Integrating Ethics Into a Computing Curriculm: A Case Study of the Therac-25

Chuck Huff[1] and Richard Brown[2]

Abstract

Almost all computer ethics courses use cases to some extent. We show how we have integrated detailed historical cases into ethical reflection throughout the computer science major, and in particular in the course *Ethical Issues in Software Design*. We then present a particular case, that of a radiation therapy machine in the mid 1980s that killed several people with massive overdoses of radiation. The overdoses were caused by poor software design. To understand the failings of the software design process we present significant technical and social detail of this case and show how this detail helped to make the case effective in teaching ethical issues.

A chapter in Atsushi Akera & William Aspray (2004). Using history to teach computer science and related disciplines. (pp. 255-278). Washington DC: Computing Research Association.

[1] Chuck Huff is Professor of Psychology at St. Olaf College. He has published research in the areas of moral reasoning, computing and education, gender and computing, social aspects of electronic interaction, and ethics in computing. He recently returned from sabbatical doing empirical research on the moral development of computer professionals in collaboration with the *Centre for Computing and Social Responsibility* at Demontfort University in Leicester, UK.

[2] Richard Brown is Associate Professor and Director of Computer Science at St. Olaf College. He recently designed the college's computer science major in a way that integrates ethical issues throughout the curriculum. He is writing an introductory textbook, *Principles of Computer Science,* that emphasizes recurring concepts in computing and is structured around three programming paradigms: functional, imperative, and object-oriented.

Computer ethics courses have now been established enough in the field to have a distinctive content, several attempts at a recommended curriculum[3] and to have generated several distinct approaches to class organization and practice. Almost all courses uses cases to some extent in their approach but some are more intentionally case-based and other are driven more by other pedagogical approaches. Some texts use a "controversies in the field" approach[4] with their primary aim that of exposing students to the arguments on the different sides of the controversy. The most common approach[5] is topic based, covering the standard topics such as privacy, liability, and intellectual property that appear in the recommended curricula. A minority approach is a social issues based organization[6] concentrating less on ethical analysis and more on social analysis. In this paper we want to present a case based approach that emphasizes the connection of ethical reflection and analysis to the practice of good software development. Central to this approach is the use of detailed historical cases. These cases can present the issues that software developers face in all their complexity while emphasizing practice in the skills they will need to deal with that complexity. Cases of the complexity we present here are not readily available, though many of the ones presented by Spinello[7] and the cases available on computingcases.org can be used for this approach.

Cases can be used to accomplish all the goals for ethical education in computing:[8]

•*Mastering a knowledge of basic facts and understanding and applying basic and intermediate ethical concepts.* Even simple cases can help student learn appropriate distinctions

---

[3] See Chuck Huff and C. Dianne Martin, "Computing Consequences: A framework for teaching ethical computing," *Communications of the ACM, 38,* (December, 1995) : 75-84. Also the ACM/IEEE Computing Curricula 2001 at http://www.computer.org/education/cc2001/steelman/cc2001/index.htm

[4] Rob Kling, ed. *Computerization and Controversy: Value Conflicts and Social Choices*, 2nd ed. 1996.

[5] Deborah G. Johnson, Computer Ethics, 2001 or Herman T. Tavani, *Ethics and Technology*, 2004.

[6] Chuck Huff and Thomas Finholt, *Social Issues in Computing: Putting Computing in its Place*, 1994.

[7] Richard A. Spinello, *Case Studies in Information and Computer Ethics*, 1997.

[8] Chuck Huff and William Frey, "Moral Pedagogy and Practical Ethics." *Science and Engineering Ethics*. In press.

between various basic concepts (e.g. consequentialist vs deontological approaches). They can also help students learn intermediate concepts (e.g. intellectual property, kinds of privacy, etc.). They do this while making it evident that the concepts are useful in real life, thus providing motivation for students. They also provide opportunities for students to practice skills like applying code of ethics and making and listening to ethical arguments.

- *Practicing moral imagination.* Cases, particularly complex ones, allow students to gain skill in the moral imagination need to detect when an ethical issue is present, to understand how software affects people, and to take the perspective of others. They also allow students practice in the moral imagination needed to construct creative solution to ethical problems (which involves trading off values, understanding organization politics, taking other's perspective, etc.)
- *Encouraging adoption of professional standards into the professional self-concept.* Cases that show the intimate connection between technical and ethical issues (like the Therac-25 case we present here) allow students to see that ethical issues cannot be separated from the technical issues they care about, and so encourage students to see ethical reflection as a part of their professional responsibility.
- *Building ethical community.* Case discussion is an ideal venue for getting students used to talking with each other and influencing each other about ethical issues.

More than a traditional lecture or didactic approach, cases provide the opportunity for the manifestly social sorts of activities that the work in moral psychology suggests are important in acquiring knowledge base and skills for moral judgment[9]. When discussing, debating, constructing, or trying cases, students actively disagree with each other and influence each other's intuitions about what is and is not the right thing to do. They are not only learning explicit knowledge (like intermediate concepts, or ethical skills) but also tacit knowledge, such as when to be suspicious, how to respond to disagreement, etc. In this way case-based

[9] Ibid.

pedagogy engages more of the systems/levels/processes of moral reasoning than do more passive instructional methods.

Taking various positions in a case can help students learn moral sensitivity and practice identifying moral issues from different perspectives. Moral imagination can be extended by wrestling with the complexities of a case and a variety of solutions. Simple self-reflection is often ineffectual because it tends to produce only rationalizations for existing moral intuitions; it does not provide reasons strong enough to motivate self-reform .[10] But moral intuition can be influenced and shaped through our interactions with others. Hence, case discussion provides an ideal tool for promoting this interaction. Students put forth unexamined opinions and find that they fail to convince others; they listen to the views and reasons offered by others and, through this, learn to reexamine their own views from a different, broader perspective. Consequently, case discussion develops the habit of putting forth rational arguments for one's moral views.

Advantages of Historical Cases

But why do historical cases instead of the more neatly packaged cases philosophers tend to use? Partly because the goals of a computer ethics course are different from the standard philosophical ethics course. Computer ethics courses emphasize the skills of detecting and framing ethical issues, taking the perspective of the other, making technically-based ethical arguments, and constructing and proposing solutions to ethical difficulties. All of this needs to be done within the complexity of the sort of ethical situations that students are likely to face in industry. Historical cases provide the needed complexity.

But they also provide the uncertainties that provide students with practice constructing solutions to deal with contingencies. You can never know all you think you need to know, even in a very detailed case. And in a detailed case it is clear from the detail that decision-makers at the time were operating under uncertainty but still had to make decisions.

---

[10] Jonathon Haidt, “The Emotional Dog and Its Rational Tail: A Social Intuitionist Approach to Moral Judgment” *Psychological Review.* 108 (2001) : 814-834.

