Jayson, Seth, "The Perils of Prosecuting the Penguin," *Business 2.0*, www.business2.com, April 26, 2004.

Koch, Richard, "Open Source Reality Check," *CIO*, www.cio.com, December 28, 2004.

McMillian, Robert, "SCO Shows Disputed Code to IBM," *Computerworld*, January 14, 2004.

Milliard, Elizabeth, "SCO Dented by Dismissal of Daimler Chrysler Suit," *LinuxInsider*, July 7, 2004.

Weiss, Todd R., "Defiant IBM Calls UNIX Indemnification Unnecessary," *Computerworld*, January 26, 2004.

Sources for Case 3

Brandel, William, "'Look and Feel' Reversal Re-ignites Copyright Fight," *Computerworld*, March 13, 1995.

Goetz, Martin A., "Copycats or Criminals?" *Computerworld*, June 12, 1995.

"Borland Prevails in Lotus Copyright Suit," Borland press release, www.borland.com, January 16, 1996.

CHAPTER 7

SOFTWARE DEVELOPMENT

QUOTE

Buying the right computer and getting it to work properly is no more complicated than building a nuclear reactor from wristwatch parts in a darkened room using only your teeth.
—Dave Barry, U.S. humorist

VIGNETTE

Antivirus Bug Brings Computers to a Standstill

At 7:30 a.m. Tokyo time on Saturday, April 23, 2005, Trend Micro released antivirus update Official Pattern Release (OPR) 2.594.00. At 9:02 a.m., it was pulled off Active Update. During those 92 minutes, hundreds of thousands of computer processors across the world came grinding to a halt.

The update referenced a data file that identified known viruses. This file contained a bug, a loop that gradually sapped nearly 100 percent of the processing power of PCs in Japan, Europe, and the Middle East. By 2 p.m. on Saturday, more than 300,000 Trend Micro customers had called to report problems. "The problem was caused by our virus labs in Manila during the checking process when a part of the test wasn't performed," explained Trend Micro spokeswoman Naomi Ikenomoto.

The company created the update to handle Rbot, a prolific worm that gained notoriety when an attack on the Italian senate caused computer monitors there to display pornographic photographs. The worm allowed hackers to steal information, and some variants of Rbot observed computer users via their own Webcams.

In the rush to protect customers from Rbot, the company failed to test the update on PCs that use Windows XP Service Pack 2, and possibly a few other systems. The company cited human error and responded by channeling more resources into quality assurance.

In the meantime, the mistake cost Trend Micro $8 million. The company lost an estimated 700 business licenses and 28,300 customer licenses. The mistake also caused the company to reduce its second-quarter revenue forecast by 16.7 percent. However, because corporations cannot quickly shift antivirus vendors, much of the impact was likely to be felt in 2006.[1,2,3]

LEARNING OBJECTIVES

As you read this chapter, consider the following questions:

1. Why do companies require high-quality software in business systems, industrial process control systems, and consumer products?
2. What ethical issues do software manufacturers face in making trade-offs between project schedules, project costs, and software quality?
3. What are the four most common types of software product liability claims, and what actions must plaintiffs and defendants take to be successful?
4. What are the essential components of a software development methodology, and what are its benefits?
5. How can Capability Maturity Model Integration improve an organization's software development process?
6. What is a safety-critical system, and what actions are required during its development?

STRATEGIES TO ENGINEER QUALITY SOFTWARE

High-quality software systems are easy to learn and easy to use. They perform quickly and efficiently to meet their users' needs. They operate safely and dependably and have a high degree of availability so that system downtime is kept to a minimum. Such software has long been required to support the fields of air traffic control, nuclear power, automobile safety, health care, military and defense, and space exploration. Now that computers and software have become an integral part of our lives, more and more users are demanding high quality in their software. They cannot afford system crashes, lost work, or lower productivity, nor can they tolerate security holes through which intruders can spread viruses, steal data, or shut down Web sites. Software manufacturers are struggling with economic, ethical, and organizational issues associated with improving the quality of their software. This chapter covers many of these issues.

A **software defect** is any error that, if not removed, could cause a software system to fail to meet its users' needs. The impact of these defects can be trivial; for example, a computerized sensor in a refrigerator's ice cube maker might fail to recognize that the tray is full and continue to make ice. Other defects could lead to tragedy—the control system for an automobile's antilock brakes could malfunction and send the car into an uncontrollable spin. The defect may be subtle and undetectable, such as a tax preparation package that makes a minor miscalculation, or the defect might be glaringly obvious, such as a payroll program that generates checks with no deductions for Social Security or other taxes.

For example, the following defects were uncovered in widely used software packages in the fall of 2005:

- Apple Computer had to create patches for 10 vulnerabilities in its Mac OS X operating systems 10.3.9 (dubbed Panther) and 10.4.2 (Tiger). Some of the bugs were serious enough that the Danish security company Secunia labeled the fixes as "highly critical." In the previous month, Apple set a firm record with patches for more than 40 bugs.[4]
- Versions of RealPlayer and Helix Player that work with the Linux operating system had to be patched against a vulnerability that enabled a hacker to execute commands remotely, provided he could convince a user to open certain files. The exploit code was even published on the Internet, leading some security firms to label the bug as critical.[5]
- Bugs in the Kaspersky Labs' Anti-Virus engine could be exploited by attackers to grab complete control of a PC protected by the company's Windows products. The Moscow-based security vendor had to implement a stopgap measure, building and releasing a package of signatures that detect possible exploits.[6]
- As of October 2005, Internet Explorer (IE) had six unpatched vulnerabilities, according to eEye Digital Security, which tracks the bugs it has submitted to Microsoft. During the same period, Danish vulnerability tracker Secunia said that IE 6.0 was afflicted with a dozen unpatched bugs.[7]

- Users of Symantec Corporation's AntiVirus Scan Engine versions 4.0 and 4.3 were advised to upgrade their software to eliminate a critical security bug. The flaw could theoretically allow an attacker to take control of an affected system.[8]
- According to security vendor Finjan Software, a bug in Google could have allowed attackers to grab a Google user's cookie. If the user was logged on to his Google account, the stolen cookie would have let the attacker view the user's saved searches or alerts, or even assume the user's identity in Google Groups.[9]

Sometimes, there are even errors in the patches implemented to correct bugs. For example, problems were reported with a Microsoft security update that was designed to fix a critical flaw in Windows. Under certain conditions, security update MS05-051 caused numerous problems, including blocking access to the Microsoft update Web site, displaying a blank logon screen without icons, and creating issues with Office applications.[10]

Software quality is the degree to which a software product meets the needs of its users. **Quality management** addresses how to define, measure, and refine the quality of the development process and the products developed during its various stages. These products, such as statements of requirements, flowcharts, and user documentation, are known as **deliverables**. The objective of quality management is to help developers deliver high-quality systems that meet the needs of their users. Unfortunately, the first release of any software rarely meets all its users' expectations. A software product usually does not work as well as its users would like until it has been used for a while, been found lacking in some ways, and then corrected or upgraded.

A primary cause for poor software quality is that developers do not know how to design quality into software from the very start, or do not take the time to do so. Software developers must define and follow a set of rigorous engineering principles and be committed to learn from past mistakes. In addition, they must understand the environment in which their systems will operate and design systems that are as immune to human error as possible.

Even if software is well-designed, programmers make mistakes in turning design specifications into lines of code. According to estimates, an experienced programmer unknowingly injects about one defect into every 10 lines of code. The programmers aren't incompetent or lazy—they're just human. Everyone makes mistakes, but in software, these mistakes result in defects.[11] For example, the Microsoft Windows XT operating system is composed of four hundred million lines of code bundled with products made by other Microsoft divisions. Even if 99.9 percent of the defects were identified and fixed before the product was released to the public, there would still be about one bug per 10,000 lines of code, or roughly 40,000 bugs in Windows XT. Thus, software that is used daily by workers worldwide probably contains thousands of bugs.

Another factor that causes poor-quality software is the extreme pressure that manufacturers feel to reduce the time to market of their products. They are driven by the need to beat the competition in delivering new functionality to users, to begin generating revenue to recover the cost of development, and to show a profit for shareholders. The resources and time budgeted to ensure quality are often cut under the intense pressure to ship the new product. When forced to choose between adding more user features or doing more testing, most software development managers decide in favor of more features. After

all, they reason, defects can always be patched in the next release, which will give customers an automatic incentive to upgrade. The additional features will make the release more useful and therefore easier to sell to customers. Although customers are stakeholders who are key to the software's success, and they may benefit from new features, they also bear the burden of errors that aren't caught or fixed during testing. Thus, many customers challenge whether the decision to cut quality in favor of feature enhancement is ethical.

