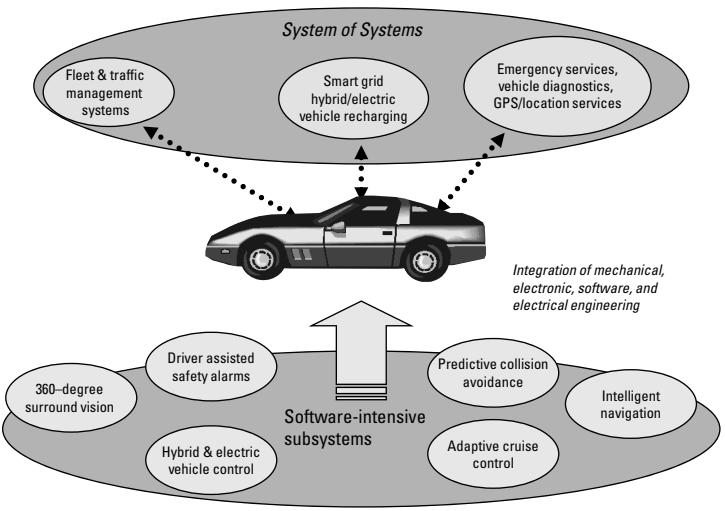

Figure 1-1: A smart car is a system of systems within a larger ecosystem.

Often, the most valuable features of a smart product aren't contained entirely within that product itself but are delivered as a result of interactions with other products or services within an ecosystem. Just like people, smart products need to collaborate and share information!

Just because you *can* share data doesn't mean you *should!* Privacy, security, and regulations (among other things) may have major impacts on your design!

Identifying New and Exciting Challenges

Developing a successful smart product that offers a personalized experience to a diverse assortment of fickle, demanding customers who wanted it done yesterday is, well, far from easy. You may be a revered expert in your chosen field, desperately trying to come to terms with the fact that neither your elite education nor your vast experience has prepared you for the enormous challenge you face today.

Before you head for the hills, in order to develop successful smart products, address the specific issues:

- **Mastering multiple capabilities:** You need expertise in multiple technical fields, including manufacturing, electronics, mechanical engineering, and software engineering. While most companies are strong in one or two of these areas, this expertise is rare to find all under one roof.
- **Becoming a world-class software house:** If you're a product manufacturer who considers software a necessary evil, you'd better start going to hypnosis, because most of the "smarts" in smart products come from software — and it doesn't code itself (not yet, anyway).
- **Integrating hardware and software development efforts:** As software-driven functionalities take center stage, you have to get your hardware and software teams to *really* work together — not just throw finished modules over the wall for integration and testing.
- **Effectively managing distributed teams:** If your development teams are located in different cities, time zones, and/

or countries, try facilitating collaborative working arrangements to ensure efficient, accurate, and cooperative results.

- **Enabling interactions with other systems:** You don't want to have the only product on the block that's incapable of interacting with the Internet, back-office IT systems, and other interconnected systems. Stand-alone products aren't that smart — and will more than likely become fall-alone failures.
- **Ensuring compliance:** Even if your product isn't directly tied to regulatory or industry standards, it may interact with a product or service that is, so play it safe — so all the other kids will want to play with you.
- **Shifting your design priorities:** Prioritize your design for upgrade capabilities rather than product stability. It's more about ensuring you have the *right* design priorities. For aero this may be safety; for defense this may be security; for consumer electronics this may be upgrade capability.
- **Shrinking product lifecycles:** As customer demand for new features shortens the service life of even the best products, your job is to make sure you hit the market at the right time with the right set of features.
- **Adapting to ever-changing requirements:** Changing customer needs, dynamic competitive and market conditions, and reprioritized corporate goals are all valid reasons for those irritating change requests. If you're smart, figure out how to profit from change.

Probing Potential Pitfalls

Failure to address the challenges associated with building a new-fangled smart product is likely to land you in hot water — not to mention tarnishing your corporate image and damaging your bottom line.

Mistakes you may think are small and easy to fix in a stand-alone product are often magnified and distributed throughout an interconnected system design. A software bug can wreak havoc if you don't catch it early on in the development process, increasing your costs and causing your schedule to slip. With the amount of software in devices doubling every two years, it's easy to understand why 66 percent of device software designs are completed over budget, and 24 percent of large projects are canceled due to unrecoverable schedule slippages.

If you're unable to develop complex products in a shorter cycle without compromising quality, you stand to lose revenue and tarnish your brand. Yet, smart, interconnected products often have hundreds — even thousands — of unique requirements, making it difficult to imagine how you can possibly maintain quality while shrinking development cycles.

If you're not equipped to respond quickly to new market demands or competitive threats, don't be surprised if you lose market share to more nimble organizations. For many organizations, just getting the initial design right is challenging enough: nearly one-third of all devices produced today simply fail to meet performance or functional specifications. You can bet these devices will be voted off the island early!

Just because you have top-notch technical talent working diligently on system design doesn't guarantee a successful outcome. It's many a system that has fallen short as a result of failures in subsystem interface specifications, requirements fidelity, or communication of key knowledge — not engineering.

Recognizing the Need for a Paradigm Shift

After you've accepted the fact that in order to thrive in the current marketplace, you need to change the way value is delivered in your products, you can start to re-examine your product planning, management, and development processes. And right away, you'll see the need to shift from a focus on cost to a focus on innovation and change — with software as the foundation for differentiation.

Back in the old days, when hardware was king, 3D computer-aided design (CAD) and mechanical bill-of-material (BOM) management were the be-all and end-all of cutting-edge sequential product development (see Figure 1-2). Your hardware development team used a CAD system to design hardware to meet a set of requirements, while your software team worked with a different, but related, set of requirements to produce the necessary code. Your CAD design and BOM were handed off to manufacturing, which figured out the fastest and cheapest way to build the product. Finally, integration and test engineers (in yet another department) loaded the software and tested the overall product.

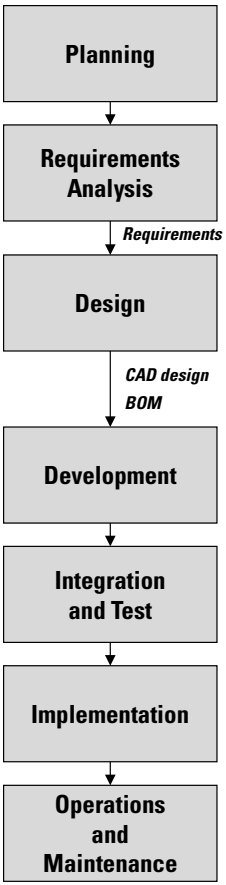

Figure 1-2: Sequential product development.

In today's world, that sort of sequential, document-driven development process simply doesn't cut the mustard. Imagine trying to respond rapidly to unexpected market events or to new feature requests. Each change requires that you start from the beginning and work through the entire sequential process. Your business would dwindle away in no time flat!

If your business depends on a smart system for its livelihood, you need to let go of the sequential, document-centric processes of the past and develop an entirely new set of core business processes for the future.

Chapter 2

Taming the Tiger with Systems Engineering

In This Chapter

- Scoping out systems engineering
- Tempering project euphoria with real-world reality
- Modeling your design

Have you ever wondered how NASA successfully managed to develop as complex a system as the Apollo spacecraft? How disparate teams of design engineers, programmers, third-party manufacturers, and others worked together to pull off the first manned flight to the moon? Well, they couldn't have done it without the help of systems engineering.

Systems engineering is an interdisciplinary approach to creating large, complex systems that meet a defined set of business and technical requirements. The aerospace and defense industries have been using systems engineering for a long time, and much of what they've learned is now being applied in other industries. As transport systems, powergrids, and telephone and network systems become smarter, you need space-age methods to build them.

Systems engineering has been around since the 1940s, but it started gaining traction beyond the likes of NASA in the 1990s. That's when manufacturers began to transform ordinary products into smart systems by incorporating information technology — marking a turning point in product development. Only recently has software crept in to assume a leading role.

By enabling seemingly boundless product functionality, software has taken a leading role in all sorts of products, enabling many new kinds of interconnections between parts of the product and between the product and the world. More connections mean exponential increases in system complexity — and leave companies no choice but to eliminate silos of development and devise new ways to manage complexity.

In this chapter, you find out what exactly what systems engineering is, how it can help you manage complexity, and how it can help you develop smarter products — and innovate your way to the top.

Getting to Know Systems Engineering

If you were to ask five experts exactly what systems engineering is, you'd probably get five — maybe even six! — different answers. That's partly because the term *systems engineering* is used for several different things and partly because the concept of systems has been morphing for decades, so the term *systems engineering* has no choice but to morph right along, too!

Not to worry, though. While some PhDs may get hot under the collar about the finer details and scope of systems engineering, most experts agree on its foundation. In this section, you take a look at what that foundation is — and how to build on it.

Seeing the forest and the trees

Systems engineering is both a practice and a process (see Figure 2-1):

- As a practice, it's concerned with the big picture: how a system functions and behaves overall, how it interfaces with its users and other systems, how its subsystems interact, and how to unite various engineering disciplines so that they work together.
- As a process, it spells out a robust, structured approach to system development that can be applied at a system-of-systems level or within specific engineering disciplines.

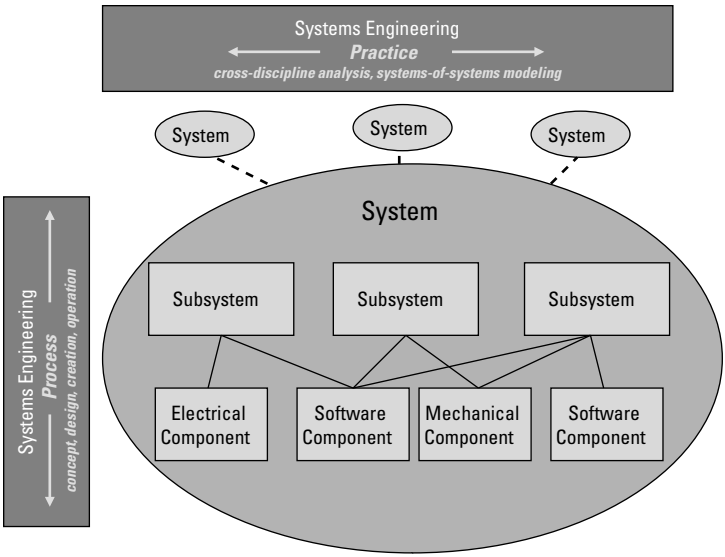

Figure 2-1: Systems engineering is a both a practice and a process.

