Chapter 7

Computer Reliability

> The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.
>
> DOUGLAS ADAMS, Mostly Harmless

7.1 Introduction

Computer databases track many of our activities. What happens when a computer is fed bad information, or when someone misinterprets the information they retrieve from a computer? We are surrounded by devices containing embedded computers. What happens when a computer program contains an error that causes the computer to malfunction? Sometimes the effects of a computer error are trivial. You are playing a game on your PC, do something unusual, and the program crashes, forcing you to start over. At other times computer malfunctions result in a real inconvenience. You get an incorrect bill in the mail, and you end up spending hours on the phone with the company's customer service agents to get the mistake fixed. Some software bugs have resulted in businesses making poor decisions that have cost them millions of dollars. On a few occasions, failures in a computerized system have even resulted in fatalities.

In this chapter we examine various ways in which computerized systems have proven to be unreliable. Systems typically have many components, of which the computer is just one. A well-engineered system can tolerate the malfunction of any single component without failing. Unfortunately, there are many examples of systems in which the computer was the weakest link and a computer error led to the failure of the entire system. The failure may have been due to a data entry or data retrieval error, poor design, or inadequate testing. Through a variety of examples, you will gain a greater appreciation for the complexity of building a reliable computerized system.

We also take a look at computer simulations, which are playing an increasingly important role in modern science and engineering. We survey some of the uses to which these simulations are put and describe how those who develop simulations can validate the underlying models.

Software engineering arose out of the difficulties organizations encountered when they

began constructing large software systems. Software engineering refers to the use of processes and tools that allow programs to be created in a more structured manner. We describe the software development process and provide evidence that more software projects are being completed on time and on budget.

At the end of the chapter we take a look at software warranties. Software manufacturers typically disclaim any liability for lost profits or other consequential damages resulting from the use of their products. We discuss how much responsibility software manufacturers ought to take for the quality of their products. Some say software should be held to the same standards as other products, while others say we ought to have a different set of expectations when it comes to the reliability of the software we purchase.

7.2 Data Entry or Data Retrieval Errors

Sometimes computerized systems fail because the wrong data have been entered into them or because people incorrectly interpret the data they retrieve. In this section we give several examples of wrong actions being taken due to errors in data entry or data retrieval.

7.2.1 Disfranchised Voters

In the November 2000 general election, Florida disqualified thousands of voters because pre-election screening identified them as felons. The records in the computer database, however, were incorrect; the voters had been charged with misdemeanors. Nevertheless, they were forbidden from voting. This error may have affected the outcome of the Presidential election [1].

7.2.2 False Arrests

As we saw in Chapter 5, the databases of the National Crime Information Center (NCIC) contain a total of about 40 million records related to stolen automobiles, missing persons, wanted persons, suspected terrorists, and much more. There have been numerous stories of police making false arrests based on information they retrieved from the NCIC. Here are three.

Sheila Jackson Stossier, an airline flight attendant, was arrested at the New Orleans airport by police who confused her with Shirley Jackson, who was wanted in Texas. She spent one night in jail and was detained for five days [2].

California police, relying upon information in the NCIC, twice arrested and jailed Roberto Hernandez as a suspect in a Chicago burglary case. The first time he was jailed for 12 days, while the second time he was held for a week before he was freed. They had confused him with another Roberto Hernandez, who had the same height and weight. Both Hernandezes had brown hair, brown eyes, and tattoos on their left arms. They also had the same birthday, and their Social Security numbers differed by only a single digit [3].

Someone used personal information about Michigan resident Terry Dean Rogan to obtain a California driver's license using his name. After this person was arrested for two homicides and two robberies, police entered information about these crimes into the NCIC under his

false identity. Over a period of 14 months, the real Terry Dean Rogan was arrested five times by Los Angeles police, three times at gun point, even though he and Michigan police had tried to get the NCIC records corrected after his first arrest. Rogan sued the Los Angeles Police Department and was awarded $55,000 [2].

7.2.3 Analysis: Accuracy of NCIC Records

Stepping away from a requirement of the Privacy Act of 1974, the Justice Department announced in March 2003 that it would no longer require the FBI to ensure the accuracy of information about criminals and crime victims before entering it in the NCIC database [4].

Should the U.S. government take responsibility for the accuracy of the information stored in NCIC databases?

The Department of Justice argues that it is impractical for it to be responsible for the information in the NCIC database [5]: Much of the information that gets entered into the database is provided by other law enforcement and intelligence agencies. The FBI has no way of verifying that all the information is accurate, relevant, and complete. Even when the information is coming from inside the FBI, agents should be able to use their discretion to determine which information may be useful in criminal investigations. If the FBI strictly followed the provisions of the Privacy Act and verified the accuracy of every record entered into the NCIC, the amount of information in the database would be greatly curtailed. The database would be a much less useful tool for law enforcement agencies. The result could be a decrease in the number of criminals arrested by law enforcement agencies.

Privacy advocates counter that the accuracy of the NCIC databases is now more important than ever, because an increasing number of records are stored in these databases. As more erroneous records are put into the database, the probability of innocent American citizens being falsely arrested also increases.

Which argument is stronger? Let's focus on one of the oldest NCIC databases: the database of stolen vehicles. The total amount of harm caused to society by automobile theft is great. Over one million automobiles are stolen in the United States every year. Victims of car theft are subjected to emotional stress, may sustain a financial loss, and can spend a lot of time trying to recover or replace the vehicle. In addition, the prevalence of automobile theft harms everyone who owns a car by raising insurance rates. In the past, car thieves could reduce the probability that a stolen car would be recovered by transporting it across a state line. Because the NCIC database contains information about stolen vehicles throughout the United States, it enables law enforcement officials to identify cars stolen anywhere in the nation. At the present time, just over half of all stolen vehicles are recovered. If we make the conservative estimate that the NCIC has increased the percentage of recovered cars by just 10 percent, more than 50,000 additional cars are returned to their owners each year, a significant benefit. One the other hand, if an error in the NCIC stolen car database leads to a false arrest, the harm caused to the innocent driver is great. However, there are only a few stories of false arrest stemming from errors in the NCIC stolen car database. The total amount of benefit derived from the NCIC database of stolen automobiles appears to be much greater than the total amount of harm it has caused. We conclude the creation and maintenance of this database has been the right course of action.

7.3 Software and billing Errors

Even if the data entered into a computer are correct, the system may still produce the wrong result or collapse entirely if there are errors in the computer programs manipulating the data. Newspapers are full of stories about software bugs or "glitches." There is a selection of stories that have appeared in print in the past few years.

7.3.1 Errors Leading to System Malfunctions

Linda Brooks of Minneapolis, Minnesota, opened her mail on July 21, 2001, and found a phone bill for $57, 346.20. A bug in Qwest's billing software caused it to charge some customers as much as $600 per minute for the use of their cell phones. About 1.4 percent of Qwest's customers, 14,000 in all, received incorrect bills. A Qwest spokesperson said the bug was in a newly installed billing system [6].

The U.S. Department of Agriculture implemented new livestock price-reporting guidelines after discovering that software errors had caused the USDA to understate the prices meat packers were receiving for beef. Since beef producers and packers negotiate cattle contracts based on the USDA price reports, the errors cost beef producers between $15 and $20 million [7].

In 1996 a software error as the U.S. Postal Service caused it to return to the senders two week's worth of mail addressed to the Patent and Trademark Office. In all, 50,000 pieces of mail were returned to sender [8].

A University of Pittsburgh study revealed that for most students, computer spelling and grammar error checkers actually increased the number of errors they made [9, l0].

Thailand's finance minister was trapped inside his BMW limousine for l0 minutes when the on-board computer crashed, locking all doors and windows and turning off the air-conditioning. Security guards had to use sledge hammers to break a window, enabling Suchart Jaovisidha and his driver to escape [11].

Between September 2008 and May 2009 hundreds of families living in public housing in New York City were charged too much rent because of an error in the program that calculated their monthly bills. For nine months the New York City Housing Authority did not take seriously the renters' complains that they were being overcharged. Instead, it took to court many of the renters who did not make the higher payments and threatened them with eviction [12].

7.3.2 Errors Leading to System Failures

On the first day a new, fully computerized ambulance dispatch system became operational in the City of London, people making emergency calls were put on hold for up to 30 minutes, the system lost track of some calls, and ambulances took up to three hours to respond. As many as 20 people died because ambulances did not arrive in time [13].

A software error led the Chicago Board of Trade to suspend trading for an hour on January 23, 1998. Another bug caused it to suspend trading for 45 minutes on April 1, 1998. In both cases, the temporary shutdown of the trading caused some investors to lose money [14]. System errors caused trading on the London International Financial Futures

and Options Exchange to be halted twice within two weeks in May 1999. The second failure idled dealers for an hour and a half [15].

Japan's air traffic control system went down for an hour on the morning of March 1, 2003, delaying departures for hours. The backup system failed at the same time as the main system, which was out of commission for four hours. Airports kept in touch via telephone, and no passengers were put at risk. However, some flights were delayed over two hours, and 32 domestic flights had to be canceled [16].

A new laboratory computer system at Los Angeles County+USC Medical Center became backlogged the day after it was turned on. For several hours on both April 16 and 17, 2003, emergency room doctors told the County of Los Angeles to stop sending ambulances, because the doctors could not get access to the laboratory results they needed. "It"s almost like practicing Third World medicine," said Dr. Amanda Garner. "We rely so much on our computers and our fast-world technology that we were almost blinded" 17].

Comair, a subsidiary of Delta Air Lines, canceled all 1,100 of its flights on Christmas Day, 2004, because the computer system that assigns crews to flights stopped running (Figure 7.1). Airline officials said the software could not handle the large number of flight cancellations caused by bad weather on December 23 and 24. About 30,000 travelers in 118 cities were affected by the flight cancellations [18].

In Augusta 2005 the passengers on a Malaysia Airlines flight from Perth, Australia, to Kuala Lumpur, Malaysia, suddenly found themselves on a rollercoaster–like ride seven miles above the Indian Ocean. When the Boeing 777 unexpectedly began a rapid climb, the pilot disconnected the autopilot, but it took him 45 seconds to regain control of the jet. The lane zoomed upward, downward, and then upward a second time before leveling out. After an investigation, Boeing reported that a software error had caused the flight computers to receive faulty information about the plane's speed and acceleration. In addition, another error had caused the flight computers to fail to respond immediately to the pilot's commands [19].

