Reuse of software: the Ariane 5 rocket and “No Fly” lists

Less than 40 seconds after the first launch of France’s Ariane 5 rocket, the rocket veered off course and was destroyed as a safety precaution. The rocket and the satellites it was carrying cost approximately $500 million. A software error caused the failure. The Ariane 5 used some software designed for the earlier, successful Ariane 4. The software included a module that ran for about a minute after initiation of a launch on the Ariane 4. It did not have to run after takeoff of the Ariane 5, but a decision was made to avoid introducing new errors by making changes in a module that operated well in Ariane 4. This module did calculations related to velocity. The Ariane 5 travels faster than the Ariane 4 after takeoff. The calculations produced numbers bigger than the program could handle (an “overflow” in technical jargon), causing the system to halt.

A woman named Jan Adams, and many other people with first initial J and last name Adams, were flagged as possible terrorists when they tried to board an airplane. The name “Joseph Adams” is on a “No Fly” list of suspected terrorists (and other people considered safety threats) that the Transportation Security Agency had given to the airlines. To compare passenger names with those on the “No Fly” list, some airlines used old software and strategies designed to help ticket agents quickly locate a passenger’s reservation record (e.g., if the passenger calls in with a question or to make a change). The software searches quickly and “casts a wide net.” That is, it finds any possible match, which a sales agent can then verify. In the intended applications for the software, there is no inconvenience to anyone if the program presents the agent with a few potential matches of similar names. In the context of tagging people as possible terrorists, a person mistakenly “matched” will likely undergo questioning and extra luggage and body searches by security agents.

Do these examples tell us that we should not reuse software? One of the goals of programming paradigms such as object-oriented code is to make software elements that can be widely used, thus saving time and effort. Reuse of working software should also increase safety and reliability. After all, it has undergone field testing in a real, operational environment; we know it works. At least, we think it works. The critical point is that it works in a different environment. It is essential to reexamine the specifications and design of the software, consider implications and risks for the new environment, and retest the software for the new use.
and devices that increase the safety of surgeries. Yet one of the classic case studies of a deadly software failure is a medical device: a radiation treatment machine.

The Therac-25 was a software-controlled radiation-therapy machine used to treat people with cancer. Between 1985 and 1987, Therac-25 machines at four medical centers gave massive overdoses of radiation to six patients. In some cases, the operator repeated an overdose because the machine's display indicated that it had given no dose. Medical personnel later estimated that some patients received more than 100 times the intended dose. These incidents caused severe and painful injuries and the deaths of three patients.

Why is it important to study a case as old as this? To avoid repeating the errors. Medical physicists operating a different radiation-treatment machine in Panama in 2000 tried to circumvent a limitation in the software in an attempt to provide more shielding for patients. Their actions caused dosage miscalculations. Twenty-eight patients received overdoses of radiation, and several died. It seems that dramatic lessons need repetition with each new generation.

What went wrong with the Therac-25?

Studies of the Therac-25 incidents showed that many factors contributed to the injuries and deaths. The factors include lapses in good safety design, insufficient testing, bugs in the software that controlled the machines, and an inadequate system of reporting and investigating the accidents. (Articles by computer scientists Nancy Leveson and Clark Turner and by Jonathan Jacky are the main sources for this discussion.)

To understand the discussion of the problems, it will help to know a little about the machine. The Therac-25 is a dual-mode machine. That is, it can generate an electron beam or an x-ray photon beam. The type of beam needed depends on the tumor being treated. The machine's linear accelerator produces a high-energy electron beam (25 million electron volts) that is dangerous. Patients must not be exposed to the raw beam. A computer monitors and controls movement of a turntable that holds three sets of devices. Depending on the intended treatment, the machine rotates a different set of devices in front of the beam to spread it and make it safe. It is essential that the proper protective device be in place when the electron beam is on. A third position of the turntable uses a light beam instead of the electron beam to help the operator position the beam precisely in the correct place on the patient's body.

8.2.2 Software and Design Problems

Design flaws

The Therac-25 followed earlier machines called the Therac-6 and Therac-20. It differed from them in that it was fully computer controlled. The older machines had hardware safety interlock mechanisms, independent of the computer, that prevented the beam from firing in unsafe conditions. The design of the Therac-25 eliminated many of these hardware safety features. The Therac-25 reused some software from the Therac-20 and Therac-6. The developers apparently assumed the software functioned correctly. This
assumption was wrong. When new operators used the Therac-20, there were frequent
shutdowns and blown fuses, but no overdoses. The Therac-20 software had bugs, but the
hardware safety mechanisms were doing their job. Either the manufacturers did not know
of the problems with the Therac-20, or they completely missed the serious implications.

The Therac-25 malfunctioned frequently. One facility said there were sometimes 40
dose-rate malfunctions in a day, generally underdoses. Thus, operators became used to
error messages appearing often, with no indication that there might be safety hazards.