No matter how you slice it, systems engineering is all about applying discipline to the system development process. And that discipline comes in two distinct flavors:

- **Technical discipline** ensures that you rigorously execute a sensible development process, from concept to production to operation.
- **Management discipline** organizes the technical effort throughout the system lifecycle, including facilitating collaboration, defining workflows, and deploying development tools.

Following some guiding principles

By blending breadth and depth, systems engineering aims to help you manage the details of a complex development effort while never losing sight of the overall goals of the project.

Begin by adhering to the following set of guiding principles:

- **Keep your eyes on the prize.** Define the desired outcome of a project right from the get-go, and don't stray from your goal no matter how crazy things get.
- **Involve key stakeholders.** Get input from customers, users, operators, C-level managers, and others as you make decisions at various stages of the development process.
- **Define the problem before assuming a solution.** By keeping an open mind about the means to your end, you're more likely to examine several alternatives — and choose the solution that best matches your desired outcome.
- **Break down the problem into manageable chunks.** Decompose your system into smaller subsystems, and then divide each subsystem into hardware or software components. Another related principle is to manage the interfaces between the chunks to ensure they integrate successfully and ultimately deliver the required emergent capabilities.
- **Delay specific technology choices.** Wait until you're well into the process before selecting physical components, so you avoid committing to technology that may be outdated or unnecessary by the time you're ready to implement your design.
- **Connect the dots between requirements and design.** Make sure you can justify your design decisions by linking them back to specific technical and business needs.
- **Test early, test often.** Take advantage of prototypes, simulators, emulators, and any other way to let everyone involved get an early look at the system. Make sure tests prove that you satisfy the requirements by linking them.

Exploring the Systems Engineering Process

It's great to have strong guiding principles, but how do you put them into action? Well, one way to do it is to develop a consistent process for systems engineering that embodies those principles.

Over the past 20 or so years, experts in complex system design have developed and refined what's known as the *V-model* of the systems engineering process (see Figure 2-2). The V-model is a graphical representation of a series of steps and procedures for developing complex systems.

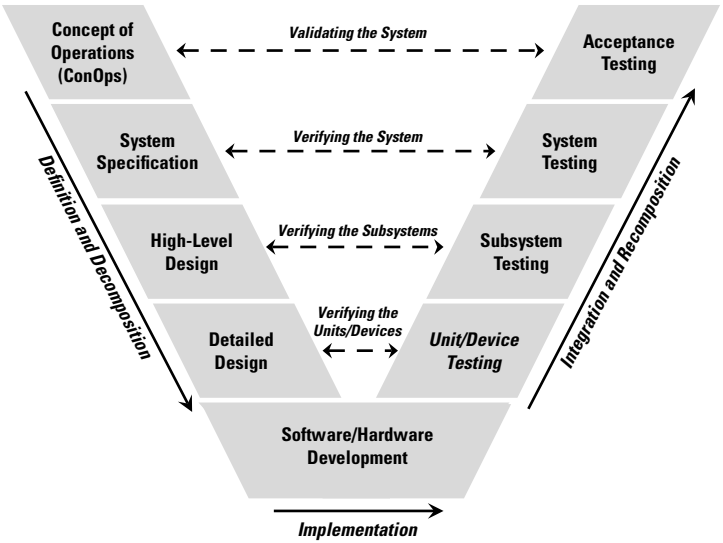

Figure 2-2: The V-model for the systems engineering process.

Tracing the "V" from left to right, you execute the systems engineering process in a series of steps, as follows:

- **Concept of Operations (ConOps):** Identify and document key stakeholder needs, overall system capabilities, roles and responsibilities, and performance measures for system validation at the end of the project.
- **System Specification:** Map out a set of verifiable system requirements that meet the stakeholder needs defined during ConOps.
- **High-Level Design:** Design a high-level system architecture that satisfies the system requirements and provides for maintenance, upgrades, and integration with other systems.

- **Detailed Design:** Drill down into the details of system design, developing component-level requirements that support within-budget procurement of hardware.
- **Software/Hardware Development:** Select and procure the appropriate technology and develop the hardware and software to meet your detailed design specs.
- **Unit/Device Testing:** Test each component-level hardware implementation, verifying its functionality against the appropriate component-level requirements.
- **Subsystem Testing:** Integrate hardware and software components into subsystems. Test and verify each subsystem against high-level requirements.
- **System Testing:** Integrate subsystems and test the entire system against system requirements. Verify that all interfaces have been properly implemented and all requirements and constraints have been satisfied.
- **Acceptance Testing:** Validate that the system meets the requirements and is effective in achieving its intended goals.

Throughout the systems engineering process, you create and refine system documentation. At each step on the left side of the "V" in Figure 2-2, you create the requirements that drive the next step, as well as a plan for verifying the implementation of the current level of decomposition. For instance, during the ConOps step, you create a high-level system requirements document that drives the System Specification step, and you create a System Validation Plan that drives Acceptance Testing. At each step on the right side of the "V", you create documentation to support system training, usage, maintenance, installation, and testing.

By linking all the steps on the left side of the "V" through requirements and referring back to these requirements as you work your way up the right side of the "V," you're much more likely to stay true to your original mission and maintain objectivity throughout the process. These linkages provide for what's known in systems engineering as *traceability.* Chapter 3 covers requirements and traceability in detail.

Managing Complexity with Models

Some models are useful, especially when you're working on a complex engineering project. If you can develop relatively inexpensive ways of designing, testing, and verifying your system *before* you go and build it, you can save yourself a lot of time and money — maybe even your job! One way to do this is to use models to design and refine your system throughout the development process.

System models allow you to capture complexity at many different levels, including system-of-systems (also known as ecosystem), system, subsystem, and component levels. They enable you to explore the details of each of these levels, known as *levels of abstraction* independently and hide or expose details as appropriate.

Models can take many different forms. At the simplest level it may just be a simple spreadsheet used to calculate some empirical system property. One the other hand, it may be a highly complex, interactive computer simulation.

For instance, say you want to understand how the automotive system (translation: car) that you're developing captures and forwards crash impact data to an emergency response system. You need to explore information about the car's sensors and how the car interacts with the external system, but you probably don't want to be distracted by extraneous details, such as wiring diagrams and component placement. The right model can show you what you need without extra detail.

Models also afford you the following benefits:

- They allow you to focus on the relevant information for the issue at hand, while ensuring consistency throughout your design.
- They capture both the structure (architecture) and behavior (functionality) of a system, illustrating relationships and interactions between system elements.

- You can use models to predict behavior at various levels of abstraction, enabling you to explore different architecture alternatives early in the development process and perform trade studies to assess which design choices make the most sense.
- You can develop executable models, which allow you to validate your design against requirements before building your system.
- Models enable you to view the details of a system from multiple perspectives (for instance, architectural, logical, functional, physical, data, and user), so you can address specific concerns.
- You can exploit models to help manage the development effort.
- Modeling offers tremendous visibility into a system, providing a powerful focal point for discussion and mutual understanding.
- Models offer a shared space to hold thoughts and decision criteria, making it easier for you and your development team to collaborate.
- Perhaps most importantly, the system model provides a synchronization point across multiple engineering disciplines, offering a solution to one of the most significant problems in the development of smart systems: how to coordinate hardware and software.

Speaking the Same Language

Just as an architect uses a standard set of symbols to represent building elements in a drawing that will be interpreted by a construction manager, so a development team should use a common vernacular to represent system models in order to promote shared understanding.

In 2001, the International Council on Systems Engineering (INCOSE) along with the Object Management Group (OMG) initiated an effort to develop a common modeling language for systems engineering applications, and within a few years, the Systems Modeling Language (SysML) was born. Developed as an adaptation of the software-centric Unified Modeling

Language (UML), SysML is the defacto standard language for modeling systems and systems-of-systems.

SysML uses a simple diagram approach to model systems (so simple, some have called them cave paintings), where the basic unit of structure — a block — can be used to represent hardware, software, facilities, personnel, or any other element of a system. Through a series of nested *structure diagrams,* you define the internal structure and intended use of a particular system element (for instance, an antilock braking system). Then, through a separate series of nested *behavior diagrams,* you show how that system element interacts with others, and with *actors* (users, outside systems, or the environment), to accomplish or *realize* that behavior.

In addition to modeling the structure (architecture) and dynamic behavior (functionality) of the system, SysML also allows you to model requirements and performance parameters. For instance, for an automotive system, you can create a requirements diagram to specify a constraint, such as "come to a stop from 65 mph within 180 feet on a clean, dry surface," and parametric diagrams to specify the equations that govern the motion of the car. Best of all, you can use powerful new software tools — think of it like building a simulator for a new race car. This means you can take it for a run and check the handling before it's even built!

By defining and organizing model constructs, SysML forces systems engineers and architects to be very clear and precise when designing a system. This reduces ambiguity and leads to higher quality, shorter development cycles, and reduced costs.

A Good Picture is Worth a Thousand (or a Billion) Words

The best way to get your arms around modeling using SysML is to take a look at a SysML diagram. Figure 2-3 shows a simple *context diagram* for a modern car that has a built-in global positioning system (GPS) receiver and a home security control system remote control.

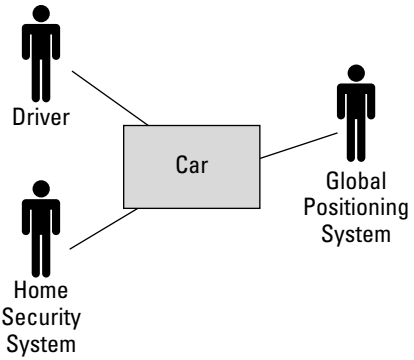

Figure 2-3: A simple context diagram for a car.

You use a context diagram to define the boundaries, or context, of your system. In this case, the system is the car, and it interacts with three *actors* outside the system: the driver, the GPS system, and the home security system. An actor is anything a system interacts with, whether it is a user, another system, or the environment. You use a context diagram to scope out the system under consideration.

