CSE503: Software Engineering

David Notkin
University of Washington
Department of Computer Science & Engineering
Spring 2006

1

Design for Safety and Reliability

- What is safety and what is reliability?
- Roughly,
  - reliability is the probability that a system will work as intended
  - safety addresses whether the system behaves in a way that protects key interests (human, economic, environmental, etc.) if and when the system fails
- "A car that doesn't stop is unreliable; a car that doesn't safe is safe."

2

What design techniques help achieve these desirable properties?

- This characterization is indeed very limited
- Specifically, approaches to reliability and safety require much broader attention than to design alone

3

Leveson says…

…that safety methodologies must address: "special management structures and procedures, system hazard analysis, software hazard analysis, requirements modeling and analysis for completeness and safety, design for safety, design of human-machine interaction, verification (both testing and code analysis), operational feedback, and change analysis."

4

NASA says…

"To most project and software development managers, reliability is equated to correctness, that is, they look to testing and the number of 'bugs' found and fixed. While finding and fixing bugs discovered in testing is necessary to assure reliability, a better way is to develop a robust, high quality product through all of the stages of the software lifecycle. That is, the reliability of the delivered code is related to the quality of all of the processes and products of software development; the requirements documentation, the code, test plans, and testing."

5

That said, we'll focus on design

- Furthermore, we'll focus on non-malicious situations
- And we'll focus on design and programming issues, rather than on language design issues

6

David B. Wortman (Ed.): *Proceedings of an ACM Conference on Language Design for Reliable Software* 1977. SIGPLAN Notices 12(3) March 1977, Operating Systems Review 11(2) April 1977, Software Engineering Notes 2(2) March 1977: included…

- Ambler: GYPSY…
- Popek, Horning, Lampson, Mitchell, London: Euclid…
- Fischer, LeBlanc: … Run-Time Checking in Pascal.
- Ambler, Hoch: A Study of Protection in Programming Languages.
- Friedman, Wise: … Applicative Programming for File Systems
- Herriot: Towards the Ideal Programming Language.
- Guttag, Horowitz, Musser: Some Extensions to Algebraic Specifications…
- Buckle: Restricted Data Types …
- Cousot, Cousot: Static Determination of Dynamic Properties of …
- Melliar-Smith, Randell: … The Role of Programmed Exception Handling.
- Love: An Experimental Investigation of the Effect of Program Structure on Program Understanding.
- Andrews, McGraw: Language Features for Process Interaction.
- Lomet: Process Structuring, Synchronization, and Recovery Using Atomic Actions.

Elements of Programming Style: Kernighan and Plauger (1978)

1. Write clearly -- don't be too clever.
2. Say what you mean, simply and directly.
3. Use library functions whenever feasible.
4. Avoid too many temporary variables.
5. Write clearly -- don't sacrifice clarity for efficiency
6. Let the machine do the dirty work.
7. Replace repetitive expressions by calls to common functions.
8. Parenthesize to avoid ambiguity.
9. Choose variable names that won't be confused.
10. Avoid unnecessary branches.
11. If a logical expression is hard to understand, try transforming it.
12. Choose a data representation that makes the program simple.
13. Write first in easy-to-understand pseudo language; then translate into whatever language you have to use.
14. Modularize. Use procedures and functions.
15. Avoid gotos completely if you can keep the program readable.

16. Don't patch bad code -- rewrite it.
17. Write and test a big program in small pieces.
18. Use recursive procedures for recursively-defined data structures.
19. Test input for plausibility and validity.
20. Make sure input doesn't violate the limits of the program.
21. Terminate input by end-of-file marker, not by count.
22. Identify bad input; recover if possible.
23. Make input easy to prepare and output self-explanatory.
24. Use uniform input formats.
25. Make input easy to proofread.
26. Use self-identifying input. Allow defaults. Echo both on output.
27. Make sure all variable are initialized before use.
28. Don't stop at one bug.
29. Use debugging compilers.
30. Watch out for off-by-one errors.
31. Take care to branch the right way on equality.
32. Be careful if a loop exits to the same place from the middle and the bottom.
33. Make sure your code does "nothing" gracefully.
34. Test programs at their boundary values.
35. Check some answers by hand.
36. 10.0 times 0.1 is hardly ever 1.0.
37. 7/8 is zero while 7.0/8.0 is not zero.

38. Don't compare floating point numbers solely for equality.
39. Make it right before you make it faster.
40. Make it fail-safe before you make it faster.
41. Make it clear before you make it faster.
42. Don't sacrifice clarity for small gains in "efficiency."
43. Let your compiler do the simple optimizations.
44. Don't strain to re-use code; reorganize instead.
45. Make sure special cases are truly special.
46. Keep it simple to make it faster.
47. Don't diddle code to make it faster -- find a better algorithm.
48. Instrument your programs. Measure before making "efficiency" changes.
49. Make sure comments and code agree.
50. Don't just echo the code with comments -- make every comment count.
51. Don't comment bad code -- rewrite it.
52. Use variable names that mean something.
53. Use statement labels that mean something.
54. Format a program to help the reader understand it.
55. Document your data layouts.
56. Don't over-comment.