There were a number of weaknesses in the design of the operator interface. The error
messages that appeared on the display were simply error numbers or obscure messages
("Malfunction 54" or "H-tilt"). This was not unusual for early computer programs when
computers had much less memory and mass storage than they have now. One had to
look up each error number in a manual for more explanation. The operator's manual
for the Therac-25, however, did not include an explanation of the error messages. The
maintenance manual did not explain them either. The machine distinguished between
errors by the amount of effort needed to continue operation. For certain error conditions,
the machine paused, and the operator could proceed (turn on the electron beam) by
pressing one key. For other kinds of errors, the machine suspended operation and had to be
completely reset. One would presume that the machine would allow one-key resumption
only after minor, non-safety-related errors. Yet one-key resumption occurred in some of
the accidents in which patients received multiple overdoses.

Atomic Energy of Canada, Ltd. (AECL), a Canadian government corporation, man-
ufactured the Therac-25. Investigators studying the accidents found that AECL produced
very little documentation concerning the software specifications or the testing plan dur-
ing development of the program. Although AECL claimed that they tested the machine
extensively, it appeared that the test plan was inadequate.

Bugs

Investigators were able to trace some of the overdoses to two specific software errors.
Because many readers of this book are computer science students, I will describe the bugs.
These descriptions illustrate the importance of using good programming techniques.
Because some readers have little or no programming knowledge, I will simplify the
descriptions.

After the operator entered treatment parameters at a control console, a software
procedure called Set-Up Test performed a variety of checks to be sure the machine was
in the correct position, and so on. If anything was not ready, this procedure scheduled
itself to rerun the checks. (The system might simply have to wait for the turntable to
move into place.) The Set-Up Test procedure can run several hundred times while setting
up for one treatment. A flag variable indicated whether a specific device on the machine
was in the correct position. A zero value meant the device was ready; a nonzero value
meant it must be checked. To ensure that the device was checked, each time the Set-Up
Test procedure ran, it incremented the variable to make it nonzero. The problem was
that the flag variable was stored in one byte. After the 256th call to the routine, the flag overflowed and showed a value of zero. (If you are not familiar with programming, think of this as an automobile’s odometer rolling over to zero after reaching the highest number it can show.) If everything else happened to be ready at that point, the program did not check the device position, and the treatment could proceed. Investigators believe that in some of the accidents, this bug allowed the electron beam to be on when the turntable was positioned for use of the light beam, and there was no protective device in place to attenuate the beam.

Part of the tragedy in this case is that the error was such a simple one, with a simple correction. No good student programmer should have made this error. The solution is to set the flag variable to a fixed value, say 1, rather than incrementing it, to indicate that the device needs checking.

Other bugs caused the machine to ignore changes or corrections made by the operator at the console. When the operator typed in all the necessary information for a treatment, the program began moving various devices into place. This process could take several seconds. The software checked for editing of the input by the operator during this time and restarted the set-up if it detected editing. However, because of bugs in this section of the program, some parts of the program learned of the edited information while others did not. This led to machine settings that were incorrect and inconsistent with safe treatment. According to the later investigation by the Food and Drug Administration (FDA), there appeared to be no consistency checks in the program. The error was most likely to occur with an experienced operator who was quick at editing input.

In a real-time, multitasking system that controls physical machinery while an operator enters—and might modify—input, there are many complex factors that can contribute to subtle, intermittent, and hard-to-detect bugs. Programmers working on such systems must learn to be aware of the potential problems and to use good programming practices to avoid them.

### 8.2.3 Why So Many Incidents?

There were six known Therac-25 overdoses. You may wonder why hospitals and clinics continued to use the machine after the first one.

The Therac-25 had been in service for up to two years at some clinics. Medical facilities did not immediately pull it from service after the first few accidents because they did not know immediately that it caused the injuries. Medical staff members considered various other explanations. The staff at the site of the first incident said that one reason they were not certain of the source of the patient’s injuries was that they had never seen such a massive radiation overdose before. They questioned the manufacturer about the possibility of overdoses, but the company responded (after the first, third, and fourth accidents) that the machine could not have caused the patient injuries. According to the Leveson and
8.2 Case Study: The Therac-25

Turner investigative report, they also told the facilities that there had been no similar cases of injuries.

After the second accident, AECL investigated and found several problems related to the turntable (not including any of the ones we described). They made some changes in the system and recommended operational changes. They declared that they had improved the safety of the machine by five orders of magnitude, although they told the FDA that they were not certain of the exact cause of the accident. That is, they did not know whether they had found the problem that caused the accident or just other problems. In making decisions about continued use of the machines, the hospitals and clinics had to consider the costs of removing the expensive machine from service (in lost income and loss of treatment for patients who needed it), the uncertainty about whether the machine was the cause of the injuries, and, later, when that was clear, the manufacturer’s assurances that they had solved the problem.