7.3.3 Analysis: E-Retailer Posts Wrong Price, Refuses to Deliver

Amazon.com shut down its British Web site on March 13, 2003, after a software error led it to offer iPaq handheld computers for £7 instead of the correct price of about £275. Before Amazon.com shut down the site, electronic bargain hunters had flocked to Amazon.com's Web site, some of them ordering as many as ten iPaqs [20]. Amazon said customers who ordered at the mistaken price should not expect delivery unless they paid the difference between the advertised price and the actual price. An Amazon.com spokespersons said, "In our Pricing and Availability Policy, we state that where an items correct price is higher than our stated price, we contact the customer before dispatching. Customers will be offered the opportunity either to cancel their order or to place new orders for the item at the correct price" [21].

Was Amazon.com wrong to refuse to fill the orders of the people who bought iPaqs for £7?

Let's analyze the problem from a rule utilitarian point of view. We can imagine a moral rule of the form: "A person or organization wishing to sell a product must always honor the advertised price." What would happen if this rule were universally followed? More

time and efforts would be spent proofreading advertisements, whether printed or electronic. Organizations responsible for publishing the advertisements in newspapers, magazines, and Web sites would also take more care to ensure no errors were introduced. There is a good chance companies would take out insurance policies to guard against the catastrophic losses that could result from a typo. To pay for these additional costs, the prices of the products sold by these companies would be higher. The proposed rule would harm every consumer who ended up paying more for products. The rule would benefit the few consumers who took advantage of misprints to get good deals on certain goods. We conclude the proposed moral rule has more harms than benefits, and Amazon.com did the right thing by refusing the ship the iPaqs.

We could argue, from a Kantian point of view, that the knowledgeable consumers who ordered the iPaqs did something wrong. The correct price was £275; the advertised price was £7. While electronic products may go on sale, retailers simply do not drop the price of their goods by 97.5 percent, even when they are being put on clearance. If consumers understood the advertised price was an error, then they were taking advantage of Amazon.com's stockholders by ordering the iPaq before the error was corrected. They were not acting in "good faith."

7.4 Notable Software Systems Failures

In this section we shift our focus to complicated devices or systems controlled at least in part by computers. An embedded system is a computer used as a component of a larger system. You can find microprocessor-based embedded systems in microwave ovens, thermostats, automobiles, traffic lights, and a myriad of other modern devices. Because computers need software to execute, every embedded system has a software component.

Software is playing an ever-larger role in system functionality [22]. There are several reasons why hardware controllers are being replaced by microprocessors controlled by software. Software controllers are faster. They can perform more sophisticated functions, taking more input data into account. They cost less, use less energy, and do not wear out. Unfortunately, while hardware controllers have a reputation for high reliability, the same cannot be said for their software replacements.

Most embedded systems are also real-time systems: computers that process data from sensors as events occur. The microprocessor that controls the air bags in a modern automobile is a real-time system, because it must instantly react to readings from its sensors and deploy the air bags at the time of a collision. The microprocessor in a cell phone is another example of a real-time system that converts electrical signals into radio waves, and vice versa.

This section contains six examples of computer system failures: the Patriot missile system used in the Gulf War, the Ariane 5 launch vehicle, AT&T's long-distance network, NASA's robot missions to Mars, the automated baggage system at Denver International Airport, and direct recording electronic voting machines. These are all examples of embedded, real-time systems. In every case at least part of the failure was due to errors in the software component of the system. Studying these errors provides important lessons for anyone involved in the development of an embedded system.

7.4.1 Patriot Missile

The Patriot missile system was originally designed by the U.S. Army to shoot down airplanes. In the 1991 Gulf War, the Army put the Patriot missile system to work defending against Scud missiles launched at Israel and Saudi Arabia.

At the end of the Gulf War, the Army claimed the Patriot missile defense system had been 95 percent effective at destroying incoming Scud missiles. Later analyses showed that perhaps as few as 9 percent of the Scuds were actually destroyed by Patriot missiles. As it turns out, many Scuds simply fell apart as they approached their targets: their destruction had nothing at all to do with the Patriot missiles launched at them.

The most significant failure of the Patriot missile system occurred during the night of February 25, 1991, when a Scud missile fired from Iraq hit a U.S. Army barracks in Dhahran, Saudi Arabia, killing 28 soldiers. The Patriot missile battery defending the area never even fired at the incoming Scud.

Mississippi congressman Howard Wolpe asked the General Accounting Office (GAO) to investigate this incident. The GAO report traced the failure of the Patriot system to a software error (Figure 7.2). The missile battery did detect the incoming Scud missile as it came over the horizon. However, in order to prevent the system from responding to false alarms, the computer was programmed to check multiple times for the presence of the missile. The computer predicted the flight path of the incoming missile, directed the radar to focus in on that area, and scanned a segment of the radar signal, called a range gate, for the target. In this case, the program scanned the wrong range gate. Since it did not detect the Scud, it did not fire the Patriot missile.

Why did the program scan the wrong range gate? The tracking system relied upon getting signals from the system clock. These values were stored in a floating-point variable with insufficient precision, resulting in a small mathematical error called a truncation. The longer the system ran, the more these truncation errors added up. The Patriot missile system was designed to operate for only a few hours at a time. However, the system at Dhahran had been in continuous operation for 100 hours. The accumulation of errors led to a difference between the actual time and the computed time of about 0.3433 seconds. Because missiles travel at high speeds, the 0.3433 second error led to a tracking error of 687 meters (about half a mile). That was enough of an error to prevent the battery from locating the Scud in the range gate area [23].

7.4.2 Ariane 5

The Ariane 5 was a satellite launch vehicle designed by the French space agency, the Centré National d'Etudes Spatiales, and the European Space Agency. About 40 seconds into its maiden flight on June 4, 1996, a software error caused the nozzles on the solid boosters and the main rocket engine to swivel to extreme positions. As a result, the rocket veered sharply off course. When the links between the solid boosters and the core stage ruptured, the launch vehicle self-destructed. The rocket carried satellites worth $500 million, which were not insured [24].

A board of inquiry traced the software error to a piece of code that converts a 64-bit floating-point value into a a 16-bit signed integer. The value to be converted exceeded the

maximum value that could be stored in the integer variable, causing an exception to be raised. Unfortunately, there was no exception handing mechanism for this particular exception, so the onboard computers crashed.

The faulty piece of code had been part of the software for the Ariane 4. The 64-bit floating-point value represented the horizontal bias of the launch vehicle, which is related to its horizontal velocity. When the software module was designed, engineers determined that it would be impossible for the horizontal bias to be so large that it could not be stored in a 16-bit signed integer. There was no need for an error handler, because an error could not occur. This code was moved "as is" into the software for the Ariane 5. That proved to be an extremely costly mistake, because the Ariane 5 was faster than the Ariane 4. The original assumptions made by the designers of the software no longer held true [25].

7.4.3 AT&T Long–Distance Network

On the afternoon of January 15, 1990, AT&T's long-distance network suffered a significant disruption of service. About half of the computerized telephone-routing switches crashed, and the remainder of the switches could not handle all of the traffics. As a result of this failure, about 70 million long-distance telephone calls could not be put through, and about 60,000 people lost all telephone service. AT&T lost tens of millions of dollars of revenue. It also lost some of its credibility as a reliable provider of long-distance service.

Investigation by AT&T engineers revealed that the network crash was brought about by a single faulty line of code in an error recovery procedure. The system was designed so that if a server discovered it was in an error state, it would reboot itself, a crude but effective way of "wiping the slate clean." After a switch rebooted itself, it would send an "OK" message to other switches, letting them know it was back on line. The software bug manifested itself when a very busy switch received an "OK" message. Under certain circumstances, handling the "OK" message would cause the busy switch to enter an error state and reboot.

On the afternoon of January 15, 1990, a System 7 switch in New York City detected an error condition and rebooted itself (Figure 7.3). When it came back on line, it broadcast an "OK" message. All of the switches receiving the "OK" messages handled them correctly, except three very busy switches in St. Louis, Detroit, and Atlanta. These switches detected an error condition and rebooted. When they came back up, all of them broadcast "OK" messages across the network, causing other switches to fail in an ever expanding wave.

Every switch failure compounded the problem in two ways. When the switch went down, it pushed more long-distance traffics onto the other switches, making them busier. Chen the switch came back up, it broadcast "OK" messages to these busier switches, causing some of them to fail. Some switches rebooted repeatedly under the barrage of "OK" messages. Within 10 minutes, half of the switches in the AT&T network had failed.

The crash could have been much worse, but AT&T had converted only 80 of its network switches to the System 7 software. It had left System 6 software running on 34 of the switches, "just in case." The System 6 switches did not have the software bug and did not crash [26, 27].

7.4.4 Robot Missions to Mars

NASA designed the $125 million Mars Climate Orbiter to facilitate communications between Earth and automated probes on the surface of Mars, including the Mars Polar Lander. Ironically, the spacecraft was lost because of a miscommunication between twos support teams on Earth.

The Lockheed Martin flight operations team in Colorado designed its software to use English units. Its program output thrust in terms of foot-pounds. The navigation team at the Jet Propulsion Laboratory in California designed its software to use metric units. Its program expected thrust to be input in terms of newtons. One foot-pound, equals 4.45 newtons. On September 23, 1999, the Mars Climate Orbiter neared the Red Planet. When it was time for the spacecraft to fire its engine to enter orbit, the Colorado team supplied thrust information to the California team, which relayed it to the spacecraft. Because of the units mismatch, the navigation team specified 4.45 times too much thrust. The spacecraft flew too close to the surface of Mars and burned up in its atmosphere.

A few months later NASA's Martian program suffered a second catastrophe. The Mars Polar Lander, produced at a cost of $165 million, was supposed to land on the south pole of Mars and provide data that would help scientists understand how the Martian climate has changed over time. On December 3, 1999, NASA lost contact with the Mars Polar Lander. NASA engineers suspect that the system's software got a false signal from the landing gear and shut down the engines 100 feet above the planet's surface.

Tony Spear was project manager of the Mars Pathfinder mission. He said, "It is just as hard to do Mars missions now as it was in the mid-70's. I'm a big believer that software hasn't gone anywhere. Software is the number one problem" [28].