As a result of the lack of consistent quality in software, many organizations avoid buying the first release of a major software product or prohibit its use in critical systems. Their rationale is that the first release usually has many defects that cause user problems. Because of the many defects in the first two popular Microsoft operating systems (DOS and Windows) and their tendency to crash unexpectedly, many believe that Microsoft did not have a reasonably reliable operating system until its third major variation—Windows NT. Even software products that have been reliable over a long period can falter unexpectedly when the operating conditions change. For instance, the software in the Cincinnati Bell telephone switch had been thoroughly tested and had operated successfully for months after it was deployed in 1985. Later that same year, however, when the time changed from daylight saving time to standard time, the switch failed because it was overwhelmed by the number of calls to the local "official time" phone number from people who wanted to set their clocks. The volume of simultaneous calls to the same number was a change in operating conditions that no one had anticipated.

The Importance of Software Quality

Most people think of business information systems when they first think about software. **Business information systems** are a set of interrelated components that include hardware, software, databases, networks, people, and procedures that collect data, process it, and disseminate the output. One common business system captures and records business transactions. For example, a manufacturer's order-processing system captures order information, processes it to update inventory and accounts receivable, and ensures that the order is filled and shipped on time to the customer. Other examples include an electronic-funds transfer system that moves money among banks and an airline's online ticket reservation system. The accurate, thorough, and timely processing of business transactions is a key requirement for such systems. A software defect can be devastating, resulting in lost customers and reduced revenue. How many times would bank customers tolerate having their funds transferred to the wrong account before they stopped doing business with that bank?

Another type of business system is the **decision support system (DSS)**, which is used to improve decision making. A DSS can help develop accurate forecasts of customer demand, recommend stocks and bonds for an investment portfolio, and schedule shift workers to minimize cost while meeting customer service goals. Again, a software defect in a DSS can be devastating to an organization or its customers.

Software is also used to control many industrial processes in an effort to reduce costs, eliminate human error, improve quality, and shorten the time it takes to make products. For example, steel manufacturers use process-control software to capture data from sensors about the equipment that rolls steel into bars and about the furnace that heats the steel before it is rolled. Without process-control computers, workers could react to defects only after the fact and would have to guess at the adjustments needed to correct the

process. Process-control computers enable the process to be monitored for variations from operating standards and to eliminate product defects *before* they can be made. Any defect in this software can lead to decreased product quality, increased waste and costs, or even unsafe operating conditions for employees. (Some consequences of these defects are discussed later in this chapter.)

Software is also used to control the operation of many industrial and consumer products—automobiles, medical diagnostic and treatment equipment, televisions, radios, stereos, refrigerators, and washers. A software defect can have relatively minor consequences, such as clothes not drying long enough, or it can lead to much more serious damage, such as a patient being overexposed to powerful X-rays.

As a result of the increasing use of computers and software in business, many companies are now in the software business whether they like it or not. The quality of software, its usability, and its timely development are critical to almost everything businesses do. The speed with which an organization develops software (whether in-house or contracted out) can put it ahead of or behind its competitors. Software problems may have been frustrating in the past, but mismanaged software can now be fatal to a business, causing it to miss product delivery dates, increase product development costs, and deliver products that have poor quality.

Business executives must struggle with ethical questions of how much effort and money they should invest to ensure high-quality software. A manager who takes a short-term, profit-oriented view may feel that any additional time and money spent on quality assurance delays a new product's release, sales revenues, and profits. However, a different manager may consider it unethical not to fix all known problems before putting a product on the market and charging customers for it.

Other key questions for executives are whether their products could cause damage and what their legal exposure would be if they did. Fortunately, software defects are rarely lethal and few personal injuries are related to software failures. However, the use of software does introduce a new dimension to product liability that concerns many executives, as explained in the following Legal Overview.

LEGAL OVERVIEW

Software Product Liability

Software product litigation is certainly not new. One lawsuit in the early 1990s involved a financial institution that became insolvent because defects in a purchased software application caused errors in its integrated general ledger system, in customers' passbooks, and in loan statements. Dissatisfied depositors responded by withdrawing more than $5 million.[12] In a 1992 case involving an automobile manufacturer, a truck stalled because of a software defect in the fuel injector. In the ensuing accident, a young child was killed, and a state supreme court later justified an award of $7.5 million in punitive damages against the manufacturer.[13]

continued

The liability of manufacturers, sellers, lessors, and others for injuries caused by defective products is commonly referred to as **product liability**. There is no federal product liability law; instead, it is found mainly in common law (made by state judges) and in Article 2 of the Uniform Commercial Code, which deals with the sale of goods.

If a software defect causes injury to purchasers, lessees, or users of the product, the injured parties may be able to sue as a result. Injuries range from physical mishaps and death to loss of revenue or an increase in expenses due to a business disruption caused by a software failure. Software product liability claims are frequently based on strict liability, negligence, breach of warranty, or misrepresentation.

Strict liability means that the defendant is held responsible for injuring another person, regardless of negligence or intent. The plaintiff must prove only that the software product is defective or unreasonably dangerous and that the defect caused the injury. There is no requirement to prove that the manufacturer was careless or negligent, or to prove who caused the defect. All parties in the chain of distribution—the manufacturer, subcontractors, and distributors—are strictly liable for injuries caused by the product and may be sued.

Defendants against a strict liability action may use several legal defenses, including the doctrine of supervening event, the government contractor defense, and an expired statute of limitations. Under the doctrine of supervening event, the original seller is not liable if the software was materially altered after it left the seller's possession and the alteration caused the injury. To establish the government contractor defense, a contractor must prove that the precise software specifications were provided by the government, the software conformed to the specifications, and the contractor warned the government of any known defects in the software. Finally, there are also statutes of limitations for claims of liability, which means that an injured party must file suit within a certain time after the injury occurs.

When sued for **negligence**, a software supplier is not held responsible for every product defect that causes customer or third-party loss. Instead, responsibility is limited to harmful defects that could have been detected and corrected through "reasonable" software development practices. Even when a contract may be written expressly to protect against supplier negligence, courts may disregard such terms as unreasonable. Negligence is an area of great risk for software manufacturers or organizations with software-intensive products.

The defendant in a negligence case may either answer the charge with a legal justification for the alleged misconduct or demonstrate that the plaintiffs' own actions contributed to their injuries (**contributory negligence**). If proved, the defense of contributory negligence can reduce or totally eliminate the amount of damages the plaintiffs receive. For example, if a person uses a pair of pruning shears to trim his fingernails and ends up cutting off a fingertip, the defendant could claim contributory negligence.

A **warranty** assures buyers or lessees that a product meets certain standards of quality. A warranty of quality may be either expressly stated or implied by law. Express warranties can

continued

be oral, written, or inferred from the seller's conduct. For example, sales contracts contain an implied warranty of merchantability, which requires that the following standards be met:

- The goods must be fit for the ordinary purpose for which they are used.
- The goods must be adequately contained, packaged, and labeled.
- The goods must be of an even kind, quality, and quantity within each unit.
- The goods must conform to any promise or affirmation of fact made on the container or label.
- The quality of the goods must pass without objection in the trade.
- The goods must meet a fair average or middle range of quality.[14]

If the product fails to meet its warranty, the buyer or lessee can sue for **breach of warranty**. Of course, most dissatisfied customers will first seek a replacement, a substitute product, or a refund before filing a lawsuit.

Software suppliers frequently write warranties to attempt to limit their liability in the event of nonperformance. Although the software is warranted to run on a given machine configuration, no assurance is given as to what that software will do. Even if the contract specifically excludes the commitment of merchantability and fitness for a specific use, the court may find such a disclaimer clause unreasonable and refuse to enforce it or refuse to enforce the entire contract. In determining whether warranty disclaimers are unreasonable, the court attempts to evaluate if the contract was made between two "equals" or between an expert and a novice. The relative education, experience, and bargaining power of the parties and whether the sales contract was offered on a take-it-or-leave-it basis are considered in making this determination.