By defining the system and its actors, you identify significant relationships, and thus, requirements and interfaces. You can then begin to map out interface specifications and data flows between the system and its actors.

Without a context diagram, you may overlook an actor or two — and come up short on your system requirements. For instance, if you failed to define the home security system as an actor in your car's ecosystem, your smart car won't be smart enough to arm your alarm system.

Chapter 3

Revolutionizing Requirements

In This Chapter

- Acknowledging that change is good
- Covering all the requirements bases by including use cases
- Cascading requirements through the development process
- Analyzing the impacts of change
- Getting a grip (on requirements management)

Back in the good ole' days, you developed requirements at the beginning of a project, you designed your product to meet those requirements, and if marketing came to you saying that the customer wanted a change (or two or three), you stomped your feet a bit before spouting off something about an onerous requirements change process that calls for 57 signatures.

Not so anymore. In a volatile, competitive market in which success hinges on satisfying customers more than ever before, you have two choices: either change or cry. In this chapter, you discover how to engineer system requirements with change in mind and how to ensure your system design satisfies the requirements.

Embracing a Philosophy of Change

In this new age of smarter products, you have to be able to respond quickly to market dynamics, such as changing customer needs, new competitive threats, or the latest regulatory standard. Product developers have also noticed over the years that requirements *should* change as your understanding of the need gets better through the development process. To the traditional system development process, however, change is the enemy. So what should you do?

Come up with a new process.

You may be happy to know that others have walked before you and have established an entirely new *requirements engineering* process. Consisting of much more than just upfront analysis and requirements definition, requirements engineering defines a soup-to-nuts process for establishing requirements, tying them in to testing, and facilitating change.

Requirements work best when they are engineered with change in mind. The philosophy of requirements engineering is that change is welcome. Change is, in fact, a goal!

Understanding Context

Before you begin to establish an initial set of requirements for a smart product or system, it pays to take some time to think about the problem your product is trying to solve. For instance, if you set out to build a car, it's critical that you understand exactly how it will be used and by whom. Will the car be used for city driving, highway driving, or racetrack driving? Does the target market consist primarily of elderly drivers, young men, or stay-at-home parents? Will the car be subject to harsh conditions, such as severe cold, salted roads, extreme heat, mountainous terrain, or high altitudes?

Understanding *context* is critical to developing systems that achieve marketing and business goals. By context, we mean the set of *actors* (for instance, users, other systems, and the

environment) with which the system interacts, and how interactions between the system and its actors take place.

Figure 3-1 shows a context diagram for a car with a built-in navigation system and a remote home security control system that is intended for use in winter climates. In this diagram, the car is treated as a "black box" that interacts with the following actors: the driver, the GPS system, the home security system, and the winter environment. Defining context helps you understand what functionality is required of the system and what exchanges occur between the system and its actors.

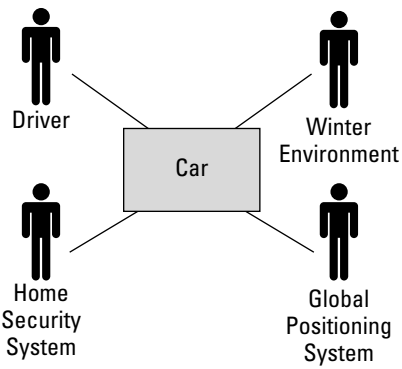

Figure 3-1: The context of a system delineates boundaries and specifies interfaces.

Understanding context also helps ensure that you have considered all the necessary requirements, relationships, and interfaces before you begin the development process. For instance, if you omit the "winter environment" actor for your car, you may fail to specify requirements for a "weather convenience package" that includes heavy-duty battery cables and a protective chassis coating.

At the highest level of abstraction, your system is a "black box" that interacts with an external set of actors. If you take a peek inside that black box, you see a set of interconnected subsystems (for instance, anti-lock brakes, navigation system, and so forth) that together make up your system. You can further decompose each subsystem into a set of interconnected components (and, perhaps, sub-subsystems). At every *level of decomposition* (for instance, system, subsystem, component) within your system, the context shifts.

Compare the context diagram for the car (see Figure 3-1) with the context diagram for the car's built-in navigation subsystem (see Figure 3-2). At the system level, the car is the black box that interacts with external actors. At the subsystem level, the navigation subsystem is the black box, and it interacts with a different set of actors: the driver, the car's electrical subsystem, and the GPS system.

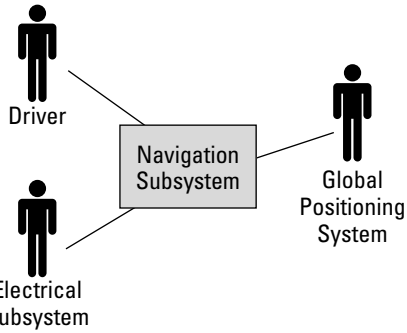

Figure 3-2: Context changes as you explore different levels within the system.

Understanding context delineates system boundaries and defines interfaces, setting the stage for the precise, accurate specification of system requirements.

Diving into Requirements

Think of requirements in three broad categories, or tiers:

- **Source Requirements.** These are the requirements you get from your customers or stakeholders. They may be broad and general, detailed and specific, comprehensive or fragmented — or most likely, some of all of these. As the saying goes, customers obey no rules when it comes to providing requirements — you get what you get.
- **Mission or Business Requirements.** These are the requirements that specify the operational context in which the system operates — not what the system does, but the part it plays in a larger world. For a new military aircraft, this may describe the kinds of missions it will fly. For a new smartphone, business requirements may

describe how it operates within the cellular carrier's communication infrastructure.

- **System/Subsystem Requirements.** These are the requirements that define what the system must be able to do. They start at a high systems level and are analyzed and decomposed to produce requirements for lower level subsystems. They may be expressed in common "shall" statements or in more advanced forms such as models and diagrams.

At each level of abstraction, requirements define what a system should do and how it should do it, but not how it should be implemented. With the ever-increasing complexity of today's smart systems, it's nearly impossible for you to nail down system requirements right from the start. And in a climate, such as consumer electronics, in which change trumps stability, you really don't want to freeze your requirements early in the development process.

Seeking broader input

It pays to capture as much information for your source requirements, from as many stakeholders as possible, early in the requirements engineering process. Of course, you start with your customers, but you also gather input about regulatory, industry, and safety standards that govern your system, interfaces and data exchanges with other systems (for instance, the GPS system), and business and marketing constraints.

What you're seeking ideally are requirements that describe the capabilities of the system rather than the functions. (For example, what the system should do rather than how it should do it.) If a customer describes a function, ask them why they want it. That should lead to a capability.

It's important, too, to specify both *functional* and *non-functional* requirements. You use functional requirements to describe what the system should do, given certain inputs. For instance, given a starting point and a destination, your navigation system should map out and display a route. You use non-functional requirements to specify performance or quality requirements or impose constraints on the design. These include requirements for things such as speed, capacity, reliability, weight, usability, and scalability.

To make sure you've covered all the bases, you develop *use cases* that describe all the possible ways the system's functions could be used. For instance, the function "map route" in your navigation system can be used to "find the nearest gas station" or "show me the hotels in the vicinity of my destination." Use cases are generally composed of sequences of one or more system functions. Use cases tell concrete stories of system usage and can be used in all three requirements tiers — source, mission/business and system/subsystem.

By capturing usage as you develop requirements, you're more likely to design a system that will provide real value.

Satisfying and deriving requirements

After you've scoped out your system context and defined an initial set of high-level system requirements, you use those requirements to drive the high-level design process (see Figure 3-3). During the design process, you develop additional requirements, such as a System Verification Plan to be satisfied during system test, or architectural design constraints to be satisfied by the detailed design.

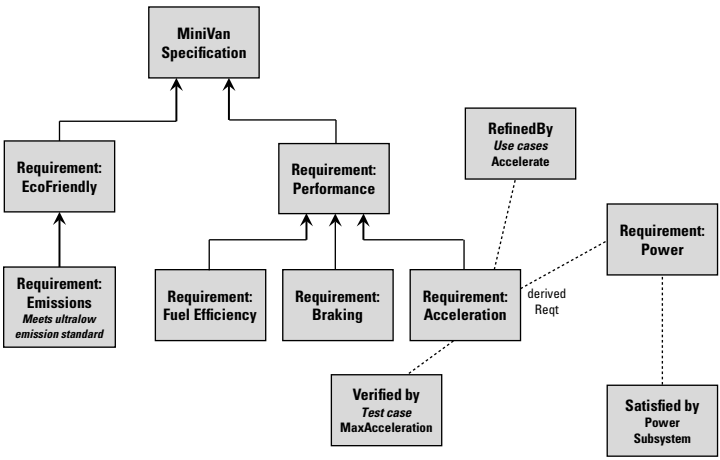

Figure 3-3: The high-level design process.

As you drill down into the depths of your design, you satisfy the requirements developed in the previous (higher) design level, and derive requirements to be used by corresponding test level as well as the next (lower) design level.

As you can see, the definition of "requirements" is expanding, as the lines between requirements and design become blurred, and as the linkages between testing and design become critical. Your final set of requirements is a combination of the initial high-level system requirements and requirements derived during the design process.

Creating Requirements Diagrams

Although old-fashioned text-based requirements are still necessary for large projects with contractual obligations, they often don't cut the mustard in today's world of complex, flexible, customer-focused smart products. Luckily, the widely-adopted Systems Modeling Language (SysML) gives you a framework for modeling requirements. While large sets of text requirements may still exist in a database, especially when they came right from the customer, there are new and better ways to visualize and work with sets of related requirements.

SysML enables you to create hierarchical requirements models that illustrate dependencies, classify requirements as original or derived, and capture design choice rationales. There are two high-level system requirements: EcoFriendly and Performance. (See Figure 3-3.) The EcoFriendly requirement has a specific Emissions sub-requirement.

The Performance requirement has three sub-requirements, one of which is Acceleration. The Acceleration sub-requirement is further refined by specific use cases. A test plan for maximum acceleration is defined (and is a requirement that the testing process must satisfy). Finally, the Acceleration sub-requirement generates a derived requirement for power, which is satisfied by the Power subsystem.

SysML requirements diagrams enable you to easily view relationships and answer questions, such as, "What requirements does the Power subsystem satisfy?" or "What would be the

impact of changing the Cargo Size requirement?" When you treat requirements as an integral part of the system architecture, it's much easier to visualize the impact of changes to requirements on your system. And isn't that one of the goals of systems engineering?