28 years of perspective: in 1978…

- Ruben Studdard was born
- Kurt Gödel died, as did two popes
- Herb Simon won the Nobel Prize
- First computer bulletin board system created
- Camp David peace accord
- Proceedings of U.S. broadcast on radio for the first time

28 years of perspective:
Key aspects of design for safety and reliability

- KISS: Keep It Simple, Stupid
- Know the enemy
  - Initialization
  - Speed kills
  - Arithmetic precision
  - Syntactic ratholes
  - …
- Remember that programs define behavior for a computer but also represent communication to programmers

One more:
“assume” makes an “ass” out of “u” and “me”

- ANSI C: `memcpy` between overlapping strings has undefined behavior
- Ada: `in out` parameter passing mechanism can be defined by the compiler (by-reference, copy-in copy-out) – programs with behavior that depend on the mechanism are “undefined”
- Expecting well-defined input
- … many many many many many more!

13

Lots of common sense…

- Yes, much of this is common sense
- However, it isn’t always applied in practice
- Checklists, and automatic checking tools, can help
- External, independent assessments are also essential

14

Techniques for reducing hazards
(Leveson)

- Taken from her in-depth study of failures of complex systems (including those without software components)

- Substitution
- Simplification
- Decoupling
- Elimination of potential for human error
- Reduction of hazardous conditions

15

Substitution

- Often this question, for software, addresses whether or not to use a computer-controlled system at all
  - Therac-25 replacement of hardware interlock by computing system
- Replacing library implementation of `free` with one that overwrites the freed memory with an unusual bit pattern
- Don’t encode “safe” and “armed” states as 0 and 1 – hardware failure modes are often “stuck at” 0 or 1, and it may be difficult to distinguish failure modes from software flaws
- A “dead man” switch on a train is a classic example
- … other examples?

16

Simplification

- Safety design errors are often found at (hardware and software) interfaces
- Simple interfaces tend to reduce errors and make designs more testable
- The Honeywell JA37B autopilot – “the first full-authority fly-by-wire system” – ran 15+ years without an in-flight anomaly
  - One of the designers (Boebert) said that there was “purposeful simplification” in the design
  - A “rate structure” design allowed no interrupts and no back branches – the control look was unwound into one loop executed at a fixed rate
  - They traded off data complexity for control simplicity

17

Boebert (via Leveson)

- One explanation for the overuse of complex designs in software control systems:
- “I laid full blame for this circumstance on CS faculty who either knew nothing other than operating systems or held OS designs up as the ultimate paradigm of software. So CS students and new grad software engineers came out thinking that an autopilot should look like Unix.”
- In other words, unsafe software is your fault and my fault!

18

Simplification #2

- Non-determinism is often more complex than determinism
  - Combinatorial blowups may compromise understandability, testing, reasoning, debugging, etc.
  - Software errors may appear as transient software faults – software engineers are likely to accept this explanation too easily

19

Determinism

- Most safety-critical systems are real-time, embedded, reactive-systems
- Determinism helps with
  - Time periodicity
  - Analyzing and predicting algorithm behavior
  - Need to test and reproduce test conditions
  - Need for human operators to rely on consistency

20

Reduce flexibility

- The lovely, malleable nature of software is often a tempting danger in safety-critical systems
- In general, one should only include minimal features and capabilities

21

Decoupling

- Tight coupling increases interdependence – a failure in one component can cause unexpected behavior in another component
- Separate safety-critical functions from non-critical functions
  - Security kernels that are small, can be validated, etc. – may even be possible to put the code in read-only memory to protect against modification
  - On-board computing systems are separated by level of criticality

22

Human errors

- Proper design can reduce human error
  - More a bit later but…
- Metal detector at SeaTac – make it audible when it doesn't detect anything illicit (as well as when it does)
- Also, at the programming language level – some features are particularly prone to error including: pointers, some control structures, defaults, implicit type conversions, global variables, overloading, …

23

Reduction of hazardous conditions

- Mostly in system design – for example, reduce amount of hazardous material in stock
- A software example might be to initialize memory not used by the program to a pattern that will cause the system to revert to a safe state if that memory is executed

24

A few other safety techniques

- Redundancy
  - Is generally most effective against random failures – and software usually only contains design errors
  - Data redundancy can help – parity bits, checksums, etc.
  - Control/algorithmic redundancy – N-version programming, which is at best expensive
- Recovery/rollback

25

Human Interaction Aspects

- Don Norman: “The problem, I suggest, is that the automation is at an intermediate level of intelligence, powerful enough to take over control that used to be done by people, but not powerful enough to handle all abnormalities…”

26

Sheridan’s list of options

- Human does everything
- Computer tells human the options available
- Computer tells human the options available and suggest one
- Computer suggests an action and implements it if asked
- Computer suggests action, informs human, and implements it if not stopped in time
- Computer selects action, informs human, and implements it if not stopped in time and then informs human
- Computer selects and implements action and tells human if asked
- Computer selects and implements action and tells human if designer decides human should be told
- Computer selects and implements action without any human involvement