The plaintiff must have a valid contract that the supplier did not fulfill in order to win a breach of warranty claim. Because the software supplier writes the warranty, this claim can be extremely difficult to prove. For example, in 1993, M.A. Mortenson Company had a new version of bid-preparation software installed for use by its estimators. During the course of preparing one new bid, the software allegedly malfunctioned several times, each time displaying the same cryptic error message. Nevertheless, the estimator submitted the bid and Mortenson won the contract. Afterward, Mortenson discovered that the bid was $1.95 million lower than intended and filed a breach-of-warranty suit against Timberline Software, makers of the bid software. Timberline acknowledged the existence of the bug, but the courts all ruled in favor of Timberline, ruling that the license agreement that came with the software explicitly barred recovery of the losses claimed by Mortenson.[15] Even if breach of warranty can be proven, the damages are generally limited to the amount of money paid for the product.

Intentional misrepresentation occurs when a seller or lessor either misrepresents the quality of a product or conceals a defect in it. For example, a cleaning product might be advertised as safe to use in confined areas, but users pass out from the fumes. A buyer or lessee who is subsequently injured can sue the seller for intentional misrepresentation or fraud. Advertising, salespersons' comments, invoices, and shipping labels are all forms of representation. Most software manufacturers use limited warranties and disclaimers to avoid any claim of misrepresentation.

Software Development Process

Developing information system software is not a simple process. It requires completing many activities that are themselves complex, with many dependencies among the various activities. System analysts, programmers, architects, database specialists, project managers, documentation specialists, trainers, and testers are all involved in large software projects. Each of these groups has a role to play, and has specific responsibilities and tasks. In addition, each group makes decisions that can affect the software's quality and the ability to use it effectively.

Many software developers have adopted a standard, proven work process (or **software development methodology**) that enables systems analysts, programmers, project managers, and others to make controlled and orderly progress in developing high-quality software. A methodology defines activities in the software development process and the individual and group responsibilities for accomplishing these activities. It also recommends specific techniques for accomplishing the various activities, such as using a flowchart to document the logic of a computer program. A methodology also offers guidelines for managing the quality of software during the various stages of development. If an organization has developed such a methodology, it is applied to any software development that the company undertakes.

As with most things in life, it is far safer and cheaper to avoid software problems at the beginning than to attempt to fix the damages after the fact. Studies at IBM, TRW, and GTE have shown that the cost to identify and remove a defect in an early stage of software development can be 100 times less than removing a defect in an operating piece of software that has been distributed to hundreds or thousands of customers.[16] This increase in cost is due to two reasons. First, if a defect is uncovered in a later stage of development, some rework of the deliverables produced in preceding stages is necessary. Second, the later the error is detected, the greater the number of people affected by the error. Consider the cost to communicate the details of a defect, distribute and apply software fixes, and possibly retrain end users for a software product that has been sold to hundreds or thousands of customers. Thus, most software developers try to identify and remove errors early in the development process as a cost-saving measure and as the most efficient way to improve software quality.

Products containing inherent defects that harm the user are the subjects of product liability suits. The use of an effective methodology can protect software manufacturers from legal liability for defective software in two ways. First, an effective methodology reduces the number of software errors that might occur. Second, if an organization follows widely accepted development methods, negligence on its part is harder to prove. However, even a *successful* defense against a product liability case can cost hundreds of thousands of dollars in legal fees. Thus, failure to develop software reasonably and consistently can be quite serious in terms of liability exposure.

Software quality assurance (QA) refers to methods within the development cycle that guarantee reliable operation of the product. Ideally, these methods are applied at each stage throughout the development cycle. However, some software manufacturing organizations that haven't established a formal, standard approach to QA consider testing to be their only QA method. Instead of checking for errors throughout the development process, they rely primarily on testing just before the product ships to ensure some degree of quality.

Several types of tests are used in software development, as discussed in the following sections.

Dynamic Software Testing

Software is developed in units called subroutines or programs. These units, in turn, are combined to form large systems. When a programmer completes a unit of software, one QA measure is to test the code by actually entering test data and comparing the results to the expected results. This is called **dynamic testing**. There are two forms of dynamic testing:

- **Black-box testing** involves viewing the software unit as a device that has expected input and output behaviors but whose internal workings are unknown (a black box). If the unit demonstrates the expected behaviors for all the input data in the test suite, it passes the test. Black-box testing takes place without the tester having any knowledge of the structure or nature of the actual code. For this reason, it is often done by someone other than the person who wrote the code.
- **White-box testing** treats the software unit as a device that has expected input and output behaviors but whose internal workings, unlike the unit in black-box testing, are known. White-box testing involves testing all possible logic paths through the software unit with thorough knowledge of its logic. The test data must be carefully constructed to make each program statement execute at least once. For example, if you wrote a program to calculate an employee's gross pay, you would develop data to test cases in which the employee worked less than 40 hours and other cases in which the employee worked more than 40 hours (to check the calculation of overtime pay).

Other Types of Software Testing

Other forms of testing include the following:

- **Static testing**—Special software programs called static analyzers are run against the new code. Rather than reviewing input and output, the static analyzer looks for suspicious patterns in programs that might indicate a defect.
- **Integration testing**—After successful unit testing, the software units are combined into an integrated subsystem that undergoes rigorous testing to ensure that all the linkages among the various subsystems work successfully.
- **System testing**—After successful integration testing, the various subsystems are combined to test the entire system as a complete entity.
- **User acceptance testing**—This independent testing is performed by trained end users to ensure that the system operates as they expect.

Capability Maturity Model Integration for Software

Capability Maturity Model Integration (CMMI) is a process improvement approach defined by the Software Engineering Institute at Carnegie Mellon University in Pittsburgh. It defines the essential elements of effective processes. The model is general enough to be used to evaluate and improve almost any process, and it is frequently used to assess software development practices. CMMI defines five levels of software development maturity (see Table 7-1) and identifies the issues that are most critical to software quality and process improvement. Identifying an organization's current maturity level enables it to specify actions to improve its future performance. The model also enables an organization to track, evaluate, and demonstrate its progress over the years.

After an organization decides to adopt CMMI, it must conduct an assessment of its software development practices (often using outside resources to ensure objectivity) and determine where they fit in the capability model. The assessment identifies areas for improvement and the action plans needed to upgrade the development process. Over the course of a few years, the organization and software engineers can raise their performances to the next level by executing the action plan.

As the maturity level increases, the organization improves its ability to deliver good software on time and on budget. For example, the Lockheed Martin Missile and Defense Systems converted to CMMI between 1996 and 2002, with outstanding improvements. Lockheed increased software productivity by 30 percent, reduced unit software costs by 20 percent, and cut the costs of finding and fixing software defects by 15 percent.[17]

CMMI can also be used as a benchmark for comparing organizations. In the awarding of software contracts, particularly with the government, organizations that bid on the contract may be required to have adopted CMMI and perform at a certain level.

Table 7-1 defines the five maturity levels of CMMI and shows how 350 organizations were assessed between April 2002 and August 2004.[18]

TABLE 7-1 CMMI maturity levels

Maturity level	Definition	Percentage of organizations at this level (as of August 2005)
Initial	Process unpredictable, poorly controlled, and reactive	5%
Managed	Process characterized for projects and is often reactive	36%
Defined	Process characterized for the organization and is proactive	29%
Quantitatively managed	Process measured and controlled	5%
Optimizing	Focus is on continuous process improvement	25%

Source: Capability Maturity Model Integration (CMMI) Overview, Carnegie Mellon University, www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview05.pdf, October 29, 2005.

KEY ISSUES IN SOFTWARE DEVELOPMENT

Although defects in any system can cause serious problems, the consequences of software defects in certain systems can be deadly. In these systems, the stakes involved in creating quality software are raised to the highest possible level. The ethical decisions involving a trade-off—if one might even be considered—between quality and other factors such as cost, ease of use, or time to market require extremely serious examination. The next sections discuss safety-critical systems and the special precautions companies must take in developing them.

Development of Safety-Critical Systems

A **safety-critical system** is one whose failure may cause injury or death. The safe operation of many safety-critical systems relies on the flawless performance of software; these systems control an automobile's antilock brakes, nuclear power plant reactors, airplane navigation, roller coasters, elevators, and numerous medical devices, to name just a few. The process of building software for such systems requires highly trained professionals, formal and rigorous methods, and state-of-the-art tools. Failure to take strong measures to identify and remove software errors from safety-critical systems "is at best unprofessional and at worst can lead to disastrous consequences."[19] However, even with these precautions, the software associated with safety-critical systems is still vulnerable to errors that can lead to injuries or death.