This connection with the real difficulties of professionals in the field is another benefit that historical cases offer. Students are motivated when they discover that they are learning knowledge and skills that will help them face similar difficulties when they find themselves in similar circumstances. Thus, historical cases need to be chosen and presented in a way that makes this connection to the difficulties students are likely to face in the modern workplace.

Finally, historical cases are helpful because they provide real outcomes of real decisions. Students can compare their own skills and knowledge to that of the decision makers in the case, and see the actual effects of at least the one options the decision maker chose. In what follows, we present a classic historical case in the sort of detail needed for extensive class use and we discuss how we have used the case in class.

The Therac-25 Case:

Therac-25 was a new generation medical linear accelerator introduced in 1983 for treating cancer. It incorporated the most recent computer control equipment. Therac-25's computerization made the laborious process of machine setup much easier for operators, and thus allowed them to spend minimal time in setting up the equipment. In addition to making setup easier, the computer also monitored the machine for safety. With the advent of computer control, hardware based safety mechanisms were transferred to the software. Hospitals were told that the Therac-25 medical linear accelerator had "so many safety mechanisms" that it was "virtually impossible" to overdose a patient. As it turned out, the computerized safety monitoring was not sufficient, and there were six documented cases of significant overdoses of radiation, three resulting in severe injury and three resulting in death from radiation sickness.

The standard reference on the Therac-25 accident is the extensive paper by Leveson & Turner.[11] This paper documents the history of the development of Therac through several generations, the introduction of the machine and the accidents it caused. It is particularly

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

helpful in documenting the safety analysis done by the company, and in showing in detail how the computer control was accomplished, even to including pseudo-code to document the software errors that produced several of the accidents. With support from the National Science Foundation (DUE-9972280 and DUE-9980786), we have been able to add to this analysis from legal documents we were able to collect, interviews we did with relevant actors (e.g. an operator of radiation therapy machines, a medical physicist), and other resources. Much of this is available on the computingcases.org website, but some is unique to this article.

A History of the Introduction and Shut Down of Therac-25

Therac-25 was released on the market in 1983 by Atomic Energy Canada, Limited (AECL). In 1987, treatments with the eleven machines then in operation was suspended. Those machines were refitted with the safety devices required by the FDA and remained in service. No more accidents were reported from these machines. At about that time, the division of AECL that designed and manufactured Therac-25 became an independent company.

The major innovations of Therac-25 were the double pass accelerator (allowing a more powerful accelerator to be fitted into a small space, at less cost) and the move to more complete computer control. The move to computer control allowed operators to set up the machine more quickly, giving them more time to speak with patients and making it possible to treat more patients in a day. Along with the move to computer control, most of the safety checks for the operation of the machine were moved to software and hardware safety interlocks removed.

> *The accident history of Therac-25.* In July of 1985, AECL was notified that a patient in Hamilton, Ontario had been overdosed. AECL sent a service engineer to the site to investigate. AECL also informed the United States Food and Drug Administration (FDA), and the Canadian Radiation Protection Board (CRPB) of the problem. In addition they notified all users of the problem and issued instructions that operators should visually confirm hardware settings before each treatment. AECL could not reproduce the malfunction, but its engineers suspected that a hardware failure in a microswitch was at

fault. They redesigned the hardware and claimed that this redesign improved the safety of the machine by five orders of magnitude. After modifications were made in the installed machines, AECL notified sites that they did not need to manually check the hardware settings anymore.

In November of 1985, AECL heard of another incident in Georgia. The patient in that incident (Katy Yarbrough) filed suit that month based on an overdose that occurred in June. There is no evidence that AECL followed up this case with the Georgia hospital. Though this information was clearly received by AECL, there is no evidence that this information was communicated internally to engineers or others who responded to later accidents. This lack of internal communication is likely the cause of later statements that there was no history of overdosing with the machine.

In January of 1986, AECL heard from a hospital in Yakima, Washington that a patient had been overdosed. The AECL technical support supervisor spoke with the Yakima hospital staff on the phone, and contacted them by letter indicating that he did not think the damage they reported was caused by the Therac-25 machine. He also notified them that there have "apparently been no other instances of similar damage to this or other patients."

In March of 1986, AECL was notified that the Therac-25 unit in Tyler, Texas had overdosed a patient. They sent both a local Texas engineer and an engineer from their Canada home office to investigate the incident the day after it occurred. They spent a day running tests on the machine but could not reproduce the specific error. The AECL engineer suggested that perhaps an electrical problem had caused the accident. This was in part based on the patient's report that the accident produced a "frying" sound from the machine and significant heat. The engineer also said that AECL knew of no accidents involving radiation overexposure with the Therac-25. An independent engineering firm checked out the electric shock theory and found that the machine did not seem capable of delivering an electric shock to a patient.

On April 11th of 1986, AECL was alerted to another overdose that had occurred in Tyler.

After communication with the medical physicist at Tyler, who had discovered how the replicate the malfunction, AECL engineers were able to reproduce the overdose and the sequences leading up to it. The memo “How to produce a malfunction 54” from the medical physicist is available on the computingcases.org website.

AECL filed a medical device report with the FDA on April 15, 1986 to notify them of the circumstances that produced the two Tyler accidents. At this point, the FDA, having been notified of the first Tyler accident by the hospital, declared Therac-25 defective and ordered the firm to contact all sites that used the machine, investigate the problem, and submit a report called a corrective action plan. AECL contacted all sites and recommended a temporary fix involving removing some keys from the keyboard at the computer console. In teaching the case, this is the point at which we “freeze action” in the case and ask students to play the role of a software developer at AECL and plan a response. It is significant that a later death from a different race conditions occurs in the historical case.

The FDA was not satisfied with the notification that AECL gave sites, and in May 1986 required AECL to re-notify all sites with more specific information about the defect in the product and the hazards associated with it. AECL was also at this time involved in meetings with a "user's group" of Therac-25 sites to help formulate its corrective action plan. After several exchanges of information among AECL and the FDA (in July, September, October, November, and December of 1986), AECL submitted a revised corrective action plan to the FDA.

In January 1987, AECL was notified of an additional overdose occurring at the Yakima, Washington hospital. After sending an engineer to investigate this incident, AECL concluded that there was a different software problem that allowed the electron beam to be turned on without the device that spread it to a safe concentration being placed in the beam.

In February, 1987, the FDA and its Canadian counterpart cooperated to require all units of Therac-25 to be shut down until effective and permanent modifications were made. After another 6 months of negotiation with the FDA, AECL received approval for its final corrective

action plan. This plan included numerous software fixes, the installation of independent, mechanical safety interlocks, and a variety of other safety related changes.

Several of the surviving victims or the deceased victim's families filed suit in US courts against AECL and the medical facilities using Therac-25. All of these suits were settled out of court. More detailed information about the specific accidents and their outcomes is available on the website.