After Spear made this observation, NASA successfully landed two Mars Exploration Rovers on the Red Planet [29]. The rovers, named Opportunity and Spirit, were launched from Earth in June and July of 2003, successfully landing on Mars in January 2004. Mission planners had hoped that each rover would complete a three-month mission, looking for clues that the Martian surface once had enough water to sustain life. The rovers greatly exceeded this goal. Opportunity found evidence of a former saltwater lake, and after five years, both rovers were still operational.

7.4.5 Denver International Airport

As airline passenger traffic strained the capacity of Stapleton International Airport, the City and County of Denver planned the construction of a much larger airport. Stapleton International Airport had earned a reputation for slow baggage handling, and the project planners wanted to ensure the new airport would not suffer from the same problem. They announced an ambitious plan to create a one-of-a-kind, state-of-the-art automated baggage handling system for the Denver International Airport (DIA).

The airport authorities signed a $193 million contract with BAE Automated Systems to design and build the automated baggage-handling system, which consisted of thousands of baggage carts traveling rollercoaster-style on 21 miles of metal tracks. According to the design, agents would label a piece of luggage and put it on a conveyor belt. Computers would route each bag along one or more belts until they reached a cart-loading point, where

each bag would be loaded into its own tub-like cart. Scanners would read the destination information from the suitcase label, and computers would then route each cart along the tracks at 20 miles per hour to the correct unloading point, where each bag would be unloaded onto a conveyor belt and carried to its final destination To monitor the movement of the bags, the system used 56 barcode scanners and 5,000 electric eyes.

There were problems from the outset of the project. The airport design was already done before the baggage handling system was chosen. As a result, the underground tunnels were small and had sharp turns, making it difficult to shoehorn in an automated baggage system. And given its ambitious goals, the project time-line was too short.

However, the most important problem with the automated baggage handler was that the complexity of the system exceeded the ability of the development team to understand it. Here are a few of the problems BAE encountered:

- Luggage carts were misrouted and failed to arrive at their destinations.
- Computers lost track of where the carts were.
- Barcode printers didn't print tags clearly enough to be read by scanners.
- Luggage had to be properly positioned on conveyors in order to load properly.
- Bumpers on the carts interfered with the electric photocells.
- Workers painted over electric eyes or knocked photo sensors out of alignment.
- Light luggage was thrown off rapidly moving carts.
- Luggage got shredded by automated baggage handlers.
- The design did not consider the problem of fairly balancing the number of available carts among all the locations needing them.

BAE attempted to solve these problems one at a time by trial and error, but the system was too complicated to yield to this problem-solving approach. BAE should have been looking at the big picture, trying to find where the specifications for the system were wrong or unattainable.

DIA was supposed to open on October 31, 1993. The opening was delayed repeatedly because the baggage-handling system was not yet operational. Eventually, the Mayor of Denver announced the city would spend $50 million to build a conventional luggage handling system using tugs and carts. (This conventional system actually ended up costing $71 million.) On February 28, 1995, flights to and from the new airport began. However, Concourse A was not open at all. Concourse C opened with 11 airlines using a traditional baggage system. The BAE automated system, far over budget at $311 million, was used only by United Airlines in Concourse B to handle outgoing baggage originating in Denver. United used a traditional system for the rest of its baggage in Concourse B.

The failure of BAE to deliver a working system on time resulted in a 16 month delay in the opening of DIA. This delay cost Denver $1 million a day in interest on bonds and operating costs. As a result, DIA began charging all of the airlines a flight fee of about

$20 per passenger, the highest airport fee in the nation. Airlines passed along this cost to consumers by raising ticket prices of flights going through Denver [30].

While the story of the Denver International Airport is noteworthy because of the large amount of money involved, it is not unusual for software projects to take longer than expected and to cost more than anticipated. In fact, most software projects are not completed on time and on budget. We'll explore this issue in greater detail in Section 7.7.

7.4.6 Tokyo Stock Exchange

December 8, 2005, was the first day that shares of JCom, a recruiting company, were made available to the public on the Tokyo Stock Exchange. That morning, an employee of Mizuho Securities received a call from a customer, who said he wished to sell one share of JCom stock at a price of 610,000 yen. At 9:27 a.m., the Mizuho Securities employee mistakenly entered an order to sell 610,000 shares of JCom at 1 yen per share. When the computer screen displayed a "Beyond price limit" warning, the employee overrode the warning by hitting the Enter key twice, sending the order to the Tokyo Stock Exchange. At 9:28 a.m. the sell order appeared on the Tokyo Stock Exchange's display board. Spotting the mistake, Mizuho Securities attempted to cancel the sell order several times between 9:29 and 9:35 a.m., but these attempts failed because of a bug in the Tokyo Stock Exchange trading program. Mizuho also phoned the Tokyo Stock Exchange, asking the TSE to cancel the sell order, but the Tokyo Stock Exchange refused.

Beginning at 9:35 a.m., Mizuho started to buy back shares of JCom, but it was only able to purchase about a half million shares. More than 96,000 shares had already been purchased by other parties. It was impossible for Mizuho to provide shares to these buyers because JCom only had 14,500 publicly traded shares. Under the terms of a special arrangement brokered by the stock exchange, Mizuho settled these accounts by paying 912,000 yen per share to the buyers. In all, Mizuho Securities lost 40 billion yen ($225 million) buying back shares. When the Tokyo Stock Exchange refused to compensate Mizuho Securities for the loss, Mizuho Securities sued the Tokyo Stock Exchange. The case has not yet been settled.

Eventually the Tokyo Stock Exchange identified the bug that prevented the order from being canceled. The bug had gone undetected for five years because it only occurred when seven different unusual conditions all happened simultaneously [31].

7.4.7 Direct Recording Electronic Voting Machines

Nearly two million ballots were not counted in the 2000 U.S. Presidential election because they registered either no choice or multiples choices. The incredibly close election in Florida was marred by the "hanging chad" and "butterfly ballot" controversies discussed in Section 6.5. To avoid a repeat of these problems, Congress passed and President Bush signed the Help America Vote Act of 2002 (HAVA). HAVA provided money to states to replace punch card voting systems and improve standards for administering elections [32].

Many states used HAVA funds to purchase direct recordings electronic (DRE) voting machines. DRE voting machines allow voters to indicate each of their choices by touching a screen or pressing a button . After all selections have been made, a summary screen displays

the voter's choices. At this point, the voter may either cast the ballot or back up to make changes.

Brazil and India have run national elections using DRE voting machines exclusively [33]. In the United States, a variety of voting technologies are still being use; during the November 2006 general election about one-third of voters cast their ballots on DRE systems. Proponents of DRE voting machines point out the speed and accuracy of machine counting They say the systems are more tamper-resistant than paper ballots, which can be marked by election workers. When the ballots are electronic, it is impossible for precincts to run out of ballots if turnout is higher than expected. In addition, touchscreen voting machines can be programmed to help voters avoid the previously mentioned errors of not choosing a candidate or selecting too many candidates [34].

Some computer experts have spoken out against the conversion to touchscreen voting machines, arguing that they are not necessarily any better than the systems they are replacing. In particular, experts worry about programming errors and the lack of a paper audit trail: a record of the original ballots cast.

Quite a few voting irregularities have been linked to DRE voting machines since 2002. Here is a selection:

- In November 2002 a programming error caused a touchscreen voting machine to fail to record 436 ballots cast in Wake County, North Carolina [35].

- Touchscreen voting machines reported that 144,000 ballots were cast in a 2003 election held in Boone County, Indiana, even though county had only 19,000 registered voters. After a programming error was fixed, the ballots were recounted, producing new results consistent with the number of votes actually cast. However, because there was no paper audit trail, there was no way to know if the new results were correct [36].

- Florida held a special election in January 2004 to determine who would represent State House District 91. When the 10,844 votes were tallied, the voting machines reported that 134 voters had not voted for a candidate, even though that was the only race on the ballot. The winning candidate received 12 more votes than the runner-up. Since the voting machines had no record of the original votes, there was no recount [36].

- In November 2004 initial printouts from all the DRE voting machines in LaPorte County, Indiana, reported exactly 300 votes, disregarding more than 50,000 votes until the problem was sorted out [37].

- In November 2004 a bug in the vote-counting software in DRE voting machines in Guilford County, North Carolina, caused the systems to begin counting backward after they reached a maximum count of 32,767. After the problem was fixed, a recount changed the outcome of two races and gave another 22,000 votes to Presidential candidate John Kerry [38].

- In 2006 some Florida voters had a hard time voting for Democratic candidates on DRE voting machines. After choosing Democrats, these voters discovered that the machine's summary screen replaced some of the Democrats with their Republican

opponents. Some voters had to repeat their votes several times in order for the proper candidate's name to appear on the summary screen [39].

- In a Congressional election held in November 2006 in Florida, more than 18,000 votes cast on DRE voting machines were not recorded. The final tally showed Republican Vern Buchanan beating Democrat Christine Jennings by only 369 votes [40].

Some computer experts are worried about the vulnerability of electronic voting machines to tampering. Finnish security specialist Harri Hursti investigated the memory cartridges used to record votes in Diebold DRE voting machines. (After the polls close, these cartridges are removed from the machines and taken to a central location, where the votes are tallied.) Hursti discovered he could use a readily available agricultural scanning device to change the vote counts without leaving a trace [41].

Computer science professor Herbert Thompson examined the centralized Diebold machine that tallies the votes from the individual DRE voting machines. According to Thompson, the system lacked even a rudimentary authentication mechanism; he was able to access the system's program without a login name or password. By inserting just five lines of code, he successfully switched 5,000 votes from one candidate to another. "I am positive an eighth grader could do this," he said [41].

Without access to the source code to touchscreen systems, there is no way to test how secure they are. The manufacturers of these systems have refused to make the software public, saying the source code is valuable intellectual property: a trade secret. The Open Voting Consortium has criticized the corporate control of elections in the United States and advocates the development of open-source software to make elections "open and transparent" [42].

Critics of touchscreen voting systems say these systems make possible an unprecedented level of election fraud. The old, lever-style mechanical voting machines were susceptible to fraud at the local level. A voting official could enter a voting booth and vote multiple times for a slate of candidates, but the number of extra votes that can be added in any precinct without attracting attention was limited. In contrast, by changing the programming of an electronic voting system, a single person could change votes across thousands of precincts [43].