In June 1994, for example, a Chinook helicopter took off from Northern Ireland with 25 British intelligence officials to a security conference in Inverness. Just 18 minutes into its flight, the helicopter crashed on the peninsula of Kintyre in Argyll, Scotland, killing everyone aboard. A handwritten memo by a senior Ministry of Defense procurement officer revealed problems with the Chinook, "many of which were traced eventually back to software design and systems integration problems which were experienced from February to July 1994." In particular, the engine management software, which controlled the acceleration and deceleration of the engines, was suspect.[20]

One of the most widely cited software-related accidents in safety-critical systems involved a computerized radiation therapy machine called the Therac-25. This medical linear accelerator was designed to deliver either protons or electrons in high-energy beams that would destroy tumors with minimal impact on the surrounding healthy tissue. Between June 1985 and January 1987, six known accidents involving massive overdoses were caused by software errors in the Therac-25, leading to serious injuries and even death.[21] The Therac bug was subtle. If a fast-typing operator mistakenly selected X-ray mode and then used a particular editing key to change to electron mode, the display would appear to show a proper setting, when in reality the Therac had been configured to focus electrons at full power to a tiny spot on the body.[22]

The Therac-25 case illustrates that accidents seldom have a single root cause and that if only the symptoms of a problem are fixed, future accidents may still occur. With the benefit of hindsight, it is clear that poor decisions were made about the development and use of the Therac-25 machine. The vendor decided to eliminate hardware backup safety mechanisms that were present in earlier versions of the machine. The vendor also elected to reuse parts of earlier Therac-20 software in a different Therac-25 machine. The software was poorly documented and system testing was incomplete. The vendor failed to take a lead role in investigating the initial accident, insisting that the cause must have been human error. Finally, hospitals did not push hard enough for answers and continued to use the Therac-25 machine, even when they knew that accidents had occurred.

When developing safety-critical systems, a key assumption must be that safety will *not* automatically result from following your organization's standard development methodology. Safety-critical software must go through a much more rigorous and time-consuming development process than other kinds of software. All tasks—including requirements definition, systems analysis, design, coding, fault analysis, testing, implementation, and change control—require additional steps, more thorough documentation, and more checking and rechecking. As a result, safety-critical software takes much longer to complete and is much more expensive.

The key to doing this additional work is to appoint a project safety engineer who has explicit responsibility for the system's safety. The safety engineer uses a logging and monitoring system to track hazards from a project's start to its finish. The hazard log is used at each stage of the software development process to assess how it has accounted for detected hazards. Safety reviews are held throughout the development process, and a robust configuration management system tracks all safety-related documentation to keep it consistent with the associated technical documentation. Informal documentation is not acceptable for safety-critical system development; formal documentation is required, including verification reviews and signatures.

The increased time and expense of completing safety-critical software can draw developers into ethical dilemmas. For example, the use of hardware mechanisms to back up or verify critical software functions can help ensure safe operation and make the consequences of software defects less critical. However, such hardware may make the final product more expensive to manufacture or harder for the user to operate, making the product less attractive than a competitor's. Companies must weigh these issues carefully to develop the safest possible product that also appeals to customers. Another key issue is deciding when the QA staff has performed enough testing. How much testing is enough when you are building a product that can cause loss of human life? At some point, software developers must determine that they have completed sufficient QA activities and then sign off to indicate their approval. Determining how much testing is sufficient demands careful decision making.

When designing, building, and operating a safety-critical system, a great deal of effort must go into considering what can go wrong, the likelihood and consequences of such occurrences, and how risks can be averted or mitigated. One approach to answering these questions is to conduct a formal risk analysis. **Risk** is the probability of an undesirable event occurring times the magnitude of the event's consequences if it does happen. These consequences include damage to property, loss of money, injury to people, and death. For example, if an undesirable event has a 1 percent probability of occurring and its consequences would cost $1,000,000, then the risk can be determined as 0.01 x $1,000,000, or $10,000. This risk would be considered greater than that of an event with a 10 percent probability of occurring, at a cost of $100 (0.10 x $100 = $10). Risk analysis is important for safety-critical systems, but is useful for other kinds of software development as well.

Another key element of safety-critical systems is **redundancy**, the provision of multiple interchangeable components to perform a single function in order to cope with failures and errors. A simple redundant system would be an automobile with a spare tire or a parachute with a backup chute attached. A more complex system used in IT is a redundant array of independent disks (RAID), which is commonly used in high-volume data storage for file servers. RAID systems use many small-capacity disk drives to store large amounts of data and to provide increased reliability and redundancy. Should one of the drives fail, it can be removed and a new one inserted in its place. Data on the failed disk can be rebuilt automatically without the server ever having to be shut down, because the data has been stored elsewhere.

N-version programming is a form of redundancy that involves the execution of a series of program instructions simultaneously by two different systems. The systems use different algorithms to execute instructions that accomplish the same result. The results from the two systems are then compared; if a difference is found, another algorithm is executed to determine which system yielded the correct result. In some cases, instructions for the two systems are written by programmers from two different companies and run on different hardware devices. The rationale for N-version programming is that both systems are highly unlikely to fail at the same time under the same conditions. Thus, one of the two systems should yield a correct result. IBM employs N-version programming to reduce disk sector failures in data storage devices. Two pieces of code in the same application save a piece of data and then compare the data to ensure that no errors occurred.[23]

During times of widespread disaster, improper planning for redundant systems can lead to major problems. For example, Hurricane Katrina knocked out 2.5 million telephone lines, four TV stations, and 36 radio stations.[24] There were inadequate backup communication systems to replace the failed systems.

After an organization determines all pertinent risks to a system, it must decide what level of risk is acceptable. This decision is extremely difficult and controversial because it involves forming personal judgments about the value of human life, assessing potential liability in case of an accident, evaluating the surrounding natural environment, and estimating the system's costs and benefits. System modifications must be made if the level of risk in the design is judged to be too great. Modifications can include adding redundant components or using safety shutdown systems, containment vessels, protective walls, or escape systems. Another approach is to mitigate the consequences of failure by devising emergency procedures and evacuation plans. In all cases, organizations must ask how safe is safe enough, if human life is at stake.

Manufacturers of safety-critical systems must sometimes decide whether to recall a product when data indicates a problem. For example, automobile manufacturers have been known to weigh the cost of potential lawsuits against that of a recall. Drivers and passengers in affected automobiles (and in many cases, the courts) have not found this approach very ethical. Manufacturers of medical equipment and airplanes have had to make similar decisions, which can be complicated when data cannot pinpoint the cause of the problem. For example, there was great controversy in 2000 over the use of Firestone tires on Ford Explorers; numerous tire blowouts and Explorer rollovers caused multiple injuries and deaths. However, it was difficult to determine if the rollovers were caused by poor automobile design, faulty tires, or improperly inflated tires. Consumers' confidence in both products was nevertheless shaken.

Reliability is the probability of a component or system performing without failure over its product life. For example, if a component has a reliability of 99.9 percent, it has one chance in one thousand of failing over its lifetime. Although this chance of failure may seem very low, remember that most systems are made up of many components. As you add more components to the system, it becomes more complex and the chance of a failure increases. For example, imagine that you are building a complex system made up of seven components, each with 99.9 percent reliability. If none of the components has redundancy built in, the product has a 93.8 percent probability of operating successfully with no component malfunctions over its lifetime. If you build the same type of system using 10 components, each with 99.9 percent reliability, the overall probability of operating without an individual component failure falls to less than 60 percent! Thus, building redundancy into systems that are both complex and safety-critical is imperative.

One of the most important and difficult areas of safety-critical system design is the human interface. Human behavior is not nearly as predictable as the reliability of hardware and software components in a complex system. The system designer must consider what human operators might do to make a system work less safely or effectively. The challenge is to design a system that not only works as it should, but leaves the operator little room for erroneous judgment. For instance, a self-medicating pain relief system must allow a patient to press a button to receive more pain reliever, but the system also must regulate itself to prevent an overdose. Additional risk can be introduced if a designer does not anticipate the information an operator needs and how the operator will react under the daily pressures of actual operation, especially in a crisis. Some people keep their wits about them and perform admirably in an emergency, but others may panic and make a bad situation worse.