Description of the Therac-25 Socio-technical system

The approach we use in describing cases is taken from Mumford (ref). The central insight is that understanding the ethical issues associated with a technology requires understanding the technology and it use within a social system. Thus, the socio-technical system is the mixture of people, technology, rules and procedures, data and data structures, and laws and regulations that structure the use of the technology and its effects on people. We provide a detailed look at the socio-technical system on the website, but give a more simplified version here.

The machine and software. There were two previous versions of Therac machines, each produced by AECL in collaboration with a French company, CGR. Therac 6 and Therac 20 (each named for the power of the beam they could produce) were based on earlier designs from CGR. By the time Therac-25 was released for sale, AECL had 13 years of experience with production of medical linear accelerators. Therac-25 was based on these previous versions. Its main innovations were (1) a "double pass" electron beam so the machine could produce more energy in less space, and (2) the addition of extensive computer control of the machine. This latter innovation allowed AECL to move much of the checking for hazardous conditions into the software.

The Therac-25's ancestors, Therac-20 and Therac-6, had used a minicomputer (a DEC PDP-11) to add some convenience to the standard hardware of a medical linear accelerator. They both could work without computer control. AECL determined to make its new model,

Therac-25, a tightly-coupled combination of software and hardware. Therac-25 software was not written from scratch, but was built up from components that were borrowed from the earlier versions of Therac.

Therac-25 was a dual mode machine. This means that it could treat the patient with relatively low energy electron beams or with X-ray beams. This dual mode allowed for further cost savings in that two machines could be replaced by one. Therac-25 also had a "field light" position that allowed a standard light beam to shine in the path of treatment to help the operator in setting up the machine. Thus there were three modes in which the Therac-25 could operate: electron beam and X-ray for treatment, and field light for setup.

Even though they are relatively low energy, the electron beams are too powerful in their raw form to treat the patient. They need to be spread thinly enough to be the right level of energy. To do this, Therac-25 placed what are called scanning magnets in the way of the beam. The spread of the beam (and thus its power) could be controlled by the magnetic fields generated by these magnets. Thus for electron beam therapy, the scanning magnets needed to be placed in the path of the beam. It was a race condition produced by a software error in setting the magnets and resulting in a turntable mismatch that produced at least two of the accidents.

X-ray treatment requires a very high intensity electron beam (25 MeV) to strike a metal foil. The foil then emits X-rays (photons). This X-ray beam is then "flattened" by a device below the foil, and the X-ray beam of an appropriate intensity is then directed to the patient. Thus, X-ray therapy requires the foil and the flattener to be placed in the path of the electron beam.

The final mode of operation for Therac-25 is not a treatment mode at all. It is merely a light that illuminates the field on the surface of the patient’s body that will be treated with one of the treatment beams. This "field light" required placing a mirror in place to guide the light in a path approximating the treatment beam’s path. This allowed accurate setup of the machine before treatment. Thus, for field light setup, the mirror needed to be placed in the path where

one of the treatment beams would eventually go.

In order to get each of these three assemblies (scanning magnets or X-ray target or field light mirror) in the right place at the right time, the Therac-25 designer placed them on a turntable. As the name suggests, this is a rotating assembly that has the items for each mode placed on it. The turntable is rotated to the correct position before the beam is started up. This is a crucial piece of the Therac-25 machine, since incorrect matching of the turntable and the mode of operation (e.g. scanning magnets in place but electron beam turned on high for X-ray) could produce potentially fatal levels of radiation. The original Leveson and Turner (ref) article includes diagrams of the machine and the turntable, and does the website.

Setup and Actuation. The Therac-25 operator sets up the patient on the table using the field light to target the beam. In doing this, treatment parameters must be entered into the machine directly in the treatment room. He or she then leaves the room and uses the computer console to confirm the treatment parameters (electron or X-ray mode, intensity, duration, etc.). The parameters initially entered in the treatment room appear on the console and the operator simply presses return to confirm each one.

The computer then makes the appropriate adjustments in the machine (moving the turntable, setting the scanning magnets, setting beam intensity etc.). This takes several seconds to do. If the operator notices an error in the input parameters, he or she can, during the setup, edit the parameters at the console without having to start all over again from inside the treatment room. It was a race condition produced by editing of parameters and resulting in a mismatch of energy with turntable position that produced at least two of the accidents.

When the computer indicates that the setup has been done correctly, the operator presses the actuation switch. The computer turns the beam on and the treatment begins. The main tasks for which the software is responsible include:

•Monitoring input and editing changes from an operator

•Updating the operator's screen to show current status of machine

•Printing in response to an operator commands

•monitoring the machine status

•rotating the turntable to correct placement

•strength and shape of beam

•operation of bending and scanning magnets

•setting the machine up for the specified treatment

•turning the beam on

•turning the beam off (after treatment, on operator command, or if a malfunction is detected)

The Therac-25 software is designed as a real-time system and implemented in machine language (a low level and difficult to read language). The software segregated the tasks above into critical tasks (e.g. setup and operation of the beam) and non-critical tasks (e.g. monitoring the keyboard). A scheduler handled the allocation of computer time to all the processes except those handled on an interrupt basis (e.g. the computer clock and handling of computer-hardware-generated errors).

The difficulty with this kind of software is the handling of things that might be occurring simultaneously. For example, the computer might be setting the magnets for a particular treatment already entered (which can take 8 seconds) while the operator has changed some of the parameters on the console screen. If this change is not detected appropriately, it may only affect the portion of the software that handles beam intensity, while the portion of the software that checks turntable position is left thinking that the old treatment parameters are still in effect. These sorts of scheduling problems when more than one process is running concurrently are called race conditions and are the primary problem that produced the accidents.

In 1983, just after AECL made the Therac-25 commercially available, AECL performed a safety analysis of the machine using Fault Tree Analysis. This involves calculating the probabilities of the occurrence of varying hazards (e.g. an overdose) by specifying which causes of the hazard must jointly occur in order to produce the hazard.

In order for this analysis to work as a Safety Analysis, one must first specify the hazards (not always easy), and then be able to specify the all possible causal sequences in the system that could produce them. It is certainly a useful exercise, since it allows easy identification of single-point-of-failure items and the identification of items whose failure can produce the hazard in multiple ways. Concentrating on items like these is a good way to begin reducing the probabilities of a hazard occurring.

In addition, if one knows the specific probabilities of all the contributing events, one can produce a reasonable estimate of the probability of the hazard occurring. This quantitative use of Fault Tree Analysis is fraught with difficulties and temptations, as AECL's approach shows.

In order to be useful, a Fault Tree Analysis needs to specify all the likely events that could contribute to producing a hazard. Unfortunately, AECL's analysis left out consideration of the software in the system almost entirely. Since much of the software had been taken from the Therac-6 and Therac-20 systems, and since these software systems had been running many years without detectable errors, the analysts assumed there were no design problems in the software. The analysts considered software failures like "computer selects wrong mode" but assigned them probabilities like 4×10^{-9}.