A Canadian government agency and some hospitals using the Therac-25 made recommendations for many more changes to enhance safety; they were not implemented. After the fifth accident, the FDA declared the machine defective and ordered AECL to inform users of the problems. The FDA and AECL spent about a year (during which the sixth accident occurred) negotiating about changes in the machine. The final plan included more than two dozen changes. They eventually installed the critical hardware safety interlocks, and most of the machines remained in use after that with no new incidents of overdoses.26

Overconfidence

In the first overdose incident, when the patient told the machine operator that the machine had “burned” her, the operator told her that was impossible. This was one of many indications that the makers and some users of the Therac-25 were overconfident about the safety of the system. The most obvious and critical indication of overconfidence in the software was the decision to eliminate the hardware safety mechanisms. A safety analysis of the machine done by AECL years before the accidents suggests that they did not expect significant problems from software errors. In one case where a clinic added its own hardware safety features to the machine, AECL told them it was not necessary. (None of the accidents occurred at that facility.)

The hospitals using the machine assumed that it worked safely, an understandable assumption. Some of their actions, though, suggest overconfidence, or at least practices that they should have avoided. For example, operators ignored error messages because the machine produced so many of them. A camera in the treatment room and an intercom system enabled the operator to monitor the treatment and communicate with the patient. (The operator uses a console outside the shielded treatment room.) On the day of an accident at one facility, neither the video monitor nor the intercom was functioning. The operator did not see or hear the patient try to get up after an overdose. He received a second
overdose before he reached the door and pounded on it. This facility had successfully treated more than 500 patients with the machine before this incident.

8.2.4 Observations and Perspective

From design decisions all the way to responding to the overdose accidents, the manufacturer of the Therac-25 did a poor job. The number and pattern of problems in this case, and the way they were handled, suggest serious irresponsibility. This case illustrates many of the things that a responsible, ethical software developer should not do. It illustrates the importance of following good procedures in software development. It is a stark reminder of the consequences of carelessness, cutting corners, unprofessional work, and attempts to avoid responsibility. It reminds us that a complex system can work correctly hundreds of times with a bug that shows up only in unusual circumstances—hence the importance of always following good safety procedures in operation of potentially dangerous equipment. This case also illustrates the importance of individual initiative and responsibility. Recall that some facilities installed hardware safety devices on their Therac-25 machines. They recognized the risks and took action to reduce them. The hospital physicist at one of the facilities where the Therac-25 overdosed patients spent many hours working with the machine to try to reproduce the conditions under which the overdoses occurred. With little support or information from the manufacturer, he was able to figure out the cause of some of the malfunctions.

To emphasize that safety requires more than bug-free code, we consider failures and accidents involving other radiation treatment systems. Three patients received overdoses in one day at a London hospital in 1966 when safety controls failed. Twenty-four patients received overdoses from a malfunctioning machine at a Spanish hospital in 1991; three patients died. Neither of these machines had computer controls. To7 Two news reporters reviewed more than 4000 cases of radiation overdoses reported to the U.S. government. Here are a few of the overdose incidents they describe. A technician started a treatment, then left the patient for 10–15 minutes to attend an office party. A technician failed to carefully check the prescribed treatment time. A technician failed to measure the radioactive drugs administered; she just used what looked like the right amount. In at least two cases, technicians confused microcuries and millicuries.* The underlying problems were carelessness, lack of appreciation for the risk involved, poor training, and lack of sufficient penalty to encourage better practices. (In most cases, the medical facilities paid small fines or none at all.)28

Most of the incidents we just described occurred in systems without computers. For some, a good computer system might have prevented the problem. Many could have occurred whether or not the treatment system was controlled by a computer. These

* A curie is a measure of radioactivity. A millicurie is one thousand times as much as a microcurie.
examples remind us that individual and management responsibility, good training, and accountability are important no matter what technology we use.

8.3 Increasing Reliability and Safety

Success actually requires avoiding many separate possible causes of failure.

—Jared Diamond

8.3.1 Professional Techniques

The New York Stock Exchange installed a $2 billion system with hundreds of computers, 200 miles of fiber-optic cable, 8000 telephone circuits, and 300 data routers. The exchange managers prepared for spikes in trading by testing the system on triple and quadruple the normal trading volume. On one day, the exchange processed 76% more trades than the previous record. The system handled the sales without errors or delays. We have been describing failures throughout this chapter. Many large, complex computer systems work extremely well. We rely on them daily. How can we design, build, and operate systems that are likely to function well?

To produce good systems, we must use good software engineering techniques at all stages of development, including specifications, design, implementation, documentation, and testing. There is a wide range between poor work and good work, as there is in virtually any field. Professionals, both programmers and managers, have the responsibility to study and use the professional techniques and tools that are available and to follow the procedures and guidelines established in the various relevant codes of ethics and professional practices. (The Software Engineering Code of Ethics and Professional Practice and the ACM Code of Ethics and Professional Conduct, in Appendix A, are two important sets of general guidelines for the latter.)

Management and communications

Management experts use the term high reliability organization (HRO) for an organization (business or government) that operates in difficult environments, often with complex technology, where failures can have extreme consequences (for example, air traffic control, nuclear power plants). Researchers have identified characteristics of HROs that perform extremely well. These characteristics can improve software and computer systems in both critical and less critical applications. One characteristic is “preoccupation with failure.” That means always assuming something unexpected can go wrong—not just planning, designing, and programming for all problems the team can foresee, but always being aware that they might miss something. Preoccupation with failure includes being alert to cues that might indicate an error. It includes fully analyzing near failures (rather than assuming