Poor interface design between systems and humans can greatly increase risk, sometimes with tragic consequences. For example, in July 1988, the guided missile cruiser *U.S.S. Vincennes* mistook an Iranian Air commercial flight for an enemy F-14 jet fighter and shot down the airliner over international waters in the Persian Gulf, killing almost 300 people. Some investigators blamed the tragedy on the confusing interface of the $500 million Aegis radar and weapons control system. The Aegis radar on the *Vincennes* locked onto an Airbus 300, but it was misidentified as a much smaller F-14 by its human operators. The Aegis operators also misinterpreted the system signals and thought that the target was descending, even though the airbus was actually climbing. A third human error was made in determining the target altitude—it was wrong by 4000 feet. As a result of this combination of human errors, the *Vincennes* crew thought the ship was under attack and shot down the plane.[25]

Quality Management Standards

The International Organization for Standardization (ISO), founded in 1947, is a worldwide federation of national standards bodies from some 100 countries. The ISO issued its 9000 series of business management standards in 1988. These standards require organizations to develop formal quality management systems that focus on identifying and meeting the needs, desires, and expectations of their customers.

The **ISO 9000** standard serves many industries and organizations as a guide to quality products, services, and management. Approximately 350,000 organizations have ISO 9000 certification in more than 150 countries. Although companies can use the standard as a management guide for their own purposes in achieving effective control, the bottom line for many is having a qualified external agency say that they have achieved ISO 9000 certification. Many companies and government agencies specify that a company must be ISO 9000-certified to win a contract from them.

To obtain this coveted certificate, an organization must submit to an examination by an external assessor and fulfill the following requirements:

1. Have written procedures for everything it does.
2. Follow those procedures.
3. Prove to an auditor that it has fulfilled the first two requirements. This proof can require observation of actual work practices and interviews with customers, suppliers, and employees.

The various ISO 9000 series of standards address the following activities:

ISO 9001	Design, development, production, installation, servicing
ISO 9002	Production, installation, servicing
ISO 9003	Inspection and testing
ISO 9000-3	The development, supply, and maintenance of software
ISO 9004	Quality management and quality systems elements

Failure mode and effects analysis (FMEA) is an important technique used to develop any ISO 9000-compliant quality system. FMEA is used to evaluate reliability and determine the effect of system and equipment failures. Failures are classified according to their impact on a project's success, personnel safety, equipment safety, customer satisfaction, and customer safety. The goal of FMEA is to identify potential design and process failures early in a project when they are relatively easy and inexpensive to correct. A failure mode describes how a product or process could fail to perform the desired functions described by the customer. An effect is an adverse consequence that the customer might experience. Unfortunately, most systems are so complex that there is seldom a one-to-one relationship between cause and effect. Instead, a single cause may have multiple effects and a combination of causes may lead to one effect or multiple effects.

Another example of a quality management standard comes from the aviation industry. Over the past 25 years, the role of software in aircraft manufacturing has expanded from controlling simple measurement devices to almost all aircraft functions, including navigation, flight control, and cockpit management. Given this expanded use of software, the aviation industry and its regulatory community needed an effective means of evaluating safety-critical software. To meet this need, the Radio Technical Commission for Aeronautics (RTCA) developed DO-178B/EUROCCAE ED-128 as the evaluation standard for the international aviation community. The RTCA is a private, nonprofit organization that develops recommendations for use by the Federal Aviation Administration (FAA) and the private sector. Aviation software developers use the RTCA standard to achieve a high level of confidence in safety-critical software.

Table 7-2 provides a useful checklist for an organization that wants to upgrade the quality of the software it produces. The preferred response to each question is *yes*.

TABLE 7-2 Manager's checklist for improving software quality

Questions	Yes	No
Has senior management made a commitment to quality software?	___	___
Have you used CMMI to evaluate your organization's software development process?	___	___
Have you adopted a standard software development methodology?	___	___
Does the methodology place a heavy emphasis on quality management and address how to define, measure, and refine the quality of the software development process and its products?	___	___
Are software project managers and team members trained in the use of this methodology?	___	___
Are software project managers and team members held accountable for following this methodology?	___	___
Is a strong effort made to identify and remove errors as early as possible in the software development process?	___	___
In the testing of software, are both static and dynamic testing used?	___	___
Are white-box testing and black-box testing used?	___	___
Has an honest assessment been made to determine if the software being developed is safety-critical?	___	___
If the software is safety-critical, are additional tools and methods employed, and do they include the following: project safety engineer, hazard logs, safety reviews, formal configuration management systems, rigorous documentation, risk analysis processes, and the FMEA technique?	___	___

Summary

1. Why do companies require high-quality software in business systems, industrial process control systems, and consumer products?

   High-quality software systems are needed because they are easy to learn and easy to use. They perform quickly and efficiently to meet their users' needs, operate safely and dependably, and have a high degree of availability that keeps unexpected downtime to a minimum.

   Such high-quality software has long been required to support the fields of air traffic control, nuclear power, automobile safety, health care, military and defense, and space exploration. Now that computers and software have become an integral part of our lives, more and more users are demanding high quality in their software. They cannot afford system crashes, lost work, or lower productivity, nor can they tolerate security holes through which intruders can spread viruses, steal data, and shut down Web sites.

2. What ethical issues do software manufacturers face in making trade-offs between project schedules, project costs, and software quality?

   Software manufacturers are under extreme pressure to reduce the time to market of their products. They are driven by the need to beat the competition in delivering new functionality to users, to begin generating revenue to recover the cost of development, and to show a profit for shareholders. The resources and time budgeted to ensure quality are often cut under the intense pressure to ship the new product. When forced to choose between adding more user features or doing more testing, most software development managers decide in favor of more features. After all, they reason, defects can always be patched in the next release, which will give customers an automatic incentive to upgrade. The additional features will make the release more useful and therefore easier to sell to customers. Many customers challenge whether the decision to cut quality is ethical.

3. What are the four most common types of software product liability claims, and what actions must plaintiffs and defendants take to be successful?

   Software product liability claims are frequently based on strict liability, negligence, breach of warranty, or misrepresentation. Strict liability means that the defendant is responsible for injuring another person, regardless of negligence or intent. The plaintiff must prove only that the software product is defective or unreasonably dangerous and that the defect caused the injury. When sued for negligence, a software supplier is not held responsible for every product defect that causes customer or third-party loss. Instead, responsibility is limited to harmful defects that could have been detected and corrected through "reasonable" software development practices.

   If the product fails to meet its warranty, the buyer or lessee can sue for breach of warranty. The plaintiff must have a valid contract that the supplier did not fulfill in order to win a breach of warranty claim. Intentional misrepresentation occurs when a seller or lessor either misrepresents the quality of a product or conceals a defect in it. Most software manufacturers use limited warranties and disclaimers to avoid any claim of misrepresentation.

4. What are the essential components of a software development methodology, and what are its benefits?

   A software development methodology defines the activities in the software development process, defines individual and group responsibilities for accomplishing objectives, recommends specific techniques for accomplishing the objectives, and offers guidelines for managing the quality of the products during the various stages of the development cycle.

   Using an effective development methodology enables a manufacturer to produce high-quality software, forecast project completion milestones, and reduce the overall cost to develop and support software. It also protects software manufacturers from legal liability for defective software in two ways: it reduces the number of software errors that could cause damage, and makes it more difficult to prove negligence.

5. How can Capability Maturity Model Integration improve an organization's software development process?

   CMMI defines five levels of software development maturity and identifies the issues that are most critical to software quality and process improvement. Its use can improve an organization's ability to predict and control quality, schedule, costs, cycle time, and productivity when acquiring, building, or enhancing software systems. CMMI also helps software engineers to analyze, predict, and control selected properties of software systems.

6. What is a safety-critical system, and what actions are required during its development?

   A safety-critical system is one whose failure may cause injury or death. In the development of safety-critical systems, a key assumption is that safety will *not* automatically result from following an organization's standard software development methodology. Safety-critical software must go through a much more rigorous and time-consuming development and testing process than other kinds of software; the appointment of a project safety engineer and the use of a hazard log and risk analysis are common.