These sorts of probabilities are likely assigned based on the remote possibility of random errors produced by things like electromagnetic noise, or perhaps the mean-time-between-failures data generally available then for PDP-11 machines. They do not at all take into account the possibility of design flaws in the software. This shows a major difficulty with Fault Tree Analysis as it was practiced by AECL. If the only items considered are "failure" items (e.g. wear, fatigue, etc.) a Fault Tree Analysis really only gives one a reliability for the system.

Hospitals. The complexity of cancer treatment organizations is one of the things students must deal with as they struggle to understand why the accidents happened and to construct a plan to respond to the accidents.

Cancer treatment facilities are often housed in large hospitals, but some are stand-alone

cancer treatment centers (like the Tyler, Texas center). Those associated with hospitals are more likely to be non-profit, while those that stand alone are more likely to be for-profit organizations. Financial pressures are likely to be strong at both for-profit and not-for-profit organizations, but they will have slightly different regulatory structures.

During the time of Therac-25 (the mid 80s) a well equipped treatment facility might have 3 different machines. The machines would be capable of producing different kinds of radiation, different strengths of beam, and capable of different kinds of exposure to the patient. Each of these machines would cost, for the machine alone, between 1 and 2 million dollars. In addition, special housing for each machine is needed, with shielding in the walls, adequate power supply, video and intercom links, etc.

Operators would be needed to run each machine. For larger facilities, a supervisor of the operators, with more training and experience might be needed. In addition, at least one MD specialist in cancer radiation therapy (a Radiation Oncologist) would be required. Finally, a medical physicist would be needed to maintain and check the machines regularly. Some facilities contract out the services of a medical physicist. Finally, all the support personnel for these specialists (nurses, secretaries, administrative staff, people to handle billing and paperwork, janitorial staff, etc.) are required.

Medical Linear Accelerators do age over time, and older machines often produce more errors. Five to ten years is a reasonable life span for a machine. Thus, simply to maintain a set of three medical linear accelerators, an institution can expect to spend 1 to 2 million dollars every third year.

Sometimes errors can be resolved and a machine kept longer using software upgrades or upgrades or retrofits of machine parts. The companies that sell linear accelerators charge maintenance contracts that can include different levels of support. Because of monetary constraints, sometimes facilities are forced to choose between software updates, manuals, and training for operators and physicists. All this is in addition to the price of the machine itself.

Production pressures are always present when an expensive medical technology is being used. These very expensive machines need to treat enough patients to pay for themselves over their lifetime. And in for-profit medical facilities the additional pressure of generating a profit is added to this production pressure. Another kind of production pressure is generated because of concern for the patient. Patients' schedules require treatments on certain days and it disrupts the patients' lives and slows down their treatment to have to reschedule them for another day while the machine is being checked out.

These production pressures generate the desire to "push patients through." If a machine gives only a portion of the prescribed dose, an operator will often repeat the treatment with enough radiation to add up to the total prescribed dose. Of course, because of liability issues and concerns for patient welfare, this can only be done when it is thought safe.

One of the advantages of the significant computerization of the Therac 25 machine was that setup for treatment could be done much more quickly. This allowed the operator more time to speak with the patient and interact with them about their health concerns. In addition, this increased efficiency allowed more patients to be scheduled during a day. Thus, more patients could be treated, but the atmosphere was not reduced to that of a factory.

Facilities that run medical linear accelerator are surely concerned about liability for injury to patients that might occur. Insurance, for medical providers, is quite expensive and errors in treatment can result in lawsuits, which in turn produce increases in insurance premiums. Standard practice in litigation is to "sue everyone with deep pockets." This means that even if an error is the result of poor design of a linear accelerator, the facility itself will be sued simply because they were involved: they have insurance and thus "deep pockets."

But it is in the interest of facilities to reduce errors without the threat of lawsuits. When a treatment must be restarted several times because of errors, it may reduce patient confidence in the facility. This can mean patients moving to another facility with which they are more comfortable. Finally, medical professionals are in their business because they want to help

people and have the knowledge and skill to do so. So a primary motivation of medical professionals is patient welfare.

FDA. In addition to dealing with the technical issues and organizational issues associated with the hospitals, students need to consider the role the FDA plays in regulating medical devices. Understanding the constraints the FDA imposes on possible solutions (and the opportunities they provide) is a crucial part of designing a responsible solution to the accidents.

The Food and Drug Administration (FDA) was created when Congress passed the Food and Drugs Act in 1906. This act was the first of a series of laws and amendments that gave the FDA jurisdiction over the regulation of foods and patent medicines. In 1938, Congress strengthened and expanded the FDA, to include the regulation of therapeutic and medical devices within its jurisdiction.

The FDA's Bureau of Medical Devices and Diagnostic Products was created in 1974, and soon operated in conjunction with the Medical Devices Amendments of 1976. The amendments helped to clarify the logistics of the regulation of medical devices, and required the FDA to "ensure their safety and effectiveness."

Radiation had been recognized as a health hazard since before World War I, and the FDA monitored the health risks that radiation emitting products posed to America's workers and consumers. As FDA's responsibilities for monitoring radiological devices grew, a bureau within the FDA called the Center for Devices and Radiological Health (CDRH) was established.

In 1980 the FDA's budget had swelled to over $320 million, with a staff of over 7,000. Many bureaus controlled areas such as biological drugs, consumer products, public health standards, and veterinary medicines.

FDA approved medical devices before they "went to market." This was called Pre-Market Approval and was a somewhat complex process. In the FDA Pre-market Approval scheme, devices were organized into three classes, as established by the 1976 Medical Device Amendments.

•Class I devices, "general controls provide reasonable assurance of safety and effectiveness," for example bedpans and tongue depressors

•Class II devices, such as syringes and hearing aids, "require performance standards in addition to general controls"

•Class III devices like heart valves and pacemakers are required to undergo pre-market approval as well as complying with general controls

In addition to classifying devices as Class I, II, or III, FDA approved devices for market in one of two ways:

•Proof of Pre-market Equivalence to another device on the market, termed 501(k)

•OR Pre-market Approval (Rigorous Testing)

If a company could show Pre-market Equivalence (proof that a new product was equivalent to one already on the market), the new product could be approved by FDA without extensive, costly, rigorous testing. In 1984 about 94% of medical devices came to market through Pre-market Equivalence.

If a product was not equivalent to one that was already on the market, FDA required that the product go through testing to gain Pre-market Approval. In 1984 only about 6% of medical devices were required to go through this testing.

Thus, it was clearly in the interest of medical device producers to show that their product had pre-market equivalence. The Therac-25, brought to market in 1983, was classified as a Class II medical device. Since AECL designed the Therac-25 software based on software used in the earlier Therac-20 and Therac-6 models, Therac-25 was approved by FDA under Pre-market Equivalency. This declaration of pre-market equivalence seems optimistic in that (1) most of the safety mechanisms were moved into the software, a major change from previous version of the machine, and (2) the confidence in the safety of much of the software was based on its performance in the older machines, which had hardware safety devices installed to block potential accidents.