Driving Design Decisions with Requirements

As your requirements are cascaded through the design process, the nature of certain requirements may lead you to make specific design choices. For example, if you are designing an airplane collision avoidance system, at the start of the process, you may have three design options, each of which satisfies the functional requirement to avoid collisions:

- Sophisticated onboard radar system
- Transponder system
- Primarily manual (communication with air traffic control)

Each design option has its own unique set of components, space requirements, and costs, and involves some level of collaboration between the system and its actors (for instance, air traffic control). As you further decompose your design, you may encounter a non-functional requirement, such as "maximum cockpit space," that limits your design options (the radar system just won't fit!).

Systems engineers use a technique called a *trade study* to evaluate options for the many design choices that must be made. It's essentially the same thing everyone does when making an important decision. Identify the important decision factors, score each alternative on each factor, add in some weighting adjustments and make the decision. Trade studies often require extensive research to evaluate the alternatives completely.

Keeping track of all the requirements and how they impact other parts of the system are necessary evils if you want to deliver successful products. Off-the-shelf software tools can help you manage system requirements.

If you overlook a requirement, you may run into real problems. For instance, you can design the most phenomenal automobile cup holder in the world, but if you locate it just below the car stereo — so your *grande latte* blocks the controls — because you failed to relate an "allow ample clearance" requirement to your design, you'll miss the mark. Mistakes like this can really cost you.

Juggling Requirements and Designs

For the airplane collision avoidance example in the previous section, say you select the "transponder system" design option. This semi-automated option involves air traffic control monitoring transponder signals from aircraft in the area. Your design choice triggers sub-requirements for a transponder subsystem design and a subsystem test plan.

You're well into your subsystem design, when you receive a requirement change request. The new requirement specifies that the aircraft must be able to independently avoid collisions. Your design choice doesn't satisfy the new requirement, so you must set aside your design choice and begin working with an alternative design option, the radar system.

Certain changes may produce derived requirements in a subsystem design that propagate back up to the system design. For instance, cost constraints may limit the choice of a system component, which may in turn constrain subsystem functionality, impacting the original requirements. In the old days (like a few years ago!) this change propagation process consumed many hours or even weeks, as engineers tracked down and modified the relevant documents and designs residing in word processing documents, presentation slides and spreadsheets. The introduction of models into the process helps a lot, as discussed in Chapter 4.

Establish a formal requirements change process so you can easily understand the impact of changes to your system design.

Linking Requirements to Testing

You've just completed the design of a new, super-responsive car navigation system. The prototype has been built, and your test team reports back that the system is working great. In fact, it calculates the route between two points with such blazing speed (thanks to your brilliant algorithm), it's bound to blow away the competition. Your boss is proud, and offers the prototype to your CEO to use for the day. The CEO is impressed, until she tries to use it to find the nearest Chinese restaurant — and loses her patience waiting for a response.

You wonder what went wrong, and then you realize that your algorithm was optimized for point-to-point route calculations, but not for finding targets within a range. And although your system test plan did test the functionality of the system, it did not test all use cases (or maybe someone forgot to write that use case).

A key component of the systems engineering process is establishing linkages between requirements and testing. You want to make sure you're building the system you set out to build — not just a system that "works."

At each level of system decomposition, as you refine and derive requirements, you create and refine test plans for verifying that the system satisfies the requirements. If requirements change, you change the test plans. Often the act of writing the test criteria for a set of requirements helps you improve and refine the requirements themselves.

It's a good practice to evaluate your requirements to be sure they *can* be tested and even specify early *how* they will be tested.

Making Sure You Leave a Trace

To make sure all your requirements are properly implemented, you must be able to trace each one throughout the development and testing process.

Requirements traceability is the ability to link every requirement to three related items:

- The stakeholder needs (source requirements) that it fulfills
- The system elements that implement or realize it
- The test case that verifies it

Traceability helps ensure that you comply with regulations and standards, avoid overlooking requirements, and stay focused on the overall goals of the project. By establishing end-to-end traceability links, you're able to evaluate exactly what is impacted by the latest requirement change or an alternative design choice — before you make the change.

Rewarding Your Hard Work

For large, complex systems, requirements management can be a nightmare. Many development teams consist of hundreds — even thousands — of architects and engineers, all of whom touch requirements, whether to create and edit them, or simply to review and understand them. Traceability often traverses four levels of decomposition as stakeholder requirements are successively cascaded through to component design and test requirements. Also, these levels of decomposition often traverse supply chain boundaries making life even more challenging.

Effective requirements management involves many engineering disciplines, including system design, architecture, software, mechanical, electrical, and test engineering. Business functions, such as marketing and procurement, also have a stake in requirements management.

Software tools can help you get a handle on the unwieldy requirements management process. Such tools are designed to maintain audit histories, preserve all changes, conduct

impact analyses, and automate the change management process. They can also alert you to overlooked requirements, over-engineered designs, and non-compliance.

Managing requirements is a tough business, but with a little help from the proper tools can result in a big payoff in terms of cost, schedule, project success — and your sanity.

Chapter 4

Getting Abstract with System Modeling

In This Chapter

- Decomposing your system into levels of abstraction
- Visualizing how the elements of your system fit together
- Picturing how your system will behave
- Refining system models through iteration

A formula is a kind of model. It captures mathematical relationships among input variables and represents them with a mathematical structure that applies over a wide range of inputs. Visualizing the formula helps you get a better understanding of the relationships between elements of the structure. And using the formula allows you to test a whole range of different possibilities until you find one that works for you.

In this chapter, you explore how you can use system models to manage complexity as well as abstract essential relationships within a system and test inexpensively before you build.

Modeling System Architecture

Architectural models enable you to capture the static nature of your system, including both the structure and the intended usage of the system. For instance, consider the simplified architectural model of a car, shown in Figures 4-1 and 4-2.

The partial structural model in Figure 4-1 decomposes the system into several levels. At the top level, you see the overall system: the car. Descend a level, and you see two of the many subsystems of a car: the anti-lock braking system (ABS) and the chassis.

The chassis subsystem further decomposes into another subsystem (the hub assembly), a mechanical component (the tire), and other subsystems and components not shown in this diagram. The hub assembly subsystem consists of an electronic component (sensor) and several mechanical components not shown in the figure.

The ABS subsystem decomposes into a mechanical component (rotor) and another subsystem (the anti-lock controller). The anti-lock controller subsystem consists of two electronic components: the traction detector and the brake modulator. The dotted line connecting the hub assembly sensor to the anti-lock controller shows that there is a relationship between the two, although the sensor is not an element of the anti-lock controller subsystem.

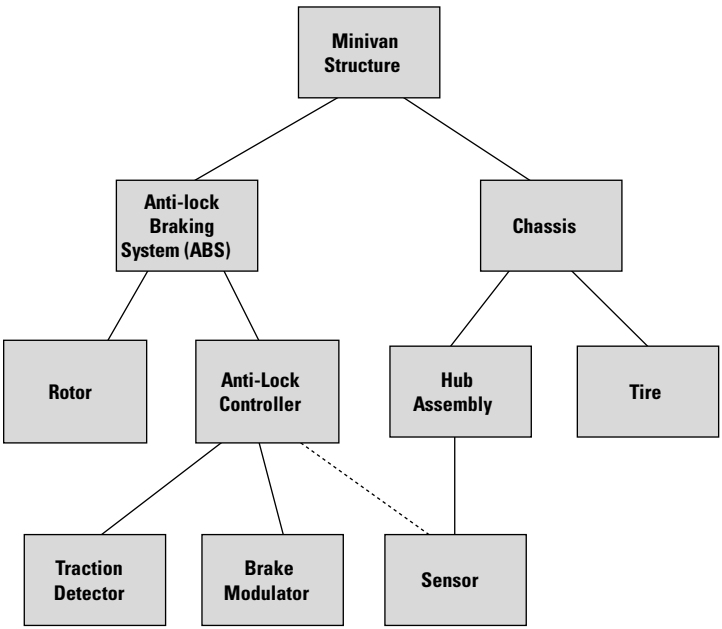

Figure 4-1: A simplified architectural model for a minivan system.

Structural models, like the simplified architectural model shown in Figure 4-1, capture the hierarchical structure of the system architecture and illustrate connections between elements of the model.

Figure 4-2 shows another piece of the full architectural model: an insider's view of how the parts of the anti-lock controller subsystem interact with each other and with the hub assembly sensor. In this case, the sensor output is connected to the input of the traction detector, and the traction detector output is connected to the input of the brake modulator. This simple diagram illustrates connections in order to indicate *intended* usage but doesn't specify the conditions under which the interactions actually occur. You use behavior models (which we discuss in the next section) to describe the sequence of events that must happen in order for the interactions to occur (for instance, the sensor reading indicates a loss of traction, which causes the traction detector to trigger the brake modulator).

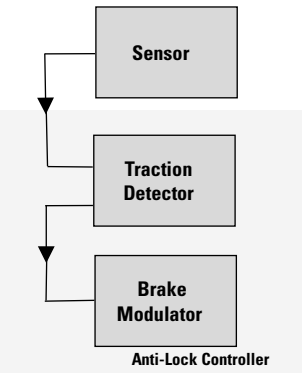

Figure 4-2: Architectural models capture usage information.

Structural models like these can show physical architecture, as in the examples above, or can be used to show logical architecture. Logical architectures are being used more and more in developing complex systems because they allow engineers to reason about functional interactions without specifying a particular physical structure. Take the car in Figure 4-1. A few decades back, car subsystems, such as brakes, engine, and radio, were all completely separate, and the physical architecture was all that was needed to understand and build them. Now, you may consider the logical architecture of a car to contain elements such as these:

- Propulsion, which may cross the old physical boundaries of brakes, electrical, engine, and controls
- Information Management, which includes all information received, stored, sent, or used within the car
- Entertainment, which may end up having physical components in common with propulsion and information
- Safety, which may have ties into many other car systems including, braking, information management, and driver user interface

By considering the logical architecture separately — and, ideally, before the physical implementation — systems engineers can produce better, more elegant approaches. A dual-view logical/physical architectural approach is a great way to manage the complexity of new, integrated smart products — which are just not designed like your grandfather's Oldsmobile.