Self-Assessment Questions

1. The impact of a software defect can be quite subtle or very serious. True or False?
2. _____ is the degree to which the attributes of a software product enable it to meet the needs of its users.
3. Which of the following is *not* a major cause of poor software quality?
   a. Developers do not know how to design quality into software or do not take the time to do it.
   b. Programmers make mistakes in turning design specifications into lines of code.
   c. Software manufacturers are under extreme pressure to reduce the time to market of their products.
   d. Many organizations avoid buying the first release of a major software product.
4. A standard, proven work process for the development of high-quality software is called a(n) _____.
5. The cost to identify and remove a defect in an early stage of software development can be 100 times less than removing a defect in an operating piece of software that has been distributed to many customers. True or False?

6. _____ is a form of testing a unit of software by entering test data and comparing the actual results to the expected results.
   a. Dynamic testing
   b. Static testing
   c. Integration testing
   d. System testing
7. _____ is an approach that defines the essential elements of an effective process and outlines a system for continuously improving software development.
   a. ISO 9000
   b. FMEA
   c. CMMI
   d. DO-178B
8. A system whose failure may cause injury or even death is called _____.
9. _____ is the probability of an undesirable event occurring times the magnitude of the event's consequence if it does happen.
   a. Risk
   b. Redundancy
   c. Reliability
   d. Availability
10. A reliability evaluation technique that can determine the effect of system and equipment failures is _____.

Review Questions

1. What is a software defect?
2. How does redundancy reduce the risk associated with a system component?
3. What must a defendant prove to win a strict liability claim? What legal defenses are commonly used by defendants?
4. What must a defendant prove to win a negligence claim? What legal defenses are commonly used by defendants?
5. What is breach of warranty? What must the plaintiff prove to win a breach of warranty claim? How can a software manufacturer defend against potential breach of warranty claims?
6. What is the difference between quality management and quality assurance?
7. What is a software development methodology? What is its purpose?
8. Why is it critical to identify and remove defects early in the software development process?
9. Identify the various types of testing used in software development.
10. What is a safety-critical system? What additional precautions are needed when developing such a system?

11. What is the purpose of the ISO 9000 quality standard? How is it similar to and different from CMMI?
12. What is FMEA? What is its goal?

Discussion Questions

1. Identify your top three criteria for a quality system. Briefly discuss your rationale for selecting these criteria.
2. Explain why the cost to identify and remove a defect in the early stages of software development might be 100 times less than removing a defect in software that has been distributed to hundreds of customers.
3. You are considering using N-version programming with two software development firms and hardware devices for the navigation system of a guided missile. Briefly describe what this means and outline several advantages and disadvantages of this approach.
4. Discuss the implications to a project team of classifying a piece of software as safety-critical.
5. You have been asked to draft a boilerplate warranty for a software contractor that will absolutely protect the firm from being sued successfully for negligence or breach of contract. Is this possible? Why or why not?
6. Discuss why an organization may elect to use a separate, independent team for quality testing rather than the group of people that originally developed the software.
7. You are considering contracting for the development of new software that is essential to the success of your midsized manufacturing firm. One candidate firm boasts that its software development practices are at level 4 of CMMI. Another firm claims that all its software development practices are ISO-9000 compliant. How much weight should you give to these certifications when deciding which firm to use? Do you think that a firm could lie or exaggerate its level of compliance with these standards?

What Would You Do?

1. Read the fictitious Killer Robot case at the Web site for the Online Ethics Center for Engineering & Science at *www.onlineethics.com* (look under Computer Science and Internet cases). The case begins with the manslaughter indictment of a programmer for writing faulty code that resulted in the death of a robot operator. Slowly, over the course of many articles, you are introduced to several factors within the corporation that contributed to the accident. After reading the case, answer the following questions:
   a. Responsibility for an accident is rarely defined clearly and rarely is traceable to one or two people or causes. In this fictitious case, it is clear that a large number of people share responsibility for the accident. Identify all the people you think were at least partially responsible for the death of Bart Mathews, and why you think so.
   b. Imagine that you are the leader of a task force assigned to correct the problems uncovered by this accident. Develop a list of the top 10 critical actions to take to avoid future problems. What process would you use to identify the most critical actions?

c. If you were in Ms. Yardley's position, what would you have done when Ray Johnson told you to fake the test results? How would you justify your decision?

2. You are the project manager for developing the latest release of your software firm's flagship product. The product release date is just two weeks away and enthusiasm for the product is extremely high among your customers. Stock market analysts are forecasting sales of more than $25 million per month. If so, earnings per share will increase by nearly 50 percent. There is just one problem: two key features promised to customers in this release have several bugs that would severely limit the features' usefulness. You estimate that at least six weeks are needed to find and fix the problems. In addition, even more time is required to find and fix 50 additional, less severe bugs uncovered by the QA team. What do you recommend to management?

3. You have been assigned to manage software that controls the shutdown of chemical reactors, but your manager insists it is not safety-critical software. The software senses temperatures and pressures within a 50,000-gallon stainless steel vat and dumps in chemical retardants to slow down the reaction if it gets out of control. In the worst possible scenario, failure to stop a runaway reaction would result in a large explosion that would send fragments of the vat flying and spray caustic soda in all directions. Your manager points out that the stainless steel vat is surrounded by two sets of protective concrete walls and that the reactor's human operators can intervene in case of a software failure. He feels that these measures would protect the plant employees and the surrounding neighborhood if the shutdown software failed. Besides, he argues, the plant is already more than a year behind its scheduled start-up date. He cannot afford the additional time required to develop the software if it is classified as safety-critical. How would you work with your manager and other appropriate resources to decide whether the software is safety-critical?

4. You are a senior software development consultant with a major consulting firm. Two years ago, you conducted the initial assessment of the ABCXYZ Corporation's software development process. Using CMMI, you determined their level of maturity to be level 1. Since your assessment, the organization has spent a lot of time and effort following your recommendations to raise its level of process maturity to the next level. The organization appointed a senior member of the IT staff to be a process management guru and paid him $150,000 per year to lead the improvement effort. This senior member adopted a methodology for standard software development and required all project managers to go through a one-week training course at a total cost of more than $2 million.

   Unfortunately, these efforts did not significantly improve process maturity because senior management failed to hold project managers accountable for actually using the standard development methodology in their projects. Too many project managers convinced senior management that the new methodology was not necessary for their project and would just slow things down. However, you are concerned that when senior management learns that no real progress has been made, they will refuse to accept partial blame for the failure and instead drop all attempts at further improvement. You want senior management to ensure that the new methodology is used on all projects—no more exceptions. What would you do?

5. You are the CEO for a small, struggling software firm that produces educational software for high school students. Your latest software is designed to help students improve their SAT and ACT scores for getting into college. To prove the value of your software, a group of 50 students who had taken the ACT test were tested again after using your software for just two weeks. Unfortunately, there was no dramatic increase in their scores. A statistician you hired to ensure objectivity in measuring the results claimed that the variation in test scores was statistically insignificant.

   A small core group of educators and systems analysts will need at least six months to start again from scratch and design a viable product. Programming and testing could take another six months. Another option would be to go ahead and release the current version of the product and then, when the new product is ready, announce it as a new release. This would generate the cash flow necessary to keep your company afloat and save the jobs of 10 or more of your 15 employees.

   Given this information about your company's product, what would you do?

Cases

1. New Wireless Technologies Reduce Medical Errors

In December 1995, seven-year-old Ben Kolb checked into Martin Memorial Hospital in Stuart, Florida, to undergo a simple ear surgery. During the procedure, the anesthesiologist injected the boy with an epinephrine solution 1000 times more concentrated than what had been specified. The boy went into cardiac arrest and died the following day.

As a result of cases like Ben Kolb's, medical institutions—which have traditionally dragged their heels when considering new IT systems—are turning to new wireless technologies for solutions. Forecasts originally predicted that 60 to 70 percent of medical institutions would adapt wireless technology by 2007. In fact, this percentage was already reached by mid-2005.

Hospitals use wireless technologies to prevent medical errors in a variety of ways. For example, doctors use handheld devices to access and update patients' medical records at their bedsides. Nurses scan bar codes on medications and on patients' wrist bands. By accessing the physician's order entry system, nurses ensure that they are administering the correct dosage and medication. Cell phones and wireless badges also improve communication between hospital staff.