A 1983 General Accounting Office (GAO) report criticized the FDA's "adverse experience warning system" as inadequate. FDA had published reports about potential hazards, including reports in their own newsletter, *The FDA Consumer*. The FDA implemented the mandatory medical-device reporting rule after Congress passed the Medical Device Reporting Legislation in 1984. This rule required manufacturers to report injuries and problems that could cause injuries or death.

Before 1986, users of medical devices (hospitals, doctors, independent facilities) were not required to report problems with medical devices. Instead, under the medical device reporting rule, manufacturers of these devices were required to report problems. The idea was that manufacturers would be the first to hear about any problems with the devices they made and that therefore reports would be timely. In addition, manufacturers would be most likely to have the correct information needed about a device to help resolve difficulties.

In the mid-1980s, the FDA's main enforcement tools for medical devices already on the market were publicity. The FDA could not force a recall, it could only recommend one. The CDRH (Center for Devices and Radiological Health monitors radiological devices) issues its public warnings and advisories in the *Radiological Health Bulletin*. Before issuing a public warning or advisory, the FDA could negotiate with manufacturers in private (and in the case of Therac 25, with regulatory agencies in Canada). In response to reports of problems with a medical device, the FDA could, in increasing order of severity:

1. Ask for information from a manufacturer.
2. Require a report from the manufacturer.
3. Declare a product defective and require a corrective action plan (CAP).
4. Publicly recommend that routine use of the system on patients be discontinued.
5. Publicly recommend a recall.

Thus, even when the FDA became aware of the problem, they did not have the power to recall Therac-25, only to recommend a recall. After the Therac-25 deaths occurred, the FDA

issued an article in the *Radiological Health Bulletin* (Dec. 1986) explaining the mechanical failures of Therac-25 and explaining that "FDA had now declared the Therac-25 defective, and must approve the company's corrective action program."

After another Therac-25 overdose occurred in Washington state, the FDA took stronger action by "recommending that routine use of the system on patients be discontinued until a corrective plan had been approved and implemented"[12] (. AECL was expected to notify Therac-25 users of the problem, and of FDA's recommendations.

After the Therac-25 deaths, the FDA made a number of adjustments to its policies in an attempt to address the breakdowns in communication and product approval. In 1990, health-care facilities were required by law to report incidents to both the manufacturer and FDA.

AECL and the state of the technical art. A crucial part of the student's job in understanding the difficulty of the case is understanding problems with synchronization of concurrent processes (explained in detail in the teaching section later). But they also need to understand what the programmers of the Therac-6, Therac-20, and Therac-25 were likely to have known about these issues. Almost nothing is known about the specific qualifications of the Therac-25 programmers (or even their identities), but we can get some idea of the current state-of the art at the time.

Although software solutions to synchronization problems were known in the mid-1970s when AECL and CGR developed the Therac-6 software, it seems unlikely that those programmers would have used them in their implementation. An elaborate and complicated solution by Dekker[13] was available, but was "a tremendous mystification,"[14] difficult to comprehend or to program correctly. Strategies that depend on adding special features to the

[12] Radiological Health Bulletin, March 1987
[13] Cited on p. 58 of Dijkstra, E.W., "Co-operating Sequential Processes," in Genuys, F., ed., *Programming Languages,* Academic Press, 1965
[14] Ibid., p.66

operating system appeared beginning with Dijkstra[15], but such operations did not appear as a standard part of common operating systems until years later, and their implementation in the Therac-6 system seems unlikely unless they had a specialist on their team. Operating systems courses at the time focused on theoretical discussions rather than practical implementation issues. John Lions' famous commentary on UNIX Version 6 source code (Lions, 1996), developed in 1975-76 for his students in Australia, is widely considered as the first operating systems course text to address practical concerns seriously (Tanenbaum, 1987).

Thus, assembly programmers in the mid-1970s would undoubtedly have employed hardware solutions for synchronization, if they were even aware of subtle issues such as race conditions. A “test and set lock” (TSL) instruction provided one approach, in which a particular machine instruction had the capability to copy a shared variable's value to another location held privately by a task, and to assign a value to that shared variable, all in a single indivisible operation. Other machine instructions (e.g., SWAP) could serve the same purpose. However, an examination of the PDP-11 instruction set[16] shows that no such instruction exists on the machine used for the Therac systems. It is conceivable that the “subtract one and branch” (SOB) instruction, designed for loop control, might have been turned to this purpose by a creative and careful programmer who had awareness of these synchronization issue.

These facts hardly exonerate the Therac programmers. The operating-system level routines for Therac-25 were written specifically for that system according to Leveson and Turner; those programmers had a responsibility to know about issues such as race conditions in multitasking real-time systems. They would most likely have heard about postponements of the release of the IBM 360 time-sharing system and of Multics. Both of these projects were late in part because of the difficulty of getting synchronization done properly. However, unless those responsible for the operating-system executive routines had prior experience writing concurrent

[15] Ibid.
[16] Available at http://www.village.org/pdp11/faq.pages/PDPinst.html

software, it seems quite conceivable that they had never have seen the subtleties of race conditions.

Teaching with the Therac-25 case

We use the Therac-25 case as the primary case in the required *Ethical Issues in Software Design* course in the computer science major at St. Olaf College. Students also use 8 – 10 other cases during the course, but they learn Therac-25 first, and learn our approach to cases this way. Two CS courses (Intro CS and *Software Design*) are prerequisites for the ethics course, so students know something about computer science and software development methods. In these prerequisite courses, students are introduced to our emphasis on socio-technical systems and do a preliminary ethical analysis of a piece of software, including stakeholder analysis and the identification of ethical issues. Thus students come to the *Ethical Issues* course prepared to discuss a complex case like Therac-25 in considerable detail and with some technical sophistication. In addition to discussing cases, the main project in the *Ethical Issues* course is a semester-long social and ethical analysis of an existing or proposed socio-technical system. This year, for instance, students in teams analyzed a proposed new data system for the school's registrar. The project requires interviewing people involved, analyzing documentation, collecting relevant data (by observation, interface mock-ups, questionnaires, etc.) and using a structured approach to describing the socio-technical system and the ethical issues it raises. As seniors, students know they will be required to use these skills to analyze their senior projects. Thus, students know that the skills they are learning in discussing Therac-25 will help them complete their class project and later coursework in the major. One student has recently taken a job upon graduation to do projects for a firm that are substantially like those in this course. The important point here is that discussion of the historical Therac-25 case is embedded in a series of requirements and opportunities that make the skills the student learns from the case quite relevant.

Prior to beginning the case, students read about socio-technical analysis and ethical

analysis and learn methods for doing both. This takes several weeks and uses simplified historical cases. Students then read the case material on the website before the first class period and are given questions that they will be expected to answer in that period. Presentation of the socio-technical context of the case and the liability issues takes this first class period. A second class is taken up analyzing the pseudo code and presenting the technical issues (see the sections below). Students are then asked in a final class period to role play an in-house AECL software developer at the time AECL submitted its first corrective action plan (CAP). The assignment is to decide how to proceed from that point.

