Where do we go from here and why?

"To err is human but to really foul things up requires a computer"

Farmer's Almanac for 1978
"Capsules of Wisdom"

"60% of all major industrial disasters from 1921 to 1989 occurred after 1975"

Nancy Leveson
"Safeware: Computers and Technology"

Why is software engineering difficult?

- Complete flexibility and ease of change
- Complexity and invisible interfaces
- Discrete state vs. analog systems
- Lack of historical usage information

Curse of flexibility

- Computers provide a level of power, speed and control not otherwise possible

"In one stroke we are free of nature's constraints. This freedom is software's main attraction, but unbounded freedom lies at the heart of all software difficulty"

Easy to achieve partial success (90% of time)

- Flexibility encourages redefinition of tasks late in development to accommodate other parts of the system (example: C-17)
- Flexibility encourages premature construction (prototyping). Not bad but ...

Limiting functionality

"And they looked upon the software, and saw that it was good. But they just had to add this one other feature…"

-G.F. McCormick
"When reach exceeds grasp"

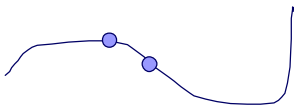
- What *can* be done vs what *should* be done
- Attempt to do too much
- Example: Denver airport automatic baggage handling system

Complexity and invisible interfaces