Supporters of touchscreen voting machines say criticisms of DRE voting machines are overblown. A report by the Pacific Research Institute maintains that DRE voting systems are more secure than traditional paper ballots, which can be tampered with by elections officials. "Open source advocates and paper trail champions want to steer e-voting off a cliff. Rather than demanding Utopian machines and spreading conspiracy theories for political gain, they should refocus their energy in a way that actually helps American voters" [44].

Nevertheless, some states are having second thoughts about DRE voting machines. In May 2007 Florida's legislature voted to replace DRE voting machines with optical scan ballots. Voters select candidates by filling in bubbles next to their names, and optical scanning machines count the marked ballots. This approach leaves a paper audit trail that makes possible manual recounts in disputed elections [45].

7.5 Therac-25

Soon after German physicist Wilhelm Roentgen discovered the xray in 1895, physicians began using radiation to treat cancer. Today, between 50 and 60 percent of cancer patients are treated with radiation, either to destroy cancer cells or relieve pain. Linear accelerators create high-energy electron beams to treat shallow tumors and xray beams to reach deeper tumors.

The Therac-25 linear accelerator was notoriously unreliable. It was not unusual for the system to malfunction 40 times a day. We devote an entire section to telling the story of the Therac-25 because it is a striking example of the harm that can be caused when the safety of a system relies solely upon the quality of its embedded software.

In a 20 month period between June 1985 and January 1987, the Therac-25 administered massive overdoses to six patients, causing the deaths of three of them. While 1987 may seem like the distant past to many of you, it does give us the advantage of 20/20 hindsight. The entire story has been thoroughly researched and documented [46]. Failures of computerized systems continue to this day, but they have not yet been fully played out and analyzed.

7.5.1 Genesis of the Therac-25

Atomic Energy of Canada Limited (AECL) and the French corporation CGR cooperated in the 1970's to build two linear accelerators: the Therac-6 and the Therac-20. Both the Therac-6 and the Therac-20 were modernizations of older CGR linear accelerators. The distinguishing feature of the Therac series was the use of a DEC PDP 11 minicomputer as a "front end." By adding the computer, the linear accelerators were easier to operate. The Therac-6 and the Therac-20 were actually capable of working independently of the PDP 11, and all of their safety features were built into the hardware.

After producing the Therac-20, AECL and CGR went their separate ways. AECL moved ahead with the development and deployment of a next-generation linear accelerator called the Therac-25. Like the Therac-6 and the Therac-20, the Therac-25 made use of a PDP 11. Unlike its predecessor machines, however, AECL designed the PDP 11 to be an integral part of the device; the linear accelerator was incapable of operating without the computer. This design decision enabled AECL to reduce costs by replacing some of the hardware safety features of the Therac-20 with software safety features in the Therac-25.

AECL also decided to reuse some of the Therac-6 and Therac-20 software in the Therac-25. Code reuse saves time and money. Theoretically, "tried and true" software is more reliable than newly written code, but as we shall see, that assumption was invalid in this case.

AECL shipped its first Therac-25 in 1983. In all, it delivered 11 systems in Canada and the United States. The Therac-25 was a large machine that was placed in its own room. Shielding in the walls, ceiling, and floor of the room prevented outsiders from being exposed to radiation. A television camera, microphone, and speaker in the room allowed the technician in an adjoining room to view and communicate with the patient undergoing treatment.

7.5.2 Chronology of Accidents and AECL Responses

MARIETTA, GEORGIA, JUNE 1985

A 61 year-old breast cancer patient was being treated at the Kennestone Regional Oncology Center. After radiation was administered to the area of her collarbone, she complained that she had been burned.

The Kennestone physicist contacted AECL and asked if it was possible that the Therac-25 had failed to diffuse the electron beam. Engineers at AECL replied that this could not happen.

The patient suffered crippling injuries as a result of the overdose, which the physicist later estimated was 75 to 100 times too large. She sued AECL and the hospital in October 1985.

HAMILTON, ONTARIO, JULY 1985

A 40 year-old woman was being treated for cervical cancer at the Ontario Cancer Foundation. When the operator tried to administer the treatment, the machine shut down after five seconds with an error message. According to the display, the linear accelerator had not yet delivered any radiation to the patient. Following standard operating procedure, the operator typed "P" for "proceed." The system shut down in the same way, indicating that the patient had not yet received a dose of radiation. (Recall it was not unusual for the machine to malfunction several dozen times a day.) The operator typed "P" three more times, always with the same result, until the system entered "treatment suspend" mode.

The operator went into the room where the patient was. The patient complained that she had been burned. The lab called in a service technician, who could find nothing wrong with the machine. The clinic reported the malfunction to AECL.

When the patient returned for further treatment three days later, she was hospitalized for a radiation overdose. It was later estimated that she had received between 65 and 85 times the normal dose of radiation. The patient died of cancer in November 1985.

FIRST AECL INVESTIGATION, JULY–SEPTEMBER 1985

After the Ontario overdose, AECL sent out an engineer to investigate. While the engineer was unable to reproduce the overdose, he did uncover design problems related to a microswitch. AECL introduced hardware and software changes to fix the microswitch problem.

YAKIMA, WASHINGTON, DECEMBER 1985

The next documented overdose accident occurred at Yakima Valley Memorial Hospital. A woman receiving a series of radiation treatments developed a strange reddening on her hip after one of the treatments. The inflammation took the form of several parallel stripes. The hospital staff tried to determine the cause of the unusual stripes. They suspected the pattern could have been caused by the slots in the accelerator's blocking trays, but these trays had already been discarded by the time the staff began their investigation. After ruling out other possible causes for the reaction, the staff suspected a radiation overdose and contacted AECL by letter and by phone.

AECL replied in a letter that neither the Therac-25 nor operator error could have produced the described damage. Two pages of the letter explained why it was technically impossible for the Therac-25 to produce an overdose. The letter also claimed that no similar accidents had been reported.

The patient survived, although the overdose scarred her and left her with a mild disability.

TYLER, TEXAS, MARCH 1986

A male patient came to the East Texas Cancer Center (ETCC) for the ninth in a series of radiation treatments for a cancerous tumor on his back. The operator entered the treatment data into the computer. She noticed that she had typed "X" (for xray) instead of "E" (for electron beam). This was a common mistake, because xray treatments are much more common. Being an experienced operator, she quickly fixed her mistake by using the up arrow key to move the cursor back to the appropriate field, changing the "X" to an "E" and moving the cursor back to the bottom of the screen. when the system displayed "beam ready," she typed "B" (for beam on). After a few seconds, the Therac-25 shut down. The console screen contained the message "Malfunction 54" and indicated a "treatment pause," a low-priority problem. The dose monitor showed that the patient had received only 6 units of treatment rather than the desired 202 units. The operator hit the "P" (proceed) key to continue the treatment.

The cancer patient and the operator were in adjoining rooms. Normally a video camera and intercom would enable the operator to monitor her patients. However, at the time of the accident neither system was operational.

The patient had received eight prior treatments, so he knew something was wrong as soon as the ninth treatment began. He was instantly aware of the overdose he felt as if someone had poured hot coffee on his back or given him an electric shock. As he tried to get up from the table, the accelerator delivered its second dose, which hit him in the arm. The operator became aware of the problem when the patient began pounding on the door. He had received between 80 and 125 times the prescribed amount of radiation. He suffered acute pain and steadily lost bodily functions until he died from complications of the overdose five months later.

SECOND AECL INVESTIGATION, MARCH 1986

After the accident, the ETCC shut down its Therac-25 and notified AECL. AECL sent out two engineers to examine the system. Try as they might, they could not reproduce Malfunction 54. They told the physicians it was impossible for the Therac-25 to overdose a patient, and they suggested that the patient may have received an electrical shock due to a fault in the hospital's electrical system.

The ETCC checked out the electrical system and found no problems with it. After double-checking the linear accelerator's calibration, they put the Therac-25 back into service.

TYLER, TEXAS, APRIL 1986

The second Tyler, Texas, accident was virtually a replay of the prior accident at ETCC. The same technician was in control of the Therac-25, and she went through the same process of

entering xray when she meant electron beam, then going back and correcting her mistake. Once again, the machine halted with a Malfunction 54 shortly after she activated the electron beam. This time, however, the intercom was working, and she rushed to the accelerator when she heard the patient moan. There was nothing she could do to help him. The patient had received a massive dose of radiation to his brain, and he died three weeks later.

After the accident, ETCC immediately shut down the Therac-25 and contacted AECL again.

YAKIMA, WASHINGTON, JANUARY 1987

A second patient was severely burned by the Therac-25 at Yakima Valley Memorial Hospital under circumstances almost identical to those of the December 1985 accident. Four days after the treatment, the patient's skin revealed a series of parallel red stripes: the same pattern that had perplexed the radiation staff in the case of the previous patient. This time, the staff members were able to match the burns to the slots in the Therac-25's blocking tray. The patient died three months later.

THERAC25 DECLARED DEFECTIVE, FEBRUARY 1987

On February 10, 1987, the FDA declared the Therac-25 to be defective. In order for the Therac-25 to gain back FDA approval, AECL had to demonstrate how it would make the system safe. Five months later, after five revisions, AECL produced a corrective action plan that met the approval of the FDA. This plan incorporated a variety of hardware interlocks to prevent the machine from delivering overdoses or activating the beam when the turntable was not in the correct position.

7.5.3 Software Errors

In the course of investigating the accidents, AECL discovered a variety of hardware and software problems with the Therac-25. Two of the software errors are examples of race conditions. In a race condition, two or more concurrent tasks share a variable, and the order in which they read or write the value of the variable can affect the behavior of the program. Race conditions are extremely difficult to identify and fix, because usually the two tasks do not interfere with each other and nothing goes wrong. Only in rare conditions will the tasks actually interfere with each other as they manipulate the variable, causing the error to occur. We describe both of these errors to give you some insight into how difficult they are to detect.

The accidents at the ETCC occurred because of a race condition associated with the command screen (Figure 7.5). One task was responsible for handling keyboard input and making changes to the command screen. A second task was responsible for monitoring the command screen for changes and moving the magnets into position. After the operator uses the first task to complete the prescription (1), the second task sees the cursor in the lower right-hand corner of the screen and begins the eight-second process of moving the magnets (2). Meanwhile, the operator sees her mistake. The first task responds to her keystrokes and lets her change the "X" to an "E" (3). She gets the cursor back to the lower right-hand

corner before eight seconds are up (4). Now the second task finishes moving the magnets (5). It sees the cursor in the lower right-hand corner of the screen and incorrectly assumes the screen has not changed. The crucial substitution of electron beam for xray goes unnoticed.