A crucial place in the decision stream

The problem that students are posed can be succinctly stated: does implementing the CAP solve the problem? Because we know the outcome of the case, we know the answer is no. But since they are in the stream of decision, in the middle of the case, they need to argue only from the information they have available and their current expertise. They need to propose a solution that works for AECL and its customers and that can be "sold" inside AECL as a feasible approach.

By this time, students certainly feel they have voluminous evidence in front of them, but still they long for more specifics. So the first step is to outline what additional information it would help to have. Answers to some of these questions are available from the website (e.g. the timeline of the accidents; were there consultants available at the time who specialized in safety analysis?). Other answers can be estimated from what we know (e.g. ratio of number-successfully-treated to accidents). Others are simply unknowable at the time (e.g. are there any other software errors?).

The next step is, in the tradition of ethical analysis, to list the relevant stakeholders. These are the people who need to be satisfied with any proposed solution. Then we list decision options (e.g. do nothing and declare the CAP the solution, recall all Therac-25s immediately, suspend treatment for a specified time period to study the software for errors, etc.

etc.) and we look for the effects of each of these decisions on the various stakeholders. Then we look for variations or combinations of the decision options that we think we can sell inside AECL and that best serve the interests of the stakeholders. After agreeing on a solution, we then make a plan for how we would sell the solution and what our options would be if we cannot get our solution implemented in a satisfactory manner.

This year, for instance, our students decided that since eliminating one software error did not assure them there were no errors left, the best plan was to hire outside safety consultants to review the software and the system. This would require, they felt, notifying the treatment centers that such a review was under way and giving them the option of suspending treatment during that time. The length and expense of the review would be negotiable with the company, but students felt they would need to protest vigorously if no review was done or if hospitals were not notified (there was some disagreement on this). Early warning reports from the review would help hospitals decide whether to suspend treatment or reinstate it as the review progressed. But they were insistent that the review needed to be limited in time (from 3 weeks to several months) to limit costs (to AECL, the treatment centers, and the patients) of not treating patients during that time.

Going through these steps together give students considerable practice in both recognizing ethical issues associated with a computing system and in coming up with creative solutions to resolve those issues. Students are also introduced to the idea of ethical dissent (ref) if their solutions are not adopted. Later cases in the class look at ethical dissent in more detail, but this case becomes a touchstone for when it might be required.

A beginning technical lesson from the case

One of the software errors was documented as the cause of a death at the Yakima, WA treatment center. It is a simple enough error for students with little CS background to understand, though it raises profound ethical issues about the reuse of software. This error involved incrementing a shared variable called Class3 instead of doing a Boolean. Class3

indicated whether a particular piece of equipment that shaped and limited the beam (the collimator) was or was not in the right position. If the collimator's position was correct, Class3 was set to zero, otherwise it was set to some non-zero integer by the simple expedient of incrementing it. Class3, however, was stored in a single byte, and so every 256 times it was incremented, it would overflow and register zero. If the operator pressed the set key at precisely that moment, the machine would schedule a treatment without the shaping device in the correct place, causing a race condition which resulted in an overdose. This was a rare enough occurrence that it was difficult to detect.

The contemporary milieu in which Therac programmers worked informs our understanding of the Yakima incrementation error. Incrementing a variable to change its state from zero (FALSE) to non-zero (TRUE) was a common and standard practice in the day. Assembly language programmers particularly made use of such tricks to save a few precious CPU cycles in their code. The problem with incrementation to change a boolean variable arises when that variable overflows. In the Yakima problem, such overflows, together with bad timing (race conditions), had fatal consequences. Using a two-byte PDP-11 word instead of a single byte for that variable would have made these overflows 256 times less likely (one in 65,536 instead of one in 256); a four-byte longword might never have overflowed (one in over four billion). But longwords were not available in the "16-bit" PDP-11 architecture, unless one built a multi-precision integer variable oneself (at a cost to performance); and the first inclination of any assembly programmer would be to try to get away with a single byte, applying the same minimal-usage philosophy to memory space as to CPU execution time.

Furthermore, it is unclear whether the requirements for the early 70s (and more simple) Therac-6 software influence the decision to use incrementation instead of a boolean.. Was incrementation of an 8-bit variable to change its (boolean) state part of the reused Therac-6 software, perhaps intended for initial calibration only, but later used throughout the control of the more complicated dual mode Therac-25? We cannot tell without knowing more specifics about

the Therac code and its evolutionary development. In any case, this hypothesis indicates the kind of imagination about possible effects when a bit of software is reused that we must expect from programmers. Such imagination is difficult enough to instill in present-day programmers, and was in very short supply among 1970s assembly language programmers.

The world has learned much about how to think about software reliability over the last three decades. It is now inconceivable that the FDA would approve a software-dependent device such as the Therac-25 with as little testing of the software functionality as that system received---in large part due to the experience gained from this very system and others that looked acceptable at the outset but later generated unanticipated problems. The word that little logical flaws such as incrementing to change boolean state could result in catastrophic failure was slow to trickle back to programmers.

Students who are introduced to this “technical “ issue in the Therac-25 case are made aware of how important documentation is when software may be reused. They become sensitized to the ethical issues that can arise from shifting requirements as software is developed. They begin to understand how one piece of software in one socio-technical context can operate flawlessly and can, in another context, produce significant harm. And they begin to become aware of what “best practices” mean, how they can shift over time, and why they are important. All of these points make it clear how closely coupled technical and ethical issues are.

An advanced technical lesson from the case

The other documented software error was the cause of two deaths in the Tyler, TX treatment facility. It involved concurrent access to a shared variable, that set up a race condition resulting in a mismatch of beam energy with the turntable position. In this case, it involved monitoring and verifying treatment data input from the operator. The variable indicated whether data entry was completed or not. Under certain conditions, a race condition would occur with this variable and if the operator set initial parameters and later edited them the later

edits would show on the screen but would not be detected by the portion of the software that implements the machine settings. This could result in the wrong piece of the turntable being in place with the high energy beam turned on, producing a massive overdose.

In technical terms, these race conditions would have been resolved by indivisible set and test operations. Explaining what this means and why it is important is a topic that comes up on operating systems courses today, but that was not widely known and only poorly understood in the early 70s when some of the Therac-6 software was produced. But students today can see that technical issues from their courses can have profound ethical implications.

A closer look at the staged development of the Therac-6, Therac-20 and Therac-25 software given “best” computing practices in the 1970s raises uncertainties about the overall responsibility for race condition failures in that system. In order to understand these uncertainties, students first need a review of (or introduction to) the technical issues.

