Chapter 1: Why software engineering?

1. This type of situation could be indicative of a crisis in that the inadequacies of software contributed to the tragic loss of life. The fact that the pilot relied completely on the software to guide the plane turned a user interface error into a severe accident. In general, of course, aviation is better off because of software engineering because the high reliability of aviation software has made flying safer and more cost-effective. The situation that caused the accident is really a user interface problem. During development, more user feedback could have been designed into the software (so that the pilot might have been notified that he had typed in the code for Bogota, not Cali) and more usability testing could have been done (so that the misunderstanding might have surfaced before the software was installed in the airplane). The other issue is that of system boundary. Although the aeronautical charts used by pilots are outside the boundary of the computer system, the developers need to have knowledge of them and the codes and abbreviations they contain.

2. One example is the processing and synthesis of sensor data. If lots of real-time sensor data are flowing in to the software system, the hard part is getting the timing right so that an accurate picture of the overall system is obtained. For instance, if the software is controlling the shape of the wing of a plane, based on real-time sensor data about the air speed, pressure, temperature, etc., decisions must be based on up-to-date information from the sensors, even when flying through rapidly changing conditions (like a storm). The actual polling of the sensors is easy; the relationships are hard. The relationships are more complicated still when dealing with asynchronous changes to the wing shape.

3. Errors are misunderstandings that reside in the developer's thought processes. When that misunderstanding leads the developer to write something in a development artifact (specification, design, code, test data, etc.) that is not correct, that incorrect information is called a fault. When the fault causes the software to behave in an incorrect manner or produce incorrect results, that behavior is called a failure. An example of an error is when a developer mistakenly believes that the diameter of a circle is three times its radius. This error could lead to a fault in the requirements document if this developer writes a requirement that says that the system, when given a radius as input and a command to compute the diameter, should return three times the radius. Or, the error could lead to a fault in the design if the design for the module for converting radii into diameters includes the faulty formula. Or, if the formula relating radius and diameter is not specified in either the requirements or the design, a fault could still be introduced in the code if the developer encodes his/her erroneous understanding of the formula. If the fault is introduced in the requirements or the design and propagates to the code, then it may become a failure if that part of the code is exercised and consequently produces a wrong output. However, if the test data contained a fault that caused the code for computing the diameter not to be executed during test, then this would allow a failure to occur during operational use, when that code is exercised.

4. A count of faults is a misleading measure of product quality because there usually is not a one-to-one correspondence between faults and failures. If many faults are located in code that is never or rarely executed, then they are unlikely to result in many failures, which is usually a more relevant measure of quality. On the other hand, if just one fault is located in code that is exercised heavily in regular use, it could result in numerous failures, and thus low-quality software. Furthermore, neither a count of faults nor a count of failures gives an indication of the severity of the problems.

5. In one small division of NASA, a group of software developers who developed software to support satellite missions decided to build an object-oriented library of software components. It was estimated that such a library would drastically reduce development time in this division, and would even allow the physicists and flight dynamics analysts in the division to put together their own applications rather than relying on software developers to write the programs they needed for mission support. The library was built using an architecture based on many years of experience in this domain, using sound software engineering principles, and was tested extensively. The components in the library were of very high technical quality. However, the library was of limited value to its intended customers, the physicists and analysts, because there were no tools or documentation that facilitated its use. Users of the library had to have a deep understanding not only of the application domain, but also of object orientation and software development, as well as the structure of the library itself. Thus, the library was not the success it was hoped to be.

The Therac 25 case is a good example of the dangers of narrowing the definition of quality. The developers of the software no doubt had tested the software extensively and considered it to be high quality. However, what was relevant was the quality (in particular, the safety) of the system as a whole: software, hardware, and operator. Because the Therac 25 developers adopted such a narrow view of quality, lives were lost.

6. The advantages of using COTS software packages include lower development costs, and no responsibility for maintenance. The disadvantages include unknown reliability, risk of vendor dropping support or refusing to make needed changes or demanding unreasonable prices for future support, and the vendor's claims being misleading.

   Developers, customers, and users must anticipate the possibility of having to abandon the COTS product and having to replace it with locally written code or another COTS product. In other words, there must always be a contingency plan.

7. Ideally, the originators of the software that failed should be responsible for the consequences. However, it is often not possible to determine precisely what part of a system caused a failure, and it is often a combination of factors, or misunderstandings between the developers of the system itself, subcontractors, and/or COTS developers. One way to resolve the issue is to specify carefully such legal responsibilities ahead of time in the initial contracts. For example, the lead company might be willing to take on all liability for failure in exchange for additional cost savings from the COTS vendor or subcontractor. In this case, of course, the company will want extensive evidence of the quality of the software it is buying. Standard measures of quality (if they existed) would be helpful. At the very least, particular tests can be specified ahead of time that the software must pass before being accepted.

8. For example, the rule that advertisements for alcohol may be shown only after 9pm should be kept within the system boundary since the content and time of the advertisement are necessary pieces of information for the system to keep track of anyway and it would not be too difficult for it to check for conformance to this rule. On the other hand, it may become too complicated for the system to keep track of the content of advertisements in general. In that case, it could not include this constraint within its boundary.

   A similar argument could be made for the rule that if an advertisement for a class of product (such as an automobile) is scheduled for a particular commercial break, then no other advertisement for something in that class may be shown during that break. If advertisement content information can easily be tracked at the proper level of detail to enforce this rule, then it makes sense to keep it within the system boundary.

   Another rule states that if an actor is in a show, then an advertisement with that actor may not be broadcast within 45 minutes of the show. In order to include this constraint within the boundaries of the system, the system would have to keep track of all actors appearing in all programs and all advertisements. This information is considerable and not otherwise useful for the system to handle, so it would make sense for this constraint to be left outside the system boundary.

9. One reasonable way to assess the impact of a failure is how severely it affected people and how many people it affected. It is part of the job of the media to assess the impact in terms of the amount of damage (i.e. money or injury) and how much of society is affected (i.e. all taxpayers or the investors in a particular business). The Ariane-5 incident involved a large amount of money "owned" by many people (taxpayers). The failure of the Panix system, presumably, only affected its users and investors. Therefore, by this measure, the Ariane-5 incident had more impact and thus deserved wider coverage. However, of course, there are deeper issues. Perhaps a particular incident that had little impact itself may be indicative of problems in the same or another system that could cause much bigger problems later. It may be considered the job of the media to bring the public's attention to such potential problems. For example, the Panix failure could provide an opportunity to educate the public about the potential dangers of Internet "invasions."