Hospitals report that these technologies have had a significant effect on patient safety. For example, Beloit Memorial Hospital in Wisconsin reported a 67 percent reduction in errors four months after implementing a wireless medication administration system. Boston-based Brigham and Women's Hospital announced a 55 percent reduction in medical errors.

Few other studies have been carried out to date, yet medical practitioners believe that wireless systems will reduce errors for several reasons. Doctors prescribe more than 15,000 different drugs, and hundreds of them are similar in spelling or sound. Bad handwriting or bad hearing translates into incorrect dosages and medications. The system also helps hospital staff identify medications that may elicit allergic reactions or may interact unfavorably with other drugs the patient is taking. The system also increases the likelihood that medications will be administered on time.

However, a few critical studies have warned of the potential harmful effects of instituting these systems. One study reported that Brigham and Women's Hospital actually experienced a drastic increase in the number of potential medical errors, which were prevented by alert nurses who caught the errors prior to administrating the medication. The problem was traced to a bug in the physician ordering software that affected orders of potassium infusions.

An article in the *Journal of the American Medical Association* (*JAMA*) in March 2005 pointed out other flaws in computerized physician order entry systems (CPOE). These systems rely on the integration of existing IT and paper-based systems, and problems with this integration can produce medical errors. Furthermore, the CPOE interface may not suit the work flow process at the hospital. The physician, for example, might be forced to view 20 screens to review all the medications a single patient is taking.

"Good computerized physician order entry systems are, indeed, very helpful and hold great promise; but, as currently configured, there are at least two dozen ways in which CPOE systems significantly, frequently, and commonly facilitate errors—and some of those errors can be deadly," said Ross Koppel, author of the *JAMA* article.

Despite Koppel's assessments, the medical community is racing forward with wireless systems, with good reason. A report issued by the Institute of Medicine (IOM) of the National Academies finds that at least 44,000 Americans are killed each year in hospitals due to medical error, and this figure climbs each year. Medical errors kill more Americans each year than highway accidents, AIDS, or breast cancer.

Because wireless systems hold out the hope of combating this growing problem, even critics like Koppel do not suggest their abandonment. Instead, critics warn against blind faith in technology and ask healthcare professionals to use these systems carefully to minimize both human and machine-produced medical errors.

Questions:

1. How should hospital staff and IT staff work together to ensure the success of safety-critical wireless systems designed to reduce medical error?
2. Wireless systems can record who administered the medicine, the dosage administered, the time of day, and other details. Is such clear accountability good or bad for the health industry? Do you think nurses or other healthcare practitioners may be hesitant to use the systems because of potential legal consequences?
3. Can you identify additional safety measures that should be built into these wireless systems?

2. Prius Plagued by Programming Error

In May 2005, the National Highway Traffic Safety Administration (NHTSA) revealed that it had opened an investigation following complaints of engines stalling in Toyota's Prius hybrid. Motorists reported that while they were driving or stuck in traffic, warning lights flashed on and the gas engine suddenly switched off. In all cases, the motorists were able to maneuver to the side of the road safely because the electric motor, the steering, and the brake system continued to function.

As the second most fuel-efficient car in the United States, the Prius delivers better mileage per gallon by shifting from its gas engine to its electric motor. The electronic control unit is responsible for the smooth transition between the gas engine and the motor. Toyota eventually determined that an error in the software used by this electronic control unit caused it to malfunction and the gas engine to stall.

Today, the average car is equipped with 30 to 40 microprocessors, so software quality is a critical issue for automakers. With 35 million lines of code per car, the potential for error is enormous. IBM automotive software specialist Stavros Stefanis reports that as many as one-third of all warranty claims result from software glitches and electronic defects. "It's a big headache for the automakers," said Stefanis.

This headache is compounded by the fact that software controls safety-related systems such as engine performance, steering, antilock brakes, and air bags. A programming error could have an enormous financial impact on an automaker in major lawsuits and recalls.

So, when NHTSA opened its investigation, Toyota, a company that has established a reputation for reliability, proved extremely cooperative. In fact, the company had already issued a service bulletin to owners of 2004 and 2005 Prius hybrids in October 2004. Toyota conducted an internal investigation, and in October 2005 recalled 75,000 of the more than 88,000 2004 and 2005 Prius hybrids sold in the United States. The company recalled an additional 85,000 units sold abroad.

Toyota spokeswoman Allison Takahashi reported that NHTSA determined passenger safety was not at issue: "We are voluntarily initiating a customer-service campaign to assure that this unusual occurrence does not cause inconvenience." Following the announcement of the campaign, NHTSA promptly closed its investigation.

The investigation, however, did not slow sales of the most popular hybrid on the market. 2005 sales were up 200 percent over the previous year when the recall was announced. In fact, Toyota had to concentrate a good deal of its software development efforts on projects that would increase production to meet the demand for new hybrids in the United States.

Although the recall's effect on sales was probably negligible, the cost of the recall may be significant. Toyota will pay for advertising and other costs involved in contacting Prius owners about the recall, and then will pay for the repairs. The recall may also serve to warn automakers about the costs of human errors that go undetected during software production.

Questions:

1. Do you agree with NHTSA's assessment that problems with the Prius were not a safety-critical issue? In such cases, who should decide whether a software bug creates a safety-critical issue—the manufacturer, consumers, government agencies, or some other group?
2. How would the issue be handled differently if it were a safety-critical matter? Would the issue be handled differently if the costs involved were not so great?
3. As the amount of hardware and software embedded in the average car continues to grow, what steps can automakers take to minimize warranty claims and ensure customer safety?

3. Patriot Missile Failure

The Patriot is an Army surface-to-air missile system that defends against aircraft and cruise missiles, and more recently, against short-range ballistic missiles. The system was designed in the 1960s and enhanced in the 1980s. The short-range antimissile capability was incorporated into the Patriot PAC-2 version of the missile.

Following the Iraqi invasion of Kuwait in August 1990, the United States deployed the Patriot PAC-2 missile to Saudi Arabia during Operation Desert Shield. At the start of Desert Shield, the U.S. arsenal included only three PAC-2 missiles. PAC-2 production was accelerated so that by January 1991, 480 missiles were available. Patriot battalions were deployed to Saudi Arabia and then to Israel to defend key assets, military personnel, and citizens against Iraqi Scud missiles. Iraq launched 81 modified-Scud ballistic missiles into Israel and Saudi Arabia during the conflict.

The Iraqis modified the Scud missile to increase its range and boost its speed by as much as 25 percent. They reduced the weight of the warhead, enlarged the fuel tanks, and modified its flight so that all of the fuel was burned during the early phase of flight, rather than continuously. As a result of these modifications, the missiles became structurally unstable and often broke into pieces in the upper atmosphere. This instability made the warhead extremely difficult to intercept; also, radar mistakenly could lock onto pieces of the fragmented missile rather than the warhead.

The Patriot missile is launched and guided to the target through three phases. First, the missile guidance system turns the Patriot launcher to face the incoming missile. Second, the computer control system guides the missile toward the incoming missile. Third, the Patriot missile's internal radar receiver guides it to intercept the incoming missile.

During Desert Shield, the Patriot's radar constantly swept the sky for any object that had the flight characteristics of a Scud missile. The range gate is an electronic detection device within the radar system that uses the last observation of an object to forecast an area in the air space where the radar system should next see the object, if it truly was a Scud. This forecast is a function of the object's observed velocity and the time of the last radar detection. If the range gate determined that the detected target was a Scud, and if the Scud was in the Patriot's firing range, the Patriot battery fired its missiles.

On February 11, 1991, the Patriot Project Office received data from Patriots deployed in Israel, identifying a 20 percent shift in the system's radar range gate after the system had been running for eight consecutive hours. This shift was significant; it meant that the target was no longer in the center of the range gate and the probability of successfully tracking the target was greatly reduced. The Army knew that the Patriot system could not track a Scud with a range gate shift of 50 percent or more.

In the Patriot radar system, time is kept from time of system start-up and measured by the system's internal clock in tenths of a second. The longer the system ran, the less accurate the elapsed time calculation became. Consequently, after the Patriot computer control system ran continuously for extended periods, the range gate made an inaccurate estimate of the area in the air space where the radar system should next see the object. This error could cause the radar to lose the target and fool the system into thinking there was no incoming Scud.