A race condition in a software system arises when the correct operation of that system depends on timing of execution. Race conditions can only occur in systems that involve multiple tasks (or processes) that carry out their instructions concurrently, or at the “same time,” perhaps through a timesharing operating system. To illustrate the subtle nature of race conditions in a multitasking system, consider Dijkstra's classic “dining philosophers” problem (1965): Imagine several individuals sitting around a dinner table, each of whom alternates between thinking and eating for indeterminate amounts of time. In order to eat, one of these “philosophers” requires two utensils. In Dijkstra's analogy, two forks were required to eat spaghetti; others have suggested chopsticks and Chinese food. Unfortunately, each philosopher must share his/her utensils with his/her neighbors. Imagine a circular table with 8 plates around the edge and 8 single chopsticks, one to the right of each plate. To eat, each philosopher must pick up one chopstick from each side of his or her plate. This means that the philosophers on either side of the dining philosopher will have to wait to eat (in the analogy, they spend the time thinking), since they each have access to only one chopstick. In this

hypothetical problem, the philosophers represent multiple concurrent tasks, and the chopsticks represent computing resources (variables, input/output devices, etc.) that those processes must share.

Correctly instructing (programming) each philosopher-task about how to proceed is a non-trivial problem, as the Appendix shows. For example, in an attempted solution algorithm for which each philosopher receives a boolean-valued shared variable that is TRUE when that philosopher may safely pick up both utensils and eat, inopportune timing may lead one philosopher A to yield eating rights to a second philosopher B between the time when B checks for eating rights and the time when B begins to think, awaiting a “wakeup” from A; if A sent B's “wakeup” signal before B has begun to sleep, then B might enter a state of permanent inactivity while unknowingly holding the sole right to eat. Thus, the correct behavior of the system of philosopher algorithms depends on timing---a race condition. Naïve efforts to repair this approach by saving “wakeups,” sharing different variables, etc., simply move the bad-timing condition to another location in the multitasking algorithm. Furthermore, in practice, race conditions are among the most difficult system bugs to find, since one cannot duplicate the precise timing conditions that cause the transient error, except by chance, unless one already knows what that transient error is. In the Therac-25 case, the Tyler race condition did not show up unless particularly well practiced operators made editing changes very quickly.

The only effective way to combat race conditions is to avoid them. In the example above, one can avoid the lost “wakeup” by insuring that philosopher B cannot be interrupted between the step of checking its shared variable and the step of entering its thinking phase, i.e., by causing the system to perform both steps in a single indivisible operation. In this way, the philosopher is assured that the unfortunate timing error of having the wakeup call occur right after checking for it but before entering the thinking stage (when it would cause some action) is made impossible.

Now that we know what a race condition is, we can see in the Therac-25 how it can kill

people. Therac-25 appeared in 1983, and its programmers modified the software from the earlier Therac-6 system, programmed in the early-1970s. Given the era when software for these systems appeared (see the "state of the technical art" section earlier), it seems unlikely that machine language programmers in 1972 would have sufficient knowledge to know how to avoid race conditions with concurrent processes. This makes the technical error more understandable. But it brings up the larger issue, again, of documentation of code when it is reused and of the effect of shifting requirements and socio-technical systems. It also makes clear the importance of including a carefully designed maintenance phase in the software life-cycle; Part of the reason the errors were not caught is that there was no clear mechanism for them to filter back to the appropriate people at AECL (or the FDA). Again, what look like technical issues become ethical ones. Students who learn about race conditions from the Therac-25 case know about the importance of a professional's responsibility to be aware of best practices and the state of the art in one's area of expertise, and the dangers of operating outside of their area of expertise.

Leveson & Turner say that "focusing on particular software bugs is not the way to make a safe system …The basic mistakes here involved poor software-engineering practices and building a machine that relies on the software for safe operation."[17] This is not the lesson that is often drawn from the case, but presenting the case in this fashion makes it clear technical decisions made by the programmers in the context of their historical environment had profoundly ethical implications for the eventual users of the system. In this way putting the case in historical perspective makes it clear that ethical decision making is inseparable from good software design methodology, and that *good* in this context deserves its double entendre.

Relevance of Therac-25 to other CS courses and concepts

Instructors can readily make connections between suitably detailed historical cases and many courses in the CS curriculum, because these cases have enough complexity to touch on

[17] Leveson and Turner, 38

substantial topics from those courses. For example, the Therac-25 discussion in this paper includes significant content from at least the following undergraduate CS courses (in addition to courses specifically in Computing Ethics).

Computer Organization: instruction set architecture; assembly language programming.

Operating Systems: synchronization (interprocess communication), including race conditions, hardware and software solutions; multitasking operating system design.

Software engineering: reuse; requirements analysis; software evolution and maintenance; ethical and professional context.

Elective courses: History of Computing (historical context of computing at the time of Therac software development); Real-time Systems (example of small dedicated system, importance of correct synchronization); Systems Programming.

In each of these examples, use of an historical case also connects course content to the complexity of problems in the applied world, in terms of computing technology, ethical issues, and professional concerns.

These cases highlight pervasive themes that appear throughout the discipline of computer science. For example, taking the "recurring concepts" of the joint ACM/IEEE recommended computing curriculum.[18] The Therac-25 example relates to at least the following.

Trade-offs and consequences: efficiency vs. correct synchronization; productivity vs. quality; computing vs. non-computing solutions.

Reuse: advantages of reuse; dangers of changing requirements; testing of systems with reused code; importance of documentation including specification of reusable code.

Evolution: example of evolutionary system development in industry; risks of poorly managed evolutionary development; evolution of both code and requirements; resiliency of a system in the face of change.

Consistency and completeness: formal methods; correctness; importance of proper

[18] Available at: http://www.computer.org/education/cc1991/

system specification, and adequacy of specifications; reliability practices for real-time systems; behavior of a system under error conditions and unanticipated situations.

Efficiency: historical changes in attitudes towards software efficiency; importance of efficiency in context; efficiency of production in an applied project.

Security: defense of a system against unanticipated requests; response of a socio-technical system to catastrophes; misuse of data types; user interface features; physical security measures; role of software in a security strategy; testing of system security.

Conclusion

To reiterate a claim made earlier, these connections of computer science principles to the historical case of the Therac-25 make it clear that good software development deserves the double entendre in the word good: that the ethical practice of computing requires deep technical knowledge and skill and commitment to good practice. This becomes most clear when we see historical cases of ethical issues in computing that have enough complexity to include technical detail. Dining philosophers help students understand what race conditions are; Therac-25 helps students understand why they matter.

Appendix: Basics of Concurrency and Race Conditions

Multitasking

- A *process* (or ***task***) is an execution of a program. Note that in a multiuser system, we would expect multiple processes executing various programs; it may be that two or more processes even execute the same program "simultaneously" through time sharing.
- ***Multitasking*** means using multiple tasks in a single system.
  Example: Dining philosopher's problem (see below), which involves multiple tasks sharing various resources.
- The use of multitasking leads to new and complicated kinds of computer bugs.
  Example: ***deadlock*** -- existence of a set of processes, each of which is ***blocked*** (unable to run) waiting for an event that can only be caused by another process in that set. See first attempted solution of dining philosopher's problem below.
- Multiple processes may require ***shared variables*** in order to carry out their work.
  Example: Two processes may use a shared "buffer" data structure for holding items that have been produced by one process but not yet consumed by another.