Modeling System Behavior

To understand how a system behaves, you need to understand how the components in the architecture (logical and physical) of a system interact. System behavior models capture dynamic information about a system, such as state transitions, actions that a system performs in response to specific events, and interactions between collaborating parts of a system. Models also capture the flow of data and control between activities, like for instance how sensor output data is passed on to activities that take place in the anti-lock controller of a car, or how control is passed from one part of a system to another.

Figure 4-3 shows a simplified state diagram describing the operational states of a car. The car remains in one of four states — off, idling, accelerating, or braking — until a specific event occurs (for instance, "engage brake") that causes the car to transition to another state.

You can map connections between behavior models and structural models to enforce consistency through a process called *allocation*. For instance, you allocate the "detect loss of traction" action to the "traction detector" component in your system structure model, and you allocate the "modulate braking force" action to the "brake modulator" component.

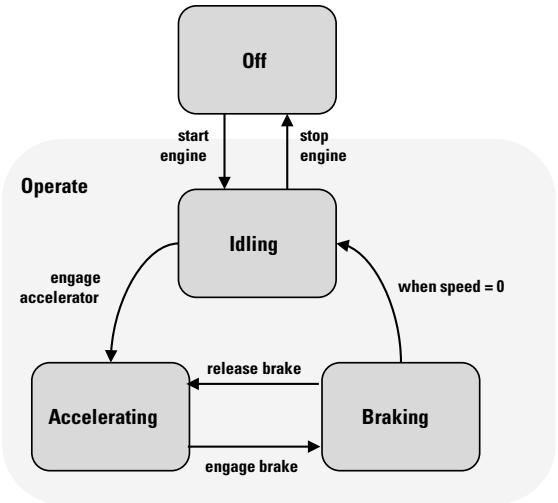

Figure 4-3: Behavioral model showing transitions between operational states.

For complex systems, you need hordes of little behavior models to represent all the activity that takes place. In many cases, one activity triggers an action in another part of the system, so your individual models are interconnected. By the time you're done modeling all the activity, you'll have one big, intricate, uniform, and consistent model of overall system behavior.

By using standard modeling constructs and semantics, such as the ones provided by Systems Modeling Language (SysML), you can use commercial software tools to automate the execution of system behavior models. Tools can automatically translate modeling constructs, such as state transition diagrams, into "if-then-execute" code statements. This way, you can simulate system behavior in the software, enabling you to perform "what-if" studies, look at design alternatives, and conduct impact analyses before you build your system. Trade studies, which used to take hours or days, can be completed in minutes in this way.

Mapping Out the Models

Industry standards, such as SysML, provide a foundation for common understanding of systems modeling throughout all

these stages. SysML models consist of a set of *diagrams* that represent the structure, behavior, requirements, and quantitative constraints of a system. SysML diagrams come in four different flavors:

- **Structure:** Describes what architectural elements (logical and physical), known as *blocks*, are contained in the system or subsystem and how they are connected. For instance, an anti-lock braking subsystem contains a traction detector and a brake modulator.
- **Behavior:** Describes how the system or subsystem behaves, including state transitions, sequences of activities, functions, and interactions. (For instance, "detect loss of traction" triggers "modulate braking force.")
- **Requirements:** Describes the specific requirements associated with the system or subsystem. (For instance, stopping distance specifics are given.)
- **Parametric:** Describes the parametric constraints placed upon the system or subsystem. (For instance, the braking force equation is a parametric equation.)

Exploring Four Stages of System Modeling

Systems are, by their very nature, recursive structures consisting of multiple levels, starting with the overall system at the highest level, and decomposing into subsystems, and then into components. The best way to model an entire system is to use a multi-step recursive process — starting at the top — that consists of the following four stages. The stages spell CURE so think of it as the cure for complexity:

- **Context:** Establish the boundaries of your system, identify the people and systems with which your system interacts (also known as the actors), and describe the interfaces (how they communicate with the system, and what they exchange). Together, the elements of this contextual model are known as the *enterprise* — meaning the system and its immediate surroundings. Use SysML block diagrams for this.
- **Usage:** Describe all the ways in which the actors use the system. Include who or what uses the system, and who

or what the system uses. This is best done in specific, step-by-step, narrative-like stories of system usage, such as use cases, and illustrated as SysML activity diagrams. Refine your system requirements, and make them specific and complete (remember, source requirements can leave out a lot of important stuff). The same usage scenarios become the basis for system testing later — based on what you've learned from this usage analysis.

- **Realization:** Define structure (architecture) and behavior (function) models that together describe how each usage is achieved by the system through collaboration among elements within the system architecture. Required behavior is realized (made real) in the elements of the system. Now, this is quite different from the traditional design process of allocating requirements to physical components and then hoping that it will work in actual usage. Here, usage is defined explicitly and then you design the system based on these specific usages, so you *know* the system is designed to meet usage requirements.
- **Execution:** Execute the behavior models to demonstrate that your design satisfies the requirements. Simple, executable models, even at high levels of abstraction are a great and cost-saving way to discover tricky problems, miscommunications, missing or ambiguous requirements, and other schedule-busting issues early on. They get everyone on the same page before anything is actually built.

Start at the highest level of decomposition in your system design: the system level (for instance, a minivan). After establishing context and usage models, you define your high-level architecture and behavior models based on the system requirements. Then you execute the models to demonstrate that they do, indeed, accomplish the system's intended uses. After you've completed the process for the system level, repeat the process for the next level of decomposition: the subsystem level.

Continue to drill down through levels of decomposition, shifting your context as you proceed to model at each level, until you reach the lowest level — the component level — where you specify the physical implementation of your design (for instance, electronics, software, or mechanical design).

At each level, reach horizontally "across the V" and perform verification and validation, using the model as the basis.

Use executable models and other kinds of mathematical and design simulations to do as much verification as you can before reaching the implementation stage.

The Holy Grail here is a complete system model — a kind of complete virtual reality version of the actual system. This isn't quite possible yet, but as modeling tools get better and learn to interconnect across engineering disciplines, we'll get closer and closer.

Understanding Why Modeling is in Fashion

You may think that modeling the architecture and behavior of a complex system is more trouble than it's worth — until you remember the pain you experienced on your last project, when no one on the development team owned up to overlooking a critical requirement, and the post-mortem meeting made the Spanish Inquisition look like a party.

Modeling allows you to capture all of the hairy details of your system design in an organized fashion, enabling you to visualize, understand, and assimilate the complexities of system structure and behavior. It allows you to explore different architecture and design options, perform trade studies, and assess the impact of changes before you begin to build your system — driving down project risk and development costs.

Models provide context for reasoning about system concerns at various levels. You can use models to explore multiple views of a system — planning, requirements, architecture, design, implementation, deployment, behavior, input data, output data, and more — so you and your colleagues can reason about big picture issues just as easily as about detailed design issues.

Using industry-standard modeling languages and techniques, such as SysML, reduces ambiguity and removes language barriers that might exist between members of diverse development teams, and provides a single master source for project development status and documentation. Improved collaboration and clear, precise documentation leads to greater efficiencies, shorter development cycles, and best of all, improved quality.

Chapter 5

Ensuring Tip-Top Quality

In This Chapter

- Integrating quality into the systems development process
- Iterating testing and design
- Using models to reveal errors early

It used to be that quality assurance was a process that took place at the end of the development cycle, as if quality was a tangible feature that you could add to the product before shipping. And when defects were discovered, it took a major effort to identify the root causes and fix the problems.

In today's world of increasingly complex software-driven smart products, quality must become an integral part of the systems development process for there to be any hope of delivering products that are not only defect-free, but demonstrate a fitness for purpose. In this chapter, you take a look at how systems engineering helps identify quality issues through validation and verification early in the development cycle, leading to improved chances for project success.

Exploring Levels of Testing

When you design a smart product that includes several subsystems along with thousands (or millions) of lines of code, you may wonder how you can possibly verify and validate the entire system. Where do you begin? How do you ensure all the pieces of the puzzle not only *fit* together properly but *work* together properly to deliver the product functionality your stakeholders (and stockholders) are looking for? Well, you actually start the quality process when you start to design the system.

Large, complex systems are decomposed into several levels, including the system level, one or more subsystem levels, and finally, the component level.

While it may seem intuitive to wait until the system is all put together to test it, good systems engineering practice includes more sophisticated techniques for ensuring quality as the system is built. As each component of the system (hardware or software) is built, it is tested by itself then integrated with others to form a subsystem. Next, the subsystem is tested by itself and then integrated with others to form the system, and the system is tested. Finally, the entire system is tested along with its operating environment (context) to make sure the whole thing does what it is supposed to do in the real world.

In addition to testing the system as it is being built, some verification and validation can be performed on models and simulations of the system, bringing problems to light early — before you bend any metal or solder any circuit boards. You can even perform verification activities during the early requirements gathering stage to ensure your interpretation of the customer need is in fact correct.

Unit testing

Testing at the lowest system level is tightly coupled with the implementation of your design. You test each hardware or software component, fix defects, and redo your implementation. For software, this may involve several iterative cycles of coding, testing, and recoding until you've thoroughly debugged the software.

After you've addressed known defects, you're ready to verify that the component satisfies the requirements allocated to it. Remember when you developed the detailed design of your system? Well, part of that detailed design phase involved defining specific component-level requirements and creating a Unit Verification Plan. Now, you use the test cases defined in that Unit Verification Plan to verify whether the component satisfies the allocated requirements. If defects are discovered, go back to your implementation and fix the defects. After the components have been thoroughly vetted, they are ready to be integrated into higher-level assemblies or subsystems.

Make sure your Unit Verification Plan thoroughly documents the specific test cases and results for each component and that you use a traceability matrix to tie these tests back to the specific requirements they verify. This ensures that if the tests all pass, the system satisfies all the requirements.

Subsystem integration and verification

Fully-verified hardware and software components are ready to be integrated into modules or subsystems. If you've tested out the interfaces first, this should proceed fairly smoothly.

The goal of this stage of testing is to ensure that all of the interfaces between components and assemblies have been properly implemented and that all subsystem requirements and constraints have been fulfilled.