What makes this bug particularly treacherous is that it only occurs with faster, more experienced operators. Slower operators would not be able to complete the edit and get the cursor back to the lower right-hand corner of the screen in only eight seconds. If the cursor happened to be anywhere else on the screen when the magnets stopped moving, the software would check for a screen edit and there would be no overdose. It is ironic that the safety of the system actually decreased as the experience of the operator increased.

Another race condition was responsible for the overdoses at the Yakima Valley Memorial Hospital. It occurred when the machine was putting the electron-beam gun back into position. A variable was supposed to be 0 if the gun was ready to fire. Any other value meant the gun was not ready. As long as the electron beam gun was out of position, one task kept incrementing that variable. Unfortunately, the variable could only store the values from 0 to 255. Incrementing it when it had the value 255 would result in the variable's value rolling over to 0, like a car's odometer.

Nearly every time that the operator hit the SET button when the gun was out of position, the variable was not 0 and the gun did not fire. However, there was a very slight chance that the variable would have just rolled over when the operator hit the SET button. In this case the accelerator would emit a charge, even though the system was not ready.

7.5.4 Postmortem

Let's consider some of the mistakes AECL made in the design, development, and support of this system.

When accidents were reported, AECL focused on identifying and fixings particular software bugs. This approach was too narrow. As Nancy Leveson and Clark Turner point out, "most accidents are system accidents; that is, they stem from complex interactions between various components and activities" [46] . The entire system was broken, not just the software. A strategy of eliminating bugs assumes that at some point the last bug will be eradicated. But as Leveson and Turner write, "There is always another software bug" [46 .

The real problem was that the system was not designed to be failsafe. Good engineering practice dictates that a system should be designed so that no single point of failure will lead to a catastrophe. By relying completely upon software for protection against overdoses, the Therac-25 designers ignored this fundamental engineering principle.

Another flaw in the design of the Therac-25 was its lack of software or hardware devices to detect and report overdoses and shut down the accelerator immediately. Instead, the Therac-25 designers left it up to the patients to report when they had received overdoses.

There are also particular software lessons we can learn from the case of the Therac-25. First, it is very difficult to find software errors in programs where multiple tasks execute at the same time and interact through shared variables. Second, the software design needs to be as simple as possible, and design decisions must be documented to aid in the maintenance of the system. Third, the code must be reasonably documented at the time it is written.

Fourth, reusing code does not always increase the quality of the final product. AECL assumed that by reusing code from the Therac-6 and Therac-20, the software would be more

reliable. After all, the code had been part of systems used by customers for years with no problems. This assumption turned out to be wrong. The earlier codes did contain errors. These errors remained undetected because the earlier machines had hardware interlocks that prevented the computer's erroneous commands from harming patients.

The tragedy was compounded because AECL did not communicate fully with its customers. For example, AECL told the physicists in Washington and Texas that an overdose was impossible, even though AECL had already been sued by the patient in Georgia.

7.5.5 Moral Responsibility of the Therac-25 Team

Should the developers and managers at AECL be held morally responsible for the deaths resulting from the use of the Therac-25 they produced?

In order for a moral agent to be responsible for a harmful event, two conditions must hold:

- *Causal condition*: the actions (or inactions) of the agent must have caused the harm.
- *Mental condition*: the actions (or inactions) must have been intended or willed by the agent.

In this case, the causal condition is easy to establish. The deaths resulted both from the action of AECL employees (creating the therapy machine that administered the overdose) and the inaction of AECL employees (failing to withdraw the machine from service or even inform other users of the machine that there had been overdoses).

What about the second condition? Surely the engineers at AECL did not intend or will to create a machine that would administer lethal overdoses of radiation. However, philosophers also extend the mental condition to include unintended harm if the moral agent's actions were the result of carelessness, recklessness, or negligence. The design team took a number of actions that fall into this category. It constructed a system without hardware interlocks to prevent overdoses or the beam from being activated when the turntable was not in a correct position. The machine had no software or hardware devices to detect an accidental overdose. Management allowed software to be developed without adequate documentation. It presumed the correctness of reused code and failed to test it thoroughly. For these reasons the mental condition holds as well, and we conclude the Therac-25 team at AECL is morally responsible for the deaths caused by the Therac-25 radiation therapy machine.

7.6 Computer Simulations

In the previous section we focused on an unreliable computer-controlled system that delivered lethal doses of radiation to cancer patients, but even systems kept behind the locked doors of a computer room can cause harm. Errors in computer simulations can result in poorly designed products, mediocre science, and bad policy decisions. In this section we review our growing reliance on computer simulations for designing products, understanding our world, and even predicting the future, and we describe ways in which computer modelers validate their simulations. 7.6.1 Uses of Simulation

Computer simulation plays a key role in contemporary science and engineering. There are many reasons why a scientist or engineer may not be able to perform a physical experiment. It may be too expensive or time-consuming, or it may be unethical or impossible to perform. Computer simulations have been used to design nuclear weapons, search for oil, create pharmaceuticals, and design safer, more fuel efficient cars. They have even been used to design consumer products such as disposable diapers [47].

Some computer simulations model past events. For example, when astrophysicists derive theories about the evolution of the universe, they can test them through computer simulations. Recently, computer simulation has demonstrated that a gas disk around a young star can fragment into giant gas planets such as Jupiter [48].

A second use of computer simulations is to understand the world around us. One of the first important uses of computer simulations was to aid in the exploration for oil. Drilling a single well costs millions of dollars, and most drillings result in "dry wells" that produce no revenue. Geologists lay out networks of microphones and set off explosive charges. Computers analyze the echoes received by the microphones to produce graphical representations of underground rock formations. Analyzing these formations helps petroleum engineers select the most promising sites to drill.

Computer simulations are also used to predict the future. Modern weather predictions are based on computer simulations. These predictions become particularly important when people are exposed to extreme weather conditions, such as floods, tornadoes, and hurricanes (Figure 7.7). Every computer simulation has an underlying mathematical model. Faster computers enable scientists and engineers to develop more sophisticated models. Over time, the quality of these models has improved.

Of course, the predictions made by computer simulations can be wrong. In 1972 the Club of Rome, an international think tank based in Germany, published a report called The Limits to Growth. The report predicted that a continued exponential increase in world population would lead to shortages of minerals and farm land, higher food prices, and significant increases in pollution [49] . A year after the report was published, the Arab oil embargo resulted in dramatically higher oil and gasoline prices in Western nations, giving credence to these alarming forecasts. As it turns out, the report's predictions were far too pessimistic. While the population of the earth has indeed increased by more than 80 percent in the past 40 years, the amount of tilled land has barely increased, food and mineral prices have dropped, and pollution is in decline in major Western cities [50

The computer model underlying The Limits to Growth was flawed. It assumed all deposits of essential resources had already been discovered. In actuality, many new deposits of oil and other resources have been found in the past four decades. The model ignored the technological improvements that allow society to decrease its use of resources, such as reducing the demand for oil by improving the fuel efficiency of cars or reducing the demand for silver by replacing conventional photography with digital photography.

7.6.1 Validating Simulations

A computer simulation may produce erroneous results for two fundamentally different reasons. The program may have a bug in it, or the model upon which the program is based may be flawed. Verification is the process of determining if the computer program correctly

implements the model. Validation is the process of determining if the model is an accurate representation of the real system [51]. In this section, we'll focus on the process of validation.

One way to validate a model is to make sure it duplicates the performance of the actual system. For example, automobile and truck manufacturers create computer models of their products. They use these models to see how well vehicles will perform in a variety of crash situations. Crashing an automobile on a computer is faster and much less expensive than crashing an actual car. To validate their models, manufacturers compare the results of crashing an actual vehicle with the results predicted by the computer model (Figure 7.8).

Validating a model that predicts the future can introduce new difficulties. If we are predicting tomorrow's weather, it is reasonable to validate the model by waiting until tomorrow and seeing how well the prediction held up. However, suppose you are a scientist using a global warming model to estimate what the climate will be like 50 years from now. You cannot validate this model by comparing its prediction with reality, because you cannot afford to wait 50 years to see if its prediction came true. However, you can validate the model by using it to predict the present.

Suppose you want to see how well your model predicts events 25 years into the future. You have access to data going back 75 years. You let the model use data at least 25 years old, but you do not let the model see any data collected in the past 25 years. The job of predicting the present, given 25 year-old data, is presumably just as hard as the job of predicting 25 years into the future, given present data. The advantage of predicting the present is that you can use current data to validate the model.

A final way to validate a computer model is to see if it has credibility with experts and decision makers. Ultimately, a model is valuable only if it is believed by those who have the power to use its results to reach a conclusion or make a decision.

7.7 Software Engineering

The field of software engineering grew out of a growing awareness of a "software crisis." In the 1960's computer architects had taken advantage of commercial integrated circuits to design much more powerful mainframe computers. These computers could execute much larger programs than their predecessors. Programmers responded by designing powerful new operating systems and applications. Unfortunately, their programming efforts were plagued by problems. The typical new software system was delivered behind schedule, cost more than expected, did not perform as specified, contained many bugs, and was too hard to modify. The informal, ad hoc methods of programming that worked fine for early software systems broke down when these systems reach a certain level of complexity.

Software engineering is an engineering discipline focused on the production of software, as well as the development of tools, methodologies, and theories supporting software production. Software engineers follow a four-step process to develop a software product:

1. Specification: defining the functions to be performed by the software
2. Development: producing the software that meets the specifications
3. Validation: testing the software

4. Evolution: modifying the software to meet the changing needs of the customer

7.7.1 Specification

The process of specification focuses on determining the requirements of the system and the constraints under which it must operate. Software engineers communicate with the intended users of the system to determine what their needs are They must decide if the software system is feasible given the budget and the schedule requirements of the customer. If a piece of software is going to replace an existing process, the software engineers study the current process to help them understand the functions the software must perform. The software engineers may develop prototypes of the user interface to confirm that the system will meet the user's needs.

The specification process results in a high-level statement of requirements and perhaps a mock-up of the user interface that the users can approve. The software engineers also produce a low-level requirements statement that provides the details needed by those who are going to actually implement the software system.

7.7.2 Development

During the development phase, the software engineers produce a working software system that matches the specifications. The first design is based on a high-level, abstract view of the system. The process of developing the high-level design reveals ambiguities, omissions, or outright errors in the specification. When these mistakes are discovered, the specification must be amended. Fixings mistakes is quicker and less expensive when the design is still at a higher, more abstract level.