Example: Dining Philosophers Problem (E. Dijkstra, 1965)

- *N* processes share *N* resources in an alternating circular arrangement. Each process has two states (computing, interacting with resources); each needs exclusive access to its two resources while interacting with them.
- Dijkstra's statement: *N* philosophers sit down together to eat spaghetti. Each philosopher spends his/her time alternately thinking then eating. In order to eat, a philosopher needs two forks; each fork is shared with exactly one other of the philosophers. What procedure will allow all the philosophers to continue to think and eat?
- Algorithm 1. Each philosopher does the following:

```
repeat forever
   think
   pick up left fork (as soon as it's available)
   pick up right fork (as soon as it's available)
   eat
   return left fork
   return right fork
```

Issue: *Deadlock* occurs if each philosopher picks up his/her left fork simultaneously, because no philosopher then can obtain a right fork.

- Algorithm 2. Each philosopher does the following:

```
repeat forever
   think
   repeat
      pick up left fork (as soon as it's available)
      if right fork is available
         pick up right fork
      else
         return left fork
   until both forks are possessed
```

```
   eat
   return left fork
   return right fork
```

Issue: Although deadlock has been prevented, s*tarvation* occurs if each philosopher picks up his/her left fork simultaneously, observes that the right fork is unavailable, then simultaneously returns the left fork and tries again, ad infinitum.

Race conditions

- A ***race condition*** exists in a system if the correct operation of that system's algorithm depends upon timing.

  Example: Third attempted solution of dining philosopher's problem below. This algorithm uses a shared array `myturn[]` in an effort to prevent problems like deadlock. But "bad luck" in terms of timing could lead to a deadlock after all.

- Example of an algorithm with race conditions: Algorithm 3 for Dining Philosophers. The philosophers use a shared variable myturn, an array of length *N*; if myturn[*p*] is *true* then it is (hopefully) safe for philosopher *p* to eat. (We use an array rather than a single variable location to allow for *concurrency*---multiple processes executing at the same time.)

  We assume that there are functions sleep() for causing a philosopher to doze off (blocked process) and wakeup(*p*) for waking up a philosopher *p*.

  Each philosopher does the following:

```
if (p == 1)
   myturn[p] = true
else
   myturn[p] = false
next = (p+1) % N
prev = (p+N−1) % N
repeat forever
   think
   if (!myturn[p])
      sleep()    /* to be awakened by prev philosopher */
   pick up left fork
   pick up right fork
   eat
   return left fork
   return right fork
   myturn[p] = false
   myturn[next] = true
   wakeup(next)
```

  Issues: Less than maximal concurrency; race conditions

- The following illustration shows how bad timing might occur with the algorithm above.

```
philosopher p=1              philosopher p=2
                    +>
                    |    ...
                    |    // myturn[p] IS FALSE (p=2)
                    |    think
                    |    if (!myturn[p])
                   <+
```

```
   ...                          |
   myturn[p] = false //p=1      |
   myturn[next] = true          |
   wakeup(next)  // WAKEUP p=2 |
repeat forever                  |
   think                        |
   ...                          |
   if (!myturn[p])              |
      sleep()   // BLOCK!       |
                               +>
                               |          sleep() // BLOCK!
```

•The next illustration shows a second race condition during initialization.

```
philosopher p=1                philosopher p=2
                              <+
if (p == 1)                    |
   myturn[p] = true            |
else                           |
   myturn[p] = false           |
next = (p+1) % N               |
prev = (p+N−1) % N             |
repeat forever                 |
   think                       |
   if (!myturn[p]) // FALSE    |
      sleep()                  |
   pick up left fork           |
   ...                         |
   myturn[p] = false           |
   myturn[next] = true         |
                              +>
                              |   if (p == 1)
                              |     myturn[p] = true
                              |   else
                              |     myturn[p] = false
                              |   // myturn[p] IS NOW FALSE
                              |   next = (p+1) % N
                              |   prev = (p+N−1) % N
                              <+
   wakeup(next)  // WAKEUP p=2 |
repeat forever                 |
   think                       |
   ...                         |
   if (!myturn[p])             |
      sleep()   // BLOCK!      |
                              +>
                              |   repeat forever
                              |      think
                              |      if (!myturn[p])
                              |         sleep() // BLOCK!
```

•Race conditions are *extremely* difficult to debug , since one must look for bad combinations of instruction execution order for multiple independent programs. The worst part about debugging race conditions is that the bugs are *intermittent* --- not usually repeatable on a subsequent run, as hard as you may try to replicate the running conditions.

•The best approach is to *avoid race conditions in the first place*: A race condition can only arise when there can be a "gap" between *retrieving* a shared variable's value and *using* that retrieved value. See dining philosopher race condition examples above

Comment: persons who haven't had a course like Operating Systems are likely to be totally unaware of this race condition issue!

Strategies for correct IPC

- Race conditions are problems in *interprocess communication (IPC)* or ***synchronization***, i.e., mechanisms and practices for creating correct systems having multiple processes.
- To prevent race conditions, one needs some kind of *atomic* or ***indivisible*** operations that leave no possibility for timing "gaps."
- An Operating Systems course examines several (equivalent) higher-level software strategies for correct IPC, including *semaphores*, *monitors* (cf. Java's `synchronized` objects), and *message passing*.

  Comment: Therac25 assembly programmers would not have any of these higher level solutions available unless they built them themselves --- an unlikely scenario for time-pressured programmers with only low-level programming resources who might not even have awareness of the issue, especially given the care that correct programming of one of these strategies would require.
- There are also hardware solutions, such as having a *test and set lock (TSL)* instruction in the ISA (machine language) of the machine being used. In a TSL instruction, a memory value can be retrieved ("tested") and a new "lock" value substituted in a single indivisible machine instruction, preventing a "gap" between "testing" and "locking." Thoughtful use of a TSL instruction can correctly solve IPC problems.
- Leveson and Turner's analysis: "*It is clear from the AECL documentation on the modifications that the software allows concurrent access to shared memory, that there is no real synchronization aside from data stored in shared variables, and that the "test" and "set" for such variables are not indivisible operations. Race conditions resulting from this implementation of multitasking played an important part in the accidents.*"
- *Comment: The Therac25 system used a standard, highly regarded computer produced by DEC (Digital Equipment Corporation) called the PDP11. The PDP-11 instruction set includes no TSL instruction. (There is also no SWAP instruction to interchange values of two independent memory locations, which could serve in place of a TSL instruction.) Thus, Therac25 programmers would have had to devise something else for correct synchronization.*