Depending on the complexity of your subsystem, you may need to develop an integration plan that defines the order in which you integrate lower-level components and subassemblies. Plan your integration so pieces that need to work together are ready for integration at the same time, if possible. At each integration step, verify the functionality of the subassembly against the appropriate set of requirements, using the Subsystem Verification Plan you defined during the design phase.

Be careful not to ignore the tests you performed to verify requirements at the component level, since many requirements are cascaded through multiple levels of system decomposition. For instance, if a system requirement specifies that a display screen should go blank when a user presses a button, you need to verify that the screen component can blank itself (component-level test), and you need to verify that pressing the button triggers the blanking of the screen (subsystem-level test).

System testing

Through progressive iteration, you integrate, test, and verify subsystems until you reach the system level (see Figure 5-1). Each iteration consists of careful, thorough testing — with

particular emphasis on the interfaces — and verification that the appropriate requirements have been fulfilled. At the highest level, you perform tests to verify that the overall system fulfills the high-level system requirements defined early in the design phase. Once again, you base your testing on a verification plan that you defined in parallel with the system requirements.

It's important to document the results of each test case and to note any unexpected responses or other anomalies. However, you should resist the temptation to fix a defect as soon as you discover it, or you may lose configuration control. Instead, document the problem, analyze the cause, and define an action plan for resolving it within the context of a systematic process.

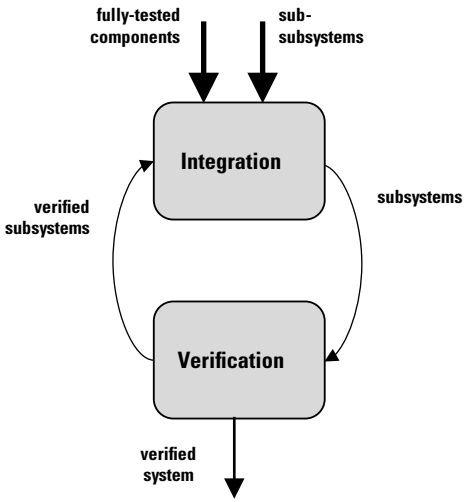

Figure 5-1: Integration and verification is an iterative process.

If all goes well, you'll have a verified system that you can demonstrate to stakeholders. You're able to prove that all system requirements have been satisfied, confirming that the system was built correctly.

System acceptance testing

"If you build it, they will come." If you buy into that philosophy, you may come up a bit short. Your system may be

designed and implemented perfectly, satisfying all the requirements and then some, but if it doesn't fulfill its intended use, it's bound to be a flop.

Take, for instance, the case of a well-engineered traffic light control system. Expertly designed to control the sequence of traffic lights in a large city, this system might fail to meet its *intended* purpose — to reduce congestion by streamlining the flow of traffic — if no one bothered to study the city's typical traffic patterns and map them to system requirements for timing sequences.

The goal of system acceptance testing is to validate that the system fulfills its intended purpose.

During the conceptual phase of your project, you identified key stakeholder needs, overall system capabilities, usage scenarios (CONOPS and use cases) and performance measures for system validation. You defined a System Validation Plan, and, if you were smart, you put it in a safe and didn't let anyone modify it to accommodate rogue objectives like skimping on quality to save a buck or two. Your System Validation Plan is the rock upon which you shall prove that your system achieves its intended goals.

Conduct system validation with real live users, and measure performance as defined by your plan, possibly including customer satisfaction. Validation may require data collection before, during, and after system deployment. After carefully documenting system performance, meet with stakeholders, analyze all the data, and assess project success.

When a system problem is observed during testing, usually there's a fault in the system that can be corrected by making a change to the system. Notice, however that the problem could be something else. If your verification plan or test procedure is wrong or outdated, then the test may fail due to no fault in the system. Similarly, if a requirement is wrong, ambiguous or incomplete (especially an interface requirement), that may be the source of the problem — all the more reason to focus on getting requirements and test plans correct early in the game.

When verification testing turns up a problem, go back and check your requirements and design, make the necessary adjustments, and repeat the testing.

For instance, say you're designing a rain-sensing wiper (RSW). The goal of the RSW is to automatically sweep the windshield upon detection of rain droplets on the exterior surface of a windshield. The high-level architecture of the RSW system includes an optical sensor with a specified operating range attached to the interior surface of the windshield, an electronic control unit, and software. Your design calls for the sensor and the software to interact in order to identify the presence of water and to activate the wiper.

You test the individual components and verify that the RSW system works as designed within the operating range of the sensor. Then you integrate the RSW, the windshield, and other subsystems into a car. When you test the operation of the RSW in the context of the entire system, lo and behold, it fails.

After an extensive root-cause analysis, you discover the source of the problem: the physical characteristics of the windshield (specifically, the optical index and thickness) are incompatible with the operating range of the sensor. You realize that you failed to define a requirement for the physical characteristics of the windshield that would ensure compatibility with the RSW system. In this case, you need to go back to the design phase, add the requirement, and redesign the windshield.

What got you in this mess was an unstated and unexamined *assumption,* a classic bug-a-boo of systems engineers. The assumption was that the sensor would work with any kind of windshield material. Of course, if you had simply stated that assumption, someone could have said, "Well, not any material!" and identified the missing requirement. This painful experience is a prime example of why it's so important to test early and often.

Testing Early, Testing Often

The sooner you discover a defect, the cheaper it is to fix. As Figure 5-2 shows, defects discovered after the product is released can cost nearly 100 times more to fix than defects found during the requirements process. But this figure is just one example. By sticking to a rigorous testing process, you can drastically reduce the cost of fixing defects.

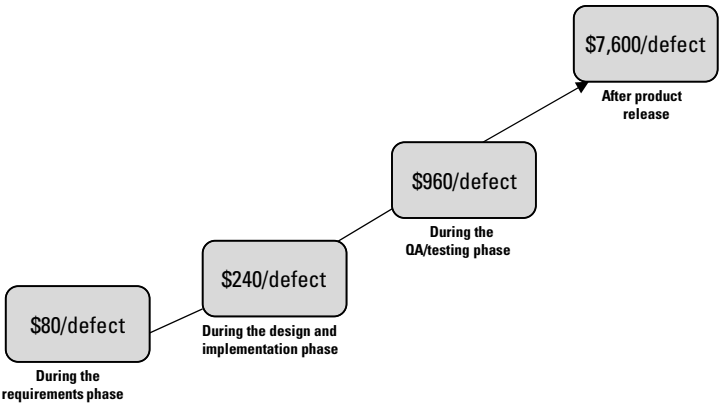

Figure 5-2: The cost of correcting defects increases dramatically throughout the development process.

The Devil Is in the . . . Interfaces

One of the biggest problems in developing and testing large, complex systems is that pieces developed independently don't always work together as planned when integrated together. If you wait until all the pieces are done *before* you integrate and test, and then find some integration problems, you're sunk — because you've probably already burned through most of your schedule and budget.

This factor is probably the largest single factor that causes schedule and cost overruns: finding integration problems late in the development process. It's a *big* risk factor to program success.

So, what can you do about it? Systems engineers have devised two main approaches to minimizing this risk:

- Verify interfaces and interactions between key subsystems and components *early* by substituting models and simulations for some (or even all) subsystems and components
- Integrate parts progressively (or iteratively) and see how they work, instead of waiting until everything is done to start integrating and finding problems

The first approach, modeling the system (see Chapter 4), can be a great help here. Usage models, expressed as activity diagrams and sequence diagrams, help you understand what pieces of the system talk to each other and what interfaces are required between them. With that knowledge, you can bring pieces that "talk" to each other to an early integration — reducing the amount of simulation and emulation required. Forcing the cooperating subsystem teams to work with each other as soon as possible surfaces issues much earlier in the design process — when they're more easily and less expensively fixed.

Models also enable you to test your understanding of the interactions between subsystems before you build — by executing the models and verifying that what you thought was going to happen actually happens. Translating thought to diagram to execution can uncover discrepancies at a time when it is easier and cheaper to fix problems.

With models that simulate the operation of components that haven't been built yet, you can test interfaces long before you commit the entire system to steel and circuitry. Think of something like a flight simulator that lets engineers test aspects of an aircraft's operation long before it leaves the ground.

The second approach is to incorporate as much progressive integration and test (iteration) into the development lifecycle as possible. Software engineers have been using iterative development lifecycles for decades. Of course, it's easier to use iteration in software development than in, say, electronics or airframe construction (because hardware typically demands a longer lead time), but you can apply some of the same ideas.

Early iterative cycles can address high-risk parts of the product by integrating a combination of prototype or simulated hardware with newly development software. With these preliminary, but executable, system builds, you can solve integration problems, perfect interfaces, and validate your trade-study choices.

Some forward-thinking systems engineers suggest that you even try integrating first, and then test the individual components as part of the integrated subsystem. While this sounds counterintuitive, it actually addresses risks earlier than waiting until system integration for things to break.

Chapter 6

Enabling Large Teams to Collaborate and Manage Changes

In This Chapter

- Seeking common ground for effective human interactions
- Automating work processes
- Linking tools and lifecycle data

Small, intimate product development teams really know how to collaborate effectively: they share critical information, use the same development tools, and notify each other whenever they change a requirement, discover a defect, or throw a party!

Multiply the size of the team by a factor of oh, say, 100, give them a wish list of 700ish requirements, tell them they have six months to build the product — and you can kiss your job (and all party invitations) goodbye.

How can you capture the magic that exists for small development teams and adapt it for large development teams? This chapter will show you how.

Learning from Facebook

There's a lot to be learned by exploring the wildly successful social networking site Facebook. Introduced in 2004 when most computer owners already had an email account,

Facebook took off immediately, and, as of early 2011, boasts 600 million users. So what is it about Facebook that makes it so popular?

The founders of Facebook figured out a way to develop a user-friendly, real-time platform for social networking that is organized around typical social interactions with friends, families, and other people with similar interests. It provides a user-friendly interface for sharing news, photos, likes, dislikes, relationship status, and other information. Instead of forcing people to communicate in a technology-driven format, as does email, Facebook uses technology as a platform to support people-driven communication.

Now, imagine applying the "Facebook paradigm shift" to the world of systems engineering. Imagine a technology-based platform that capitalizes on the way development teams, engineers, and stakeholders interact. Like Facebook, such a platform would leverage Internet connectivity to bring people closer together — making a large, widely dispersed team as effective as a small one working together in the same room.