Gradually the software engineers add levels of detail to the design. As this is done, the various components of the system become clear. Designers pay particular attention to ensure the interfaces between each component are clearly spelled out. They choose the algorithms to be performed and data structures to be manipulated.

Since the emergence of software engineering as a discipline, a variety of structured design methodologies have been developed. These design methodologies result in the creation of large amounts of design documentation in the form of visual diagrams. Many organizations use computer-assisted software engineering (CASE) tools to support the process of developing and documenting an ever more detailed design.

Another noteworthy improvement in software engineering methodologies
is object-oriented design. In a traditional design, the software system is viewed as a group of functions manipulating a set of shared data structures. In an object-oriented design, the software system is seen as a group of objects passing each other messages. Each object has its own state and manipulates its own data based on the messages it receives.

Object-oriented systems have several advantages over systems constructed in a more traditional way:

1. *Because each object is associated with a particular component of the system, object-oriented designs can be easier to understand.*

More easily understood designs can save time during the programming, testing, and maintenance phases of a software project.

2. *Because each object hides its state and private data from other objects, other objects cannot accidentally modify its data items.*

   The result can be fewer errors like the race conditions described in Section 7.5.

3. *Because objects are independent of each other, it is much easier to reuse components of an object-oriented system*

   A single object definition created for one software system can be copied and inserted into a new software system without bringing along other, unnecessary objects.

When the design has reached a great enough level of detail, software engineers write the actual computer programs implementing the software system. Many different programming languages exist; each language has its strengths and weaknesses. Programmers usually implement object-oriented systems using an object-oriented programming language, such as C++, Java or C#.

7.7.3 Validation

The purpose of validation (also called testing) is to ensure the software satisfies the specification and meets the needs of the user. In some companies, testing is an assignment given to newly hired software engineers, who soon move on to design work after proving their worth. However, good testing requires a great deal of technical skill, and some organizations promote testing as a career path.

Testing software is much harder than testing other engineered artifacts, such as bridges. We know how to construct scale models that we can use to validate our designs. To determine how much weight a model bridge can carry, we can test its response to various loads. The stresses and strains on the members and the deflection of the span change gradually as we add weight, allowing us to experiment with a manageable number of different loading scenarios. Engineers can extrapolate from the data they collect to generate predictions regarding the capabilities of a full-scale bridge. By increasing the size of various components, they can add a substantial margin of error to ensure the completed bridge will not fail.

A computer program is not at all like a bridge. Testing a program with a small problem can reveal the existence of bugs, but it cannot prove that the program will work when it is fed a much larger problem. The response of a computer program to nearly identical data sets may not be continuous. Instead, programs that appear to be working just fine may fail when only a single parameter is changed by a small amount. Yet programmers cannot exhaustively test programs. Since exhaustive testing is impossible, programs can never be completely tested. Software testers strive to put together suites of test cases that exercise all the capabilities of the component or system being validated.

To reduce the complexity of validating a large software system, testing is usually performed in stages. In the first stage, each individual module of the system is tested independently. It is easier to isolate and fix the causes of errors when the number of lines of code is relatively small. After each module has been debugged, modules are combined into larger

subsystems for testing. Eventually, all of the subsystems are combined in the complete system. When an error is detected and a bug is fixed in a particular module, all of the test cases related to the module should be repeated to see if the change that fixed one bug accidentally introduced another bug.

7.7.4 Evolution

Successful software systems evolve over time to meet the changing needs of their users. The evolution of a software system resembles the creation of a software system in many ways Software engineer must understand the needs of the users, assess the strengths and weaknesses of the current system, and design modifications to the software. The same CASE tools used to create a new software system can aid in its evolution. Many of the data sets developed for the original system can be reused when validating the updated system.

7.7.5 Software Quality Is Improving

There is evidence that the field of software engineering is becoming more mature. The Standish Group [53] regularly tracks thousands of IT projects. As recently as 1994, about one-third of all software projects were canceled before completion. About one-half of the projects were completed but had time and/or cost overruns, which were often quire large. About one-sixth of the projects were completed on time and on budget, although the completed systems often had fewer features than original planned. Another survey by the Standish Group in 2006 showed that the probability of a software project being completed on time and on budget had doubled, to about one in three. Only about one-sixth of the software projects surveyed were canceled. As before, about half of the projects were late and/or over budget, although the time and cost overruns were not as large as in the first survey. Overall, the Standish Group reports indicate that were the ability of companies to produce software on time and on budget is improving.

Still, with oily about one in three software projects being completed on time and on budget, the industry has a long way to go. Rapid change is a fact of life in the software industry. In order to stay competitive, companies must release products quickly. Many organizations feel a tension between meeting tight deadlines and strictly following software engineering methodologies.

7.8 Software Warranties

As mentioned earlier, Leveson and Turner state that “there is always another software bug“ [46]. If perfect software is impossible, what kind of warranty should a consumer expect to get from a software company? In this section we survey the software warranties offered by software manufacturers, how these warranties have held up in court, and the debate over new legislation spelling out conditions for licensing computer software in the United States.

7.8.1 Shrinkwrap Warranties

Consumer software is often called shrinkwrap software because of the plastic wrap surrounding the box containing the software and manuals. Not too many years ago, consumer software manufacturers provided no warranty for their products at all. Purchasers had to accept shrinkwrap software "as is." Today, many shrinkwrap software manufacturers, including Microsoft, provide a 90-day replacement or money-back guarantee if the program fails [54]. Here is the wording Microsoft included with its limited warranty for Microsoft Office 2000:

> LIMITED WARRANTY FOR SOFTWARE PRODUCTS ACQUIRE IN THE U.S. AND CANADA. Microsoft warrants that (a) the SOFTWARE PRODUCT will perform substantially in accordance with the accompanying written materials for a period of ninety (90) days from the date of receipt .. .
>
> CUSTOMER REMEDIES. Microsoft's and its suppliers' entire liability and your exclusive remedy shall be, at Microsoft's option, either (a) return of the price paid, if any, or (b) repair or replacement of the SOFTWARE PRODUCT that does not meet Microsoft's Limited Warranty and which is returned to Microsoft with a copy of your receipt.

At least Microsoft is willing to state that its software will actually do more or less what the documentation says it can do. The warranty for Railroad Tycoon, distributed by Gathering of Developers, promises only that you'll be able to install the software:

> LIMITED WARRANTY. Owner warrants that the original Storage Media holding the SOFTWARE is free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of purchase as evidenced by Your receipt. If for any reason You find defects in the Storage Media, or if you are unable to install the SOFTWARE on your home or portable computer, You may return the SOFTWARE and all ACCOMPANYING MATERIALS to the place You obtained it for a full refund. This limited warranty does not apply if You have damaged the SOFTWARE by accident or abuse.

I wonder what would happen if you actually did go back to the store with an opened game and asked for a full refund.

While vendors may be willing to give you a refund if you cannot get their software to install on your computer, they are certainly not going to accept any liability if your business is harmed because their software crashes at the wrong time. Later in the Microsoft Office 2000 Professional warranty, we find these words:

> TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL
> MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE

> USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, MICROSOFT'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS UELA SHALL BE LIMITED TO THE GREATER OF THE AMOUNT ACTUALLY PAID FOR THE SOFTWARE PRODUCT OR U.S. $5.00; PROVIDED, HOWEVER, IF YOU HAVE ENTERED INTO A MICROSOFT SUPPORT SERVICES AGREEMENT, MICROSOFT'S ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT.

Here is even blunter language from the license agreement accompanying Harmonic Visions's Music Ace program:

> WE DO NOT WARRANT THAT THIS SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT ITS OPERATION WILL BE UNINTERRUPTED OR ERRORFREE. WE EXCLUDE AND EXPRESSLY DISCLAIM ALL EXPRESS AND IMPLIED WARRANTIES NOT STATED HEREIN, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

In other words, don't blame us if the program doesn't do what you hoped it would do, or if it crashes all the time, or if it is full of bugs.

7.8.2 Are Software Warranties Enforceable?

How can software manufacturers get away with disclaiming any warranties on their products? It's not clear that they can. Article 2 of the Uniform Commercial Code (UCC) governs the sale of products in the United States. In 1975 Congress passed the Magnuson-Moss Warranty Act. One goal of the act was to prevent manufacturers from putting unfair warranties on products costing more than $25. A second goal was to make it economically feasible for consumers to bring warranty suits by allowing courts to award attorneys' fees. Together, the Magnuson-Moss Warranty Act and Article 2 of the UCC protect the rights of consumers. A computer program is a product. Hence unfair warranties on shrinkwrap software could be in violation of these laws.

An early court case, *StepSaver Data Systems v Wyse Technology and The Software Link* , seemed to affirm the notion that software manufacturers could be held responsible for defects in their products, despite what they put in their warranties. However, two later cases seemed to indicate the opposite. In ProCD v. Zeidenberg, the court ruled that the customer was bound to the license agreement, even if the license agreement does not appear on the outside of the shrinkwrap box. Mortenson v. Timberline Software showed that a warranty disclaiming the manufacturer's liability could hold up in court.

STEPSAVER DATA SYSTEMS V. WYSE TECHNOLOGY AND THE SOFTWARE LINK

StepSaver Data Systems, Inc. sold timesharing computer systems consisting of an IBM PC AT server, Wyse terminals, and an operating system provided by The Software Link, Inc. (TSL). In 19861987 StepSaver purchased and resold 142 copies of the Multilink Advanced operating system provided by TSL.

To purchase the software, StepSaver called TSL and placed an order, then followed up with a purchase order. According to StepSaver, the TSL phone sales representatives said that Multilink was compatible with most DOS applications. The box containing the Multilink software included a licensing agreement in which TSL disclaimed all express and implied warranties.

StepSaver's timesharing systems did not work properly, and the combined efforts of StepSaver, Wyse, and TSL could not fix the problems. StepSaver was sued by twelve of its customers. In turn, StepSaver sued Wyse Technology and TSL.

The Third Circuit of the U.S. Court of Appeals ruled in favor of StepSaver [55]. It based its argument on Article 2 of the UCC. The court held that the original contract between StepSaver and TSL consisted of the purchase order, the invoice, and the oral statements made by TSL representatives on the telephone. The license agreement had additional terms that would have materially altered the contract. However, StepSaver never agreed to these terms.