27

Need to understand humans (and computers)

- Allowing a user to verify a set of data/instructions through a sequence of “enter” commands will (if there are few errors) train most humans to simply repeatedly hit the “enter” key
  - A minor variant is online acceptance of licenses with the default to “yes” vs. “no”
- Human alertness, human error tolerance, providing feedback

28

Some principles

- Make safe-enhancing actions easy – otherwise they are more likely to be bypassed by humans
- Allow humans to quickly place the system into a safe state with a simple action
- Require complicated procedures to initiate any potentially hazardous function
- Keep things simple, natural and similar to what humans have experienced before – reduces errors when humans are under stress or are distracted
  - Safety-critical systems should surely assume that operators are sometimes under stress
- Presentation of more is not always better – cognitive overload
- Modalities are important

29

Some concrete, low-level techniques (many from *Writing Solid Code*)

- Use all tools available – compiler options, lint, purify, …
- Use `assert` statements aggressively
  - But don’t use them to identify errors (such as, “is user-entered data in range”) but rather illegal conditions (such as “no more memory”)
- Don’t assume an invocation will succeed

30

Think about the interface, even at a low level

- `tolower` – returns the lowercase equivalent of an uppercase letter vs. returns the lowercase equivalent of an uppercase letter if one exists (and otherwise returns the original argument)

-
  ```
  #define BASE10 1
  #define BASE16 0
  void UnsignedToStr(unsigned u, char *strResult,
  flag fDecimal
  ```
  - Versus two separate functions versus base as a numeric parameter

31

More

- Don't reference memory you don't own
- Don't use output memory as workspace buffers
- Write your code for the "average" programmer – kind of like using complex legal language vs. stuff people can understand

32

Attitude: yours

- Don't fix bugs later, fix them now
- Fix the cause, not the symptom
- Don't blame testers for finding your bugs
- Don't keep trying solutions until you find one that works; invest in finding the correct solution

33

Saltzer and Schroeder: eight principles [1975]

- Least Privilege
- Fail-Safe Defaults
- Economy of Mechanism
- Complete Mediation
- Open Design
- Separation of Privilege
- Least Common Mechanism
- Psychological Acceptability

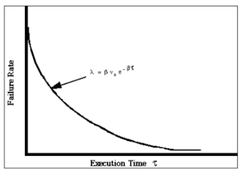
34

Some patterns…

- I also read a ton of stuff on the web – defensive programming, quality programming, secure programming, etc.
- The themes of simplicity and personal responsibility seem to dominate, even for non-safety-critical systems

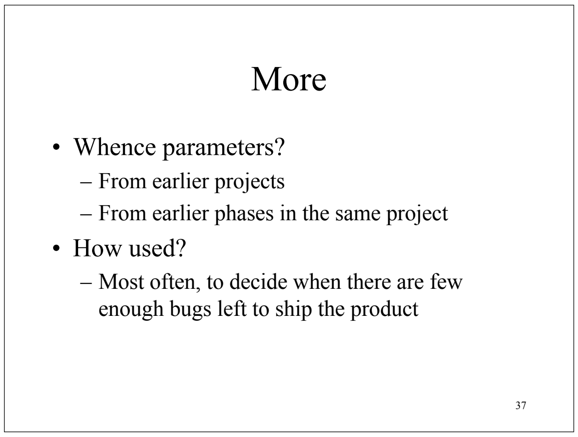
35

A tad on "classic" software reliability



- These models describe "the failure behavior of a system during operations and system test, given model parameters and the expected decrease in the failure rate per failure and the number of failures needed to remove all faults.
- Reliability can be predicted by estimating these parameters from other parameters known early in the lifecycle."
- `www.thedacs.com`

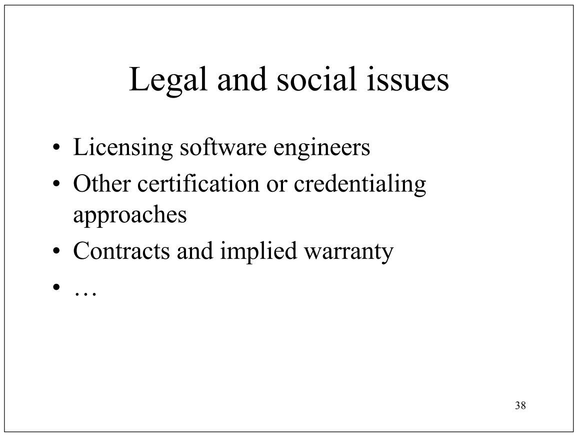
36



More

- Whence parameters?
  - From earlier projects
  - From earlier phases in the same project
- How used?
  - Most often, to decide when there are few enough bugs left to ship the product

37



Legal and social issues

- Licensing software engineers
- Other certification or credentialing approaches
- Contracts and implied warranty
- …

38