Getting Everyone on the Same Page

It takes a heck of a lot more than just brilliant engineering to create a smart product that is successful in the marketplace. Research shows that a third of all produced devices do not meet performance or functionality requirements, and that 24 percent of all projects are canceled due to unrecoverable schedule delays. Many times, the reason for a catastrophic system failure is not related to the system's engineering design; rather, it is due to failures of knowledge or communication.

It's no wonder large systems development efforts suffer from poor communications. Most large development teams are widely dispersed across cities, companies, and countries. Language and cultural barriers make communication difficult, and time differences often hamper collaboration. Even for employees within the same company, organizational silos can impede communication, reduce productivity, and propagate the "blame game."

Poor communication can cause many problems, including:

- Lack of clarity of system goals
- Multiple interpretations of system requirements
- Incomplete or overlooked requirements
- Time wasted gathering information manually from multiple sources
- Teams working with outdated documents
- Gaps or redundancies in responsibilities

To add to these problems, development teams are under enormous pressure to increase productivity — even as system complexity increases. What's more, smart products with loads of software require more documentation, and the learning curve for new team members is steep.

The best way to overcome the communications difficulties inherent in a large team environment is to provide a common foundation for development and maintenance and to establish a common language for communicating across that foundation.

Laying the foundation

Many of today's smart systems contain multiple subsystems from various sources and millions of lines of code — developed by engineering teams from multiple companies, countries, and cultures. Think of a modern aircraft — the airframe is built in one country by one company, the engine by another, avionics by a third, and integration software by yet another! To streamline the development and testing processes, it's essential to provide a unified systems development platform.

Using a common development platform breaks down barriers between teams, enabling engineers to work together throughout the development lifecycle. A unified platform makes it easier for distributed teams to integrate their work and share knowledge, shaving precious time off the development cycle. By reducing miscommunication and streamlining workflows, you can expect to see substantial improvements in quality — and higher team satisfaction.

What's more, engineers can share project status with everyone on the team through management dashboards, making it easier for development managers to keep the project on track.

Speaking the same language

There's nothing more effective in uniting diverse development teams spanning multiple cultures and engineering disciplines than adopting a model-driven design approach based on a common, domain-independent language. By providing a visual reference for system design, modeling breaks down language barriers, making it easier for everyone on the development team to understand the system and share cross-functional knowledge. And a shared understanding translates directly into productivity improvements, since engineers are no longer wasting time resolving misunderstandings and reworking designs.

The Systems Modeling Language (SysML) is emerging as the accepted standard for model-driven systems development. It supports all phases of systems development, including requirements specification, system analysis and design, verification, and validation of systems that consist of hardware, software, data, people, and even facilities. It also has the advantage of being completely compatible with the Unified Modeling Language (UML) which makes the movement of models from the systems perspective to the software perspective much easier.

There are many commercially available software tools that support development using SysML, each with its own unique development environment. To facilitate effective collaboration, teams should standardize on a common work platform consisting of the most effective tools available on the market.

Going Beyond E-mail and Document Sharing

Establishing a common development platform is a giant step in the right direction toward facilitating effective teamwork — but

it's not enough. If one member of a development team makes a change (for instance, to a requirement or a system model), and it is not communicated immediately to the rest of the team, the result is chaos.

Tracking changes

When engineering a system, critical information is constantly changing — by design. Systems engineering involves (among other things) iterative design and test processes: You design your system, develop models, test the models, redesign to fix defects, and so forth. So you're constantly updating models, test results, and other information. Requirements can change, too, as market and business needs evolve.

Traditionally, large engineering teams have relied on text-based documentation as the source of all project-related information. File cabinets full of requirements documents, high-level architecture documents, design documents, and other documents often serve as the foundation for all system knowledge. But documents are limiting in that they simply *record* information and don't easily allow you to change the information. If you rely solely on documentation, your documents will be obsolete before they've been approved!

Systems development teams need a consistent, flexible mechanism for capturing critical information and facilitating updates while maintaining integrity.

Communicating changes

The traditional method of creating a few dozen critical documents and exchanging information via email just doesn't work well in the complex development environments of today. With the number of requirements for complex systems in the hundreds, or even thousands, relying on e-mail exchange only results in chaos and confusion.

Systems development teams need a vehicle that facilitates effective exchanges among distributed team members. A robust platform for sharing information is the only way to avoid the problems that result when multiple versions of critical documents are floating around all over the place.

Deciding What's Important to Share

If you had to sit down and make a laundry list of all the different types of critical information that must be managed in order to develop a complex system, you'd probably never get your laundry done. So when you're deciding what information should be exchanged with everyone on your development team, be careful to avoid listing every single piece of data describing your system.

Your goal is to facilitate sharing and collaboration among members of a systems development team, not to overwhelm them with extraneous information. So make a list of the minimal information that should be shared in order to facilitate collaboration, such as

- Development priorities
- Project approvals
- Schedules
- Employee roles and responsibilities
- Requirements
- Change requests
- Conceptual models
- Use cases
- Test plans
- Defects
- Critical issues
- Traceability data
- Budget information
- Procurement information

Exploring Ways to Facilitate Sharing

It's unrealistic to expect each of the potentially hundreds of companies participating in the product development process to use the exact same set of tools from the same tool vendor. Therefore, there needs to be a way for everyone on the virtual team to be able to share development data — no matter which team or partner created it.

Effective collaboration begins with a platform on which the overall development process can be managed and tracked. Using this platform, stakeholders and engineers process and produce data that is shared, analyzed, and reported on efficiently and effectively throughout the systems development lifecycle.

Automation and federation tools that leverage technology to streamline communication and automate workflows are the next major development in collaboration.

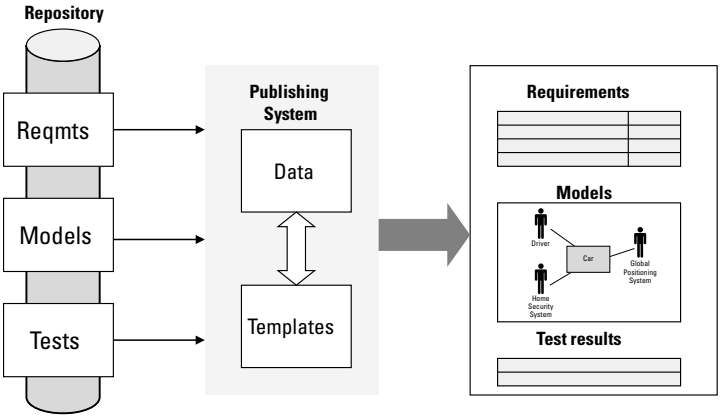

Figure 6-1: A virtual repository of design information can help improve collaboration.

Creating a virtual repository of live information

Individuals need real-time access to up-to-date information about the project, no matter where they are located and no matter what the format of the data. Traditionally, systems engineers dedicate a significant amount of time keeping track of information and making sure that everyone involved is working from the same documents.

By organizing critical information in a virtual repository, you create a single source of knowledge about critical aspects of the development process and can automate the process of information sharing.

A *virtual* repository doesn't physically exist; rather, data is pulled as necessary from globally distributed locations on an as needed basis. For instance, mechanical engineering data may be pulled from a physical data repository managed by the mechanical engineering team in Seattle, whereas electrical engineering data may be pulled from an electrical engineering database located in Tokyo. Individuals need not know (or care) where the data physically resides — as long as they get what they need when they need it.

Figure 6-1 shows a conceptual model for such a virtual repository and how it can be used. A business process manager acts as the brains behind the operation, managing the execution of the development activities. It can access data as necessary depending on what it needs (and when it needs it) by using open standards. Instead of handing down text-based documents from product managers to architects to design engineers to test engineers, and so forth, all of the people involved in the development process can access the data they need transparently.

A virtual repository such as this provides a single source of up-to-date information that everyone can access wherever they are located. What's more, people can exchange information, share ideas, and capture decisions and thought processes — almost as if they worked in the same office together.

Why a "virtual" repository?

Why a "virtual" repository and not a "real" one you ask? Well, there are many reasons why it may be a better idea than "jam everything in one database."

- First, if you try to define a "universal database" that would have to know how to store everything you might ever want or need, it would be a big job!
- Second, you would have to be able to support every vendor's tool that currently or may exist in the future.
- And third, one massive database that is accessed from all over the planet will have good performance for some and very lousy for others, not to mention issues about upgrading to new versions, single point failures (of network or hardware), and many other challenges of a single physical data store.

Virtual is definitely better — distributed, optimized for each tool's data and performance, easily upgraded, and easy to connect to others.

Warning: Going with a "put everything in one place" solution may lock you into a proprietary solution and one vendor, removing your choice as a consumer!

Simplifying tool integration

While sharing resources and assets via a virtual repository sounds great on the surface, in reality, it's not that easy. In-house tools, open-source projects, and multiple vendors can create barriers due to incompatible data formats and other factors.

Over the past few years, vendors of popular system lifecycle development tools, such as the types of tools listed in Table 6-1, have come together to define ways to facilitate the integration of systems lifecycle tools. Forming a community known as the Open Services for Lifecycle Collaboration (OSLC), these companies are committed to promoting new forms of collaboration by eliminating barriers between tools.

Inspired by the Internet architecture, OSLC specifies a set of loosely-coupled standards, common resource formats, and services designed to facilitate the sharing of system resources. OSLC makes it easier to use tools from any vendor in combination, while easily sharing system lifecycle data, such as requirements, change requests, test cases, and defects.

Table 6-1 Key Systems Engineering Tools

Type of Tool	*Key Capabilities*
Requirements Management and Traceability	End-to-end live traceability to source, mission, and system/subsystem requirements
Model-based Systems Development	Model requirements, system functionality, realization, trade studies, execution, and validation
Change and Configuration Management	Manage collaboration, change, shared repository, and configuration
Automated Documentation Generation	Generate requirements, design, and specification documents
Integrated Systems and Software Engineering	Flow-down of requirements and models, embedded development

Automating document production

Even in a model-centric development environment, documentation and reports are required for contractual obligations, compliance, technical reviews, and project management. Documentation often involves a lot of manual labor, such as taking a diagram out of a modeling tool, doing a screen capture, and pasting it into a document. What's more, engineers spend a lot of time collecting information from different sources, ensuring they have the latest information, summarizing and reformatting information in order to produce a customized report.