The court wrote, "In the absence of a party"s express assent to the additional or different terms of the writing, section 2207 [of the UCCA] provides a default rule that the parties intended, as the terms of their agreement, those terms to which both parties have agreed along with any terms implied by the provision of the UCC." The court noted that the president of StepSaver had objected to the terms of the licensing agreement. He had refused to sign a document formalizing the licensing agreement. Even after this, TSL had continued to sell to StepSaver, implying that TSL wanted the business even if the contract did not include the language in the licensing agreement. That is why the court ruled that the purchase order, the invoice, and the oral statements constituted the contract, not the license agreement.

PROCD, INC. V. ZEIDENBERG

ProCD invested more than $10 million to construct a computer database containing information from more than 3,000 telephone directories. ProCD also developed a proprietary technology to compress and encrypt the data. It created an application program enabling users to search the database for records matching criteria they specified. ProCD targeted its product, called SelectPhone, to two different markets: companies interested in generating mailing lists, and individuals interested in finding the phone numbers or addresses of particular people they wanted to call or write. Consumers who wanted SelectPhone for personal use could purchase it for $150; companies paid much more for the right to put the package to commercial use. ProCD included in the consumer version of SelectPhone a license prohibiting the commercial use of the database and program. In addition, the license terms were displayed on the user's computer monitor every time the program was executed.

Matthew Zeidenberg purchased the consumer version of SelectPhone in 1994. He formed a company called Silken Mountain Web Services, Inc., which resold the information in the SelectPhone database. The price it charged was substantially less than the commercial price of SelectPhone. ProCD sued Matthew Zeidenberg for violating the licensing agreement.

At the trial, the defense argued that Zeidenberg could not be held to the terms of the licensing agreement, since they were not printed on the outside of the box containing the software. The U.S. Court of Appeals for the Seventh Circuit ruled in favor of ProCD. Judge Frank Easterbrook wrote, "Shrinkwrap licenses are enforceable unless their terms are objectionable on grounds applicable to contracts in general (for example, if they violate a rule of positive law, or if they are unconscionable)" [56].

MORTENSON V. TIMBERLINE SOFTWARE

M. A. Mortenson Company was a national construction contractor with a regional office in Bellevue, Washington. Timberline Software, Inc. produced software for the construction industry. Mortenson had used software from Timberline for several years. In July 1993 Mortenson purchased eight copies of a bidding package called Precision Bid Analysis.

Timberline's licensing agreement included this paragraph:

> LIMITATION OF REMEDIES AND LIABILITY.
>
> NEITHER TIMBERLINE NOR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION OR DELIVERY OF THE PROGRAMS OR USER MANUALS SHALL BE LIABLE TO YOU FOR ANY DAMAGES OF ANY TIME, INCLUDING BUT NOT LIMITED TO, ANY LOST PROFITS, LOST SAVINGS, LOSS OF ANTICIPATED BENEFITS, OR OTHER INCIDENTAL, OR CONSEQUENTIAL DAMAGES, ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAMS, WHETHER ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT TORT, OR UNDER ANY WARRANTY, OR OTHERWISE, EVEN IF TIMBERLINE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR FOR ANY OTHER CLAIM BY ANY OTHER PARTY. TIMBERLINE'S LIABILITY FOR DAMAGES IN NO EVENT SHALL EXCEED THE LICENSE FEE PAID FOR THE RIGHT TO USE THE PROGRAMS.

In December 1993 Mortenson used Precision Bid Analysis to prepare a bid for the Harborview Medical Center in Seattle. On the day the bid was due, the software malfunctioned. It printed the message "Abort: Cannot find alternate" 19 times. Mortenson continued to use the software and submitted the bid the software produced. After the firm won the contract, Mortenson discovered that its bid was $1.95 million too low.

Mortenson sued Timberline for breach of express and implied warranties. It turns out Timberline had been aware of the bug uncovered by Mortenson since May 1993. Timberline had fixed the bug and already sent a newer version of the program to some of its other customers who had encountered it. It had not sent the improved program to Mortenson. Nevertheless, Timberline argued that the lawsuit be summarily dismissed because the licensing agreement limited the consequential damages that Mortsenson could recover from

Timberline. The King County Superior Court ruled in favor of Timberline. The ruling was upheld by the Washington Court of Appeals and the Supreme Court of the State of Washington [57].

7.8.3 Moral Responsibility of Software Manufacturers

Should producers of shrinkwrap software be held responsible for defects in their programs?

Let's consider the consequences of holding manufacturers of shrinkwrap software liable for damages, such as lost profits, caused by errors encountered by licensees. Currently, manufacturers rely upon consumers to help them identify bugs in their products. If they must find these bugs themselves, they will need to hire many more software testers. The result will be higher prices and longer program development times.

Prudent companies would most likely purchase insurance to protect them from potential lawsuits. This insurance could be very expensive, depending upon the maximum liability to which a company could be held responsible. The cost of the insurance would be passed along to consumers in the form of higher prices.

These changes in the consumer software industry would affect small, startup companies more than large, established firms. The changes would slow the entry of new companies into the field. The result would be a decrease in level of innovation and vitality in the software industry.

Consumers could license software with a higher degree of confidence, knowing that the companies stood by their products. While there might be fewer products available, and their prices would be higher; they would be more reliable.

The result of our utilitarian analysis depends upon how much weight we give to the various consequences. Let's suppose we conclude that software manufacturers should not be held liable for lost profits and other negative consequences arising from errors in their programs, because the harms are greater than the benefits. We may still ask the following question: What are the rights of consumers who license shrinkwrap software?

Consider this hypothetical scenario. A consumer goes to the store, pays $49.95, and brings home a copy of *Incredible Bulk*. The game is usable, but it contains some annoying bugs. The next year the company releases *Incredible Bulk II* If the consumer wants all the latest bug fixes, he needs to buy the second edition. Of course, *Incredible Bulk II* has cool new features. And, as you might expect, some of these features are buggy. Never fear! The bugs will be fixed when *Incredible Bulk III* comes out in 12 months. Is this a fair arrangement?

From a social contract point of view, this arrangement is unfair. Consumers should have the right to be informed of bugs the manufacturer knows about. This knowledge allows a consumer to enter into a contract with his eyes wide open. Ideally, a manufacturer would be open about disclosing the weaknesses in its product. More realistically, consumer organizations can test software products and provide reviews for potential buyers.

If a consumer purchases the right to use product A, and the manufacturer removes defects from product A, the consumer should not have to purchase additional features in order to get access to the fixes to the original product. Manufacturers should make software patches containing bug fixes available on the Web for free downloading by their customers.

Withholding these patches until the next major release of the software is wrong from a social contract point of view.

Summary

Computers are part of larger systems, and ultimately it is the reliability of the entire system that is important. A well-engineered system can tolerate the malfunction of any single component without failing. This chapter has presented many examples of how the computer turned out to be the "weak link" in the system, leading to a failure. These examples provide important lessons for computer scientists and others involved in the design, implementation, and testing of large systems.

Two sources of failure are data-entry errors and data-retrieval errors. While it's easy to focus on a particular mistake made by the person entering or retrieving the data, the system is larger than the individual person. For example, in the case of the 2000 general election in Florida, incorrect records in the computer database disqualified thousands of voters. The data-entry errors caused the voting system to work incorrectly. Sheila Jackson Stossier was arrested by police who confused her with Shirley Jackson. The data-retrieval error caused the criminal justice system to perform incorrectly.

When the topics are software and billing errors, it is easier to identify the system that is failing. For example, when Qwest sent out 14,000 incorrect bills to its cellular phone customers, it's clear that the billing system had failed.

In Sections 7.4 and 7.5, the larger systems were easy to spot. Several embedded systems were dissected to determine the causes of their failures. The program for the Patriot missile's radar tracking system had a subtle flaw: a tiny truncation error occurred every time the clock signal was stored in a floating-point variable. Over a period of 100 hours, all those tiny errors added up to a significant amount, causing the radar system to lose its target. The Ariane 5 blew up because a single assignment statement caused the on board computers to crash. The AT&T long-distance network collapsed because of one faulty line of code.

A well-engineered system does not fail when a single component fails. In the case of hardware, this principle is easier to apply. For example, a jetliner may have three engines. It is designed to be able to fly on any two of the engines, so if a single engine fails, the plane can still fly to the nearest airport and land. When it comes to software, the goal is much harder to meet If we have two computers in the system, that will provide redundancy in case one of the computers has a hardware failure. However, if both computers are running the same software, there is still no software redundancy. A software bug that causes one computer to fail will cause both computers to fail. The partial collapse of the AT&T long-distance network is an example of this phenomenon. All 80 switches containing the latest version of the software failed. Fortunately, 34 switches were running an older version of the software, which prevented a total collapse of AT&T's system.

Imagine what it would take to provide true redundancy in the case of software systems. Should companies maintain two entirely different billing systems so that the bills produced by one system could be double-checked by the other? Should the federal government support two completely different implementations of the National Crime Information Center? These alternatives seem unrealistic. On the other hand, redundancy seems much more feasible when

we look at data-entry and data-retrieval operations. Two different data-entry operators could input records into databases, and the computer could check to make sure the records agreed. This would reduce the chance of bad data being entered into databases in the first place. Two different people could look at the results returned from a computer query, using their own common sense and understanding to see if the output makes sense. A paper audit trail is a practical way to add redundancy to an electronic voting machine.

Chile it may be infeasible to provide redundant software systems, safety-critical systems should never rely completely upon a single piece of software. The Therac-25 overdoses occurred because the system lacked the hardware interlocks of the earlier models.

The stories of computer system failures contain other valuable lessons. The Ariane 5 and Therac-25 failures show it can be dangerous to reuse code. Assumptions that were valid when the code was originally written may no longer be true when the code is reused. Since some of these assumptions may not be documented, the new design team may not have the opportunity to check if these assumptions still hold true in the new system.

The automated baggage system at the Denver International Airport demonstrates the difficulty of debugging a complex system. Cackling one problem at a time, solving it, and moving on to the next problem proved to be a poor approach, because the overall system design had serious flaws. For example, BAE did not even realize that simply getting luggage carts to where they were needed in a fair manner was an incredibly difficult problem. Even if BAE had solved all the low-level technical problems, this high-level problem would have prevented the system from meeting its performance goals during the busiest times.