Army officials assumed that other Patriot users were not running their systems for eight hours or more at a time, and that the Israeli experience was an anomaly. However, the Patriot Project Office analyzed the Israeli data and confirmed some loss in targeting accuracy. As a result, they made a software change to compensate for the inaccurate time calculation. This change was included in a modified software version that was released on February 16, 1991.

The Patriot Project Office sent a message to Patriot users on February 21, 1991, informing them that very long run times could cause a shift in the range gate, resulting in difficulty tracking the target. The message also advised users that a software change was on the way that would improve the system's targeting. However, the message did not specify what constituted "very long run times." Patriot Project Office officials assumed that users would not continuously run the batteries for so long that the Patriot would fail to track targets. Therefore, they did not think that more detailed guidance was required.

On February 25, 1991, a Patriot missile defense system operating at Dhahran, Saudi Arabia, failed to track and intercept an incoming Scud. The enemy missile subsequently hit an Army barracks and killed 28 Americans. The ensuing investigation revealed that at the time of the incident, the Patriot battery had been operating continuously for more than 100 hours. The long run time resulted in an inaccurate time calculation, which in turn caused the range gate to shift so much that the system could not track, identify, or engage the incoming Scud.

By cruel fate, modified software that fixed the inaccurate time calculation arrived in Dhahran the very next day. Army officials attributed the delay to the difficulties of arranging air and ground transportation during wartime.

The Army did not have the luxury of collecting definitive performance data during Operation Desert Storm. After all, they were operating in a war zone, not a test range. As a result, there was insufficient and conflicting data on the effectiveness of the Patriot missile. At one extreme was an early report that claimed the Patriot destroyed about 96 percent of the Scuds engaged in Saudi Arabia and Israel. (This presumably did not include Scuds the Patriot failed to engage due to the software error.) At the other extreme, in only about 9 percent of the engagements did observers actually see a Scud destroyed or disabled after a Patriot detonated nearby. Of course, some "kills" could have been effected out of their range of vision.

Questions:

1. With the benefit of hindsight, what steps could have been taken during development of the Patriot software to avoid the problems that led to the loss of life? Do you think these steps would have improved the Patriot's effectiveness enough to make it obvious that the missile was a strong deterrent against the Scud? Why or why not?
2. What ethical decisions do you think the U.S. military made in deploying the Patriot missile to Israel and Saudi Arabia and in reporting the effectiveness of the Patriot system?
3. What key lessons can be taken from this example of safety-critical software development and applied to the development of business information system software?

End Notes

[1] Crawford, Michael, "Trend Micro Bug Down To Over-Quick Testing," *TechWorld*, www.techworld.com/news/index.cfm?RSS&NewsID=3559, April 26, 2005.

[2] Kallender, Paul, "Trend Micro Lowers Forecast, Blames Software Bug: The Flaw Affected Thousands Of Its Customers," *Computerworld*, www.computerworld.com/softwaretopics/os/windows/story/0,10801,103183,00.html, July 14, 2005.

[3] Leyden, John, "PC-cillin Killed My PC," *The Register*, www.theregister.co.uk/2005/04/25/pc-cillin_duff_update, April 25, 2005.

[4] Keizer, Greg, "Apple Patches Bevy Of Tiger, Panther Bugs (Again)," *TechWeb News*, www.informationweek.com, September 25, 2005.

[5] Keizer, Greg, "RealNetworks Fixes Linux RealPlayer Flaw," *TechWeb News*, www.informationweek.com, October 3, 2005.

[6] Keizer, Greg, "Kaspersky Says It's Fixed AV Scanner Flaw," *TechWeb News*, www.informationweek.com, October 4, 2005.

[7] Keizer, Greg, "Microsoft Plans 9 Patches Next Week," *TechWeb News*, www.informationweek.com, October 7, 2005.

[8] McMillan, Robert, "Symantec AntiVirus Scan Carries Critical Bug," *CIO*, www.cio.com, October 7, 2005.

[9] Keizer, Greg, "Google Plugs Cross-Scripting Security Hole," *TechWeb News*, www.informationweek.com, October 10, 2005.

[10] Sanders, Tom, "Windows Chokes on Latest Microsoft Patch," *StickyMinds*, www.stickyminds.com, October 17, 2005.

[11] Humphrey, Watts S., "Why Quality Pays," *Computerworld*, www.computerworld.com, May 20, 2002.

[12] Bierha, Bruce A., "Faulty Software or Puffery," *Computerworld*, www.computerworld.com, March 28, 1994.

[13] Kaner, Cern, "Bad Software – Who Is Liable?," keynote address of Quality Assurance Institute Regional Conference, Seattle, www.badsoftware.com, June 1998.

[14] Cheeseman, Henry R., *Contemporary Business Law*, 3rd Edition, page 362, Prentice-Hall, Upper Saddle River, NJ, 2000.

[15] "Software Company Not Liable for $2 Million Bug in Bid," *Engineering Times*, www.nspe.org, July 2000.

[16] Boehm, Barry W., *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice-Hall, 1981.

[17] Web site for Carnegie Mellon Software Engineering Institute, www.sei.cmu.edu/cmm, October 29, 2005.

[18] Capability Maturity Model Integration (CMMI) overview, Carnegie Mellon University, www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview05.pdf, October 29, 2005.

[19] Bowen, Jonathan P., "The Ethics of Safety-Critical Systems," www.cs.rdg.ac.uk, October 29, 2005.

[20] Collins, Tony, "Minister Denies Chinook Claim," *Computer Weekly*, www.computerweekly.com, July 1, 1999.

[21] Leveson, Nancy G. and Turner, Clark S., "An Investigation of the Therac-25 Accidents," *IEEE Computer*, Vol. 26, No. 7, pages 18-41, July 1993.

[22] "Getting Serious with Year 2000, Sometimes Bugs Can Be Deadly," *InfoWeek*, www.infoweek.com, April 13, 1998.

[23] Mearian, Lucas, "Bulletproof Storage," *Computerworld*, www.computerworld.com, April 11, 2005.

[24] Gross, Grant, "FCC Head: Hurricane Shows Need for Redundant Telecom," *Computerworld*, www.computerworld.com, September 22, 2005.

[25] Church, George J., "High-Tech Horror," *Time*, www.time.com, July 18, 1988.

Sources for Case 1

"Panacea or Pandora's Box: Penn Study Shows that Computerized Physician-Order Entry Systems Often Facilitate Medication Errors," University of Pennsylvania Health System, Department of Public Affairs Web site, www.uphs.upenn.edu/news/News_Releases/mar05/CPOE.htm, March 08, 2005.

Berger, Robert G. and Kichak, J. P., "Computerized Physician Order Entry: Helpful or Harmful," *Journal of the American Medical Informatics Association*, www.pubmedcentral.nih.gov/articlerender.fcgi?artid=353014, March-April 2004.

Havenstein, Heather, "Wireless Leaders & Laggards: Health Care," *Computerworld*, www.computerworld.com/printthis/2005/0,4814,101711,00.html, May 16, 2005.

Sources for Case 2

Associated Press, "Toyota Recalls 160,000 Prius Hybrids," *Washington Post*, www.washingtonpost.com/wp-dyn/content/article/2005/10/14/AR2005101400127.html, October 14, 2005.

Duvall, Mel, "Software Bugs Threaten Toyota Hybrids," *Baseline*, www.baselinemag.com/article2/0,1397,1843934,00.asp?kc=BANKT0209KTX1K0100464, August 4, 2005.

Nauman, Matt, "Toyota to Fix Software Problem on 75,000 Prius Hybrids," *San Jose Mercury News*, www.siliconvalley.com/mld/siliconvalley/news/local/12893923.htm, October 13, 2005.

Sources for Case 3

Carlone, Ralph V., "Patriot Missile Defense—Software Problem Led to Systems Failure at Dhahran, Saudi Arabia," GAO Report B-247094, www.fas.org, February 24, 1992.

"Data Does Not Exist to Say Conclusively How Well Patriot Performed," GAO Report B-250335, www.fas.org, September 22, 1992.

"Taiwan Interested in Latest Patriot Missiles," *Asian Political News*, www.findarticles.com, November 30, 1998.

"U.S. To Sell 14 Upgraded Missile Systems to S. Korea," *Asian Political News*, www.findarticles.com, November 15, 1999.

"Frontline: The Gulf War: Weapons: MIM -104 Patriot," PBS Web site, www.pbs.org, January 20,1996.