Document automation tools can streamline the production of custom reports, while ensuring consistency of information within each report.

Automation tools enable you to access a central repository of information, identify the types of information you need, select it, and request a report. Not only is this process simple, but also it facilitates reuse and consistency of information. What's more, you can consolidate critical information from multiple sources — even multiple vendor's products — into a single report.

Surely, this saves time producing the documents in the first place, but the big win is when something changes. Make the necessary changes in the requirements and models, press a button, and presto — out come the revised documents.

Chapter 7

Ten Ways to Win with Systems Engineering

In This Chapter

- Fixing design flaws early in the development process
- Developing flexible business models
- Gaining control of complex embedded software
- Improving requirements management
- Reusing code to accelerate product launches
- Using collaboration tools
- Delivering complex solutions to market on time
- Using an integrated development platform
- Test early, test often
- Sharing requirements to increase efficiency

Systems engineering can give you the competitive edge you need to succeed in developing smart products that deliver tangible value to your customers (and your bottom line). By making systems engineering a part of your core business processes, you're more likely to produce the products people really want with fewer costly defects while enhancing your ability to respond to market dynamics.

In this chapter, you look at ten examples of companies that adopted best practices in systems engineering — and got real, measurable results.

Fixing Design Flaws before They Go Viral

One of the biggest challenges in engineering smart products is identifying defects earlier in the development process. Most defects are introduced during the design process, but not detected until the testing phase or, worse yet, after the product has made it to production.

For expensive, mass-produced products, letting design flaws slip through the cracks can be enormously costly, but for mass-produced defense systems, overlooked defects can also be dangerous. To complicate matters further, most defense systems are composed of multiple complex subsystems designed and built by an array of approved subcontractors.

As a subcontractor to the United States Department of Defense, Brockwell Technologies based in Huntsville, Alabama, creates real-time embedded weapon systems applications and performs embedded diagnostics for military vehicles. Developing interconnected weapons systems presents a unique challenge: how to make changes to one system without affecting other systems.

To reduce the possibility of introducing defects in interactions between subsystems, Brockwell Technologies incorporated model-based system design and testing into its business processes. By modeling both system structure and behavior, Brockwell engineers can prototype complex systems and visualize how well they will function. This predictive modeling technique enables engineers to identify design flaws early in the development process — before the systems are mass-produced.

Brockwell's investment in systems engineering is a win-win for the Department of Defense and for Brockwell: Brockwell improved the reliability and safety of its weapons systems while decreasing time-to-market by 40 percent.

Engineering a Flexible Business Model

If you're planning to start a new business, or enter a new market, the smartest thing you can do is develop a flexible business infrastructure designed to be the poster child of systems engineering. That's exactly what Atlanta-based Hughes Telematics, Inc. (HTI) did when it entered the red-hot telematics market several years ago.

HTI saw an opportunity to outrun the veterans in the industry by setting up its business infrastructure in a way that would allow it to adapt business relationships and service delivery models quickly. With a flexible process framework built on open technology, HTI reasoned it would be able to jump on new service opportunities before the existing telematics providers — with their proprietary technology and rigid business models — could react.

HTI designed all of its core business processes — including front office, back office, and operations — with one goal in mind: launching new services quickly. What's more, to streamline the development of new services, HTI installed a common systems development platform and a host of collaboration tools. HTI's systems development infrastructure enabled it to manage work products and deliverables, store deliverables in a central repository, facilitate collaboration among globally distributed teams, maintain version control, and rapidly communicate updates.

With a combination of open systems, flexible processes, and enhanced collaboration, HTI is able to bring new services to market in less than 30 days. As the focal point of telematics continues to shift from the technology itself to innovative services that use vehicle data in new ways, HTI is, indeed, well-positioned to win the race.

Gaining Control of Complex Embedded Software

Smart companies know that when their product is just one part of a larger system-of-systems solution, ensuring consistent

product quality is of the utmost importance. After all, you don't want your company to be branded as the "weak link" in the system. But as the size and complexity of embedded software grows, it becomes more and more difficult to ensure quality.

For a long time, manual software development was a way of life for a German technology service provider that specializes in measurement and control technology and process engineering. But when the company set out to develop complex embedded software for systems that remotely manage and control photovoltaic systems, it realized that it had to adopt a new software environment.

With four specific goals in mind — decrease product defects, improve traceability, increase re-use of software modules, and ensure consistent product quality — the company selected a platform that uses model-driven development for real-time and embedded systems engineering. The new system enables the company to identify and repair problems earlier by testing models during the design phase — ensuring consistent, high quality software. As a bonus, the company can now create reusable source code modules and subsystems, giving the company an edge on the competition.

Improving Efficiency with Requirements Consistency

To build the right product — and build the product right — you have to have a solid understanding of customer and stakeholder needs, and carefully define your system requirements around those needs. Without effective requirements management, you can easily lose sight of your objectives — and your customers.

With hundreds of geographically dispersed developers all using different requirements management tools, one Australian company was struggling with development inefficiencies and inadequate requirements traceability. Management was becoming increasingly concerned about the fact that they couldn't test the requirements in a uniform way, since each development team had its own way of managing requirements. Worst of all, the lack of consistency increased the opportunity for error and had the potential to jeopardize

the company's relationship with its biggest customer, the Australian Defense Force.

The organization deployed a unified requirements management solution designed to support requirements analysis throughout the company. By providing ubiquitous access to a centralized requirements repository, the solution eliminates confusion, costly rework, and duplication of effort. Most importantly, the solution facilitates end-to-end requirements traceability — ensuring that the product rolling out the door fulfills the customer's needs.

Leveraging Reusable Code to Accelerate Product Launch

If your portfolio includes a line of products with similar core capabilities, you've probably got a lot of similar software code floating around. Smart companies structure their code into reusable modules, so they can expedite the development of future products.

That's exactly what Océ N.V. had in mind when it set out to produce the world's fastest cut sheet printer. A market leader in digital document management technology and services, Océ develops advanced software applications that deliver documents and data over internal networks and the Internet to printing devices and archives locally and throughout the world.

Normally, Océ codes each new printer from scratch, but when faced with the enormous task of coordinating code for 17 processors distributed throughout the new printer, the company decided to re-examine its development processes.

Océ deployed model-driven development tools to decompose the printer system into smaller, more easily designed subsystems. This enabled its developers to model a set of concurrent finite state machines, which were then coded into a set of reusable modules for multiple target environments. Now, whenever a change is needed, Océ simply updates the models and regenerates the code — slashing the turnaround time for change requests.

Today, over 50 percent of the company's software code is reusable, leading to enormous improvements in efficiency and quality. In fact, Océ was even able to launch a working prototype of a new printer in just two months — six months sooner than it previously could.

Streamlining Development with Collaboration Tools

Companies that develop complex, intelligent, highly instrumented products know that success depends as much on managing the technical work as it does on superior engineering. Without a system-level discipline coordinating the efforts of diverse engineering groups, complex products are destined to become newsworthy failures.

When General Motors (GM) set out to design the Chevy Volt, it put a tremendous effort into establishing best practices for systems engineering. GM examined both its development practices and its technical work management practices with an eye to improving both.

Traditionally, GM systems engineers spent the majority of their time checking on developers' workloads and making sure that everyone had the same version of requirements and other documents. To free up the engineers to focus more on functionality and quality, GM decided to deploy a commercial tool to coordinate development teams and manage a single version of requirements and other documentation.

GM then deployed model-driven systems development practices to help manage complexity (the Volt runs on about 10 million lines of code). GM developers used models to visualize the interactions between embedded systems and ran simulations in order to test the models.

The combination of collaboration tools and model-driven systems development really paid off: GM completed the Volt's development in just 29 months — a record for GM, where new car development typically takes five years or more.

Delivering Complex Solutions to Market on Time

To succeed in today's ultra-competitive environment, companies that develop extremely large, complex systems know they need to get smart about focusing their development process on requirements. The ability to capture and manage requirements can mean the difference between winning and losing a big contract.

A defense company wanted to reduce the risks involved in delivering complex, multimillion dollar system-of-systems solutions to market on time. The company devised an innovative idea: bring together requirements management and enterprise architecture frameworks in a way that enables the company to deliver complex, requirements-driven systems on schedule.

The company based its solution on commercially available products for requirements management and system architecture planning. Now, the company has the ability to define not just technology solutions, but operational, technical, training, and systems solutions — across the complete lifecycle. As a result, it is able to deliver high-quality, large, complex systems to market faster than ever. What's more, when customer requirements change, the company can show the impact of those changes very quickly.

Improving Productivity with an Integrated Development Platform

Companies that develop safety-critical systems are often required to demonstrate that their development process meets national and international safety standards. If their development environment is fragmented, it's difficult for them to demonstrate compliance.

With a diverse set of development tools and development teams spanning two major sites, another Australian company needed to make some changes in order to ensure the success of future projects. It provides safety-critical rail signaling and control solutions around the world, and its customers require backward compatibility with their existing rail equipment.

The company needed an integrated development environment that promoted compliance with global safety and reliability standards. The company deployed a uniform requirements and configuration management solution that not only fulfilled customer requirements, but also enabled its dispersed development teams to collaborate seamlessly.

Test Early, Test Often

Getting a prototype or executable model into the hands of stakeholders early and often is incredibly valuable. Being able to get early feedback on your concepts saves you pain later.

Being able to run simple tests as early as possible also pays off during final testing. Giving early access to the testing teams, even if they know the product is not really ready or is just a prototype also helps them improve their test plans and procedures in the same way it helps the designers check out their ideas.

Sharing Requirements to Save Time and Money

It's common for large development organizations to have two or more geographically dispersed development teams working on parallel releases with shared requirements. A lot of time and effort could be saved if these teams had a mechanism for sharing requirements.

A Michigan-based company is a leading global supplier of mobile electronics and transportation systems. One of the company's goals was to improve communication among its global development teams so they would be more productive as they worked on parallel releases with shared requirements.

Implementing a requirements management tool made it easy for the company's developers to share requirements. They can import requirements from a repository to re-use in a new project, avoiding unnecessary duplication of effort, saving time, and reducing development costs. Because the requirements management tool promotes consistency and enhances collaboration, the company is able to deliver higher quality products in a shorter amount of time.