Finally, systems can fail because of miscommunications among people. The Mars Climate Orbiter is an example of this kind of failure. The software written by the team in Colorado used English units, while the software written by the team in California used metric units. The output of one program was incompatible with the input to the other program, but a poorly specified interface allowed this error to remain undetected until after the spacecraft was destroyed.

Computer simulations are used to perform numerical experiments that lead to new scientific discoveries and help engineers create better products. For this reason, it is important that simulations provide reliable results. Simulations are validated by comparing predicted results with reality. If a simulation is designed to predict future events, it can be validated by giving it data about the past and asking it to predict the present. Finally, simulations are validated when their results are believed by domain experts and policymakers.

The discipline of software engineering emerged from a growing realization of a "software crisis." While small programs can be written in an ad hoc manner, large programs must be carefully constructed if they are to be reliable. Software engineering is the application of engineering methodologies to the creation and evolution of software artifacts. Surveys of the IT industry reveal that more projects are being completed on time and on budget, and fewer projects are being canceled. This may be evidence that software engineering is having a positive impact. However, since most projects are still not completed on time and on budget, there remains much room for improvement. For many companies, shipping a product by a particular date continues to be a higher priority than following a strict software development methodology.

Should software manufacturers be held accountable for the quality of their software, or is a program a completely different kind of product than a socket wrench? An examination

of the software warranties manufacturers include in their licensing agreements reveals that while some vendors refund the purchase price of software that does not meet the needs of the purchaser, they do not want to be held liable for any damages that occur from the use of their software. These warranties seem to fly in the face of Article 2 of the Uniform Commercial Code and the Magnuson-Moss Warranty Act. Some courts have ruled that software manufacturers cannot disclaim liability for consequential damages, but other court decisions imply that software warranties disclaiming liability are enforceable.

References

[1] Jennifer DiSabatino. "Unregulated Databases Hold Personal Data." *Computerworld*, 36(4), January 21, 2002.

[2] Peter G. Neumann. "More on False Arrests." *The Risks Digest*, 1(5), September 4, 1985.

[3] Rodney Hoffman. "NCIC Information Leads to Repeat False Arrest Suit." *The Risks Digest*, 8(71), May 17, 1989.

[4] Ted Bridis. "U.S. Lifts FBI Criminal Database Checks." *Associated Press*, March 25, 2003.

[5] Department of Justice, Federal Bureau of Investigation. "Privacy Act of 1974; Implementation:" *Federal Register*, 68(56), March 24, 2003.

[6] "Computer Glitch Is to Blame for Faculty Bills, Qwest Says." *The Deseret News (Salt Lake Cite, Utah)*, July 24, 2001.

[7] "USDA Changes Livestock PriceReporting Guidelines." *Amarillo (Texas) GlobeNews*, July 24, 2001.

[8] "Software Error Returns Patent Office Mail:" *The New York Times*, August 9, 1996.

[9] "Spelling and Grammar Checkers Add Errors." *Wired News*, March 18, 2003.

[10] D. F. Galletta, A. Durcikova, A. Everard, and B. Jones. "Does SpellChecking Software Need a Warning Label?" *Communications of the ACM*, forthcoming.

[11] Reuters. "Official Trapped in Car After Computer Fails." *NYTimes.com*, May 12, 2003.

[12] Manny Fernandez. "Computer Error Caused Rent Troubles for Public Housing Tenants." *The New York Times*, August 5, 2009.

[13] Ian MacKinnon and Stephen Goodwin. "Ambulance Chief Quits after Patients Die in Computer Failure." *The Independent (London)*, October 29, 1992.

[14] Aaron Lucchetti and Gregory Zuckerman. "Software Glitch Halts Trading on CBOT on April Fool's Day." *The Wall Street Journal*, page C19, April 2, 1998.

[15] "Liffe Glitch Halts All Electronic Trading for a Second Time." *The Wall Street Journal*, May 12, 1999.

[16] "Flights at Japanese Airports Delayed." *Associated Press*, March 1, 2003.

[17] "LA County's Main Hospital Has Computer Breakdown, Delays Ensue." *Associated Press*, April 22, 2003.

[18] "Computer Glitches Shut Down Comair Flights." *Associated Press*, December 26, 2004.

[19] Daniel Michaels and Andy Pasztor. "Flight Check: Incidents Prompt New Scrutiny of Airplane Software Glitches." *The Wall Street Journal*, May 30, 2006.

[20] Robert Fry. "It's a Steal: BargainHunting or Barefaced Robbery?" *The Times (London)*, April 8, 2003.

[21] "Amazon Pulls British Site after iPaq FireSale." *NYtimes.com*, March 19, 2003.

[22] Victor L. Winter and Sourav Bhattacharya. Preface. In *High Integrity Software*, edited by Victor L. Winter and Sourav Bhattacharya. Kluwer Academic Publishers, Boston, MA, 2001.

[23] E. Marshall. "Fatal Error: How Patriot Overlooked a Scud." *Science*, 255(5050):1347, March 13, 1992.

[24] JeanMarc Jézéquel and Bertrand Meyer. "Design by Contract: The Lessons of Ariane." *Computer*, pages 129-130, January 1997.

[25] J. L. Lions. "ARIANE 5: Flight 501 Failure, Report by the Inquiry Board." European Space Agency, July 19, 1996. ravel.esrin.esa.it/docs/esax1819eng.pdf.

[26] Ivars Peterson. "Finding Fault: The Formidable Task of Eradicating Software Bugs." *Science News*, 139, February 16, 1991.

[27] Bruce Sterling. *The Hacker Crackdown: Law and Disorder on the Electronic Frontier*. Bantam Books, New York, NY, 1992.

[28] Jeff Foust. "Why Is Mars So Hard?" *The Space Review*, June 2, 2003. www.thespace review.com.

[29] Jet Propulsion Laboratory, California Institute of Technology. "NASA Facts: Mars Exploration Rover," October 2004. http://marsrover.jpl.nasa.gov.

[30] Richard de Neufville. "The Baggage System at Denver: Prospects and Lessons." *Journal of Air Transport Management*, 1(4):229-236, December 1994.

[31] Tetsuo Tamai. "Social Impact of Information System Failures." *Computer*, June 2009.

[32] Stefan Lovgren. "Are Electronic Voting Machines Reliable?" *National Geographic News*, November 1, 2004. http://news.nationalgeographic.com.

[33] Jarrett Blanc. "Challenging the Norms and Standards of Election Administration: Electronic Voting." In *Challenging the Norms and Standards of Election Administration.* IFES, Washington, DC, 2007.

[34] Sonia Arrison and Vince Vasquez. *Upgrading America's Ballot Box: The Rise of E-Voting*, 2nd edition. Pacific Research Institute, San Francisco, CA, 2006.

[35] "Electronic Ballots Fail to Win Over Wake Voters, Election Officials; Machines Provide Improper Vote Count at Two Locations," *WRALTV* (Raleigh-Durham, NC), November 2,

2002.

[36] Barbara Simons. "Electronic Voting Systems: the Good, the Bad, and the Stupid." *RFID* 2(7), October 2004.

[37] William Rivers Pitt. "Worse than 2000: Tuesday's Electoral Disaster" (editorial). *truthout*, November 8, 2004. www.truthout.org.

[38] Mark Johnson. "Winner So Far: Confusion." *The Charlotte Observer*, November 5, 2004.

[39] Charles Rabin and Darran Simon. "Glitches Cited in Early Voting: Early Voters Are Urged to Cast Their Ballots with Care Following Scattered Reports of Problems with Heavily Used Machines." *The Miami Herald*, October 28, 2006.

[40] "Florida Candidate Disputes Election Results:" *CNN.com*, December 20, 2006.

[41] Marc Caputo and Gary Fineout. "New Tests Fuel Doubts about Vote Machines." *The Miami Herald*, December 15, 2005.

[42] "Transparent Liberty, Accountable Election Systems" (leaflet). The Open Voting Consortium. www.openvoting.org.

[43] Cheryl Gerber. "Voting 2.0." *Chronogram*, January 2006.

[44] "More E-Voting Red Tape Threatens Ballot Booth Benefits, New Study Says" (press release). Pacific Research Institute, October 31, 2006.

[45] Terry Aguayo and Christine Jordan Sexton. "Florida Acts to Eliminate Touch-Screen Voting System." *The New York Times*, May 4, 2007.

[46] Nancy Leveson and Clark Turner. "An Investigation of the Therac-25 Accidents." *Computer*, 26(7):18-41,1993.

[47] William J. Kauffman III and Larry L. Smarr. *Supercomputing and the Transformation of Science*. Scientific American Library, New York, NY, 1993.

[48] Lucio Mayer, Tom Quinn, James Wadsley, and Joachim Stadel. "Forming Giant Planets via Fragmentation of Protoplanetary Disks:" *Science*, November 29, 2002.

[49] Donnella H. Meadows, Dennis I. Meadows, Jorgen Randers and William W. Behrens III. *The Limits To Growth*,. Universe Books, New York, NY, 1972.

[50] Bjorn Lomborg and Olivier Rubin. "Limits to Growth." *Foreign Policy*, October/ November 2002.

[51] G. S. Fishman and P. J. Kiviat. "The Statistics of Discrete Event Simulation:" *Simulation*, 10:185-195, 1968.

[52] Ian Sommerville. *Software Engineering*. 6th ed. Addison-Wesley, Harlow, England, 2001.

[53] David Rubinstein. "Standish Group Report: There's Less Development Chaos Today." *SD Times*, March 1, 2007. http://sdtimes.com.

[54] Scot Petersen. "Taking the Rap for Bad Software:" *PC Week*, page 29, February 28,

2000.

[55] United States Court of Appeals for the Third Circuit. *StepSaver Data Systems, Inc. v. Wyse Technology and The Software Link, Inc.*, 1991.939 F. 2d 91.

[56] United States Court of Appeals for the Seventh Circuit. *ProCD, Inc., v. Matthew Zeidenberg and Silken Mountain Web Services, Inc., Appeal from the United States District Court for the Western District of Wisconsin*, 1996.96-1139.

[57] Supreme Court of the State of Washington. *M.A. Mortenson Co. v. Timberline Software Corp., et al. Opinion*, 2000.

[58] "Texans Get Soaked:" *IEEE Software*, page 114, September/October 1997.

[59] Elizabeth Hays. "L Trains Do Compute." *New York Daily News*, January 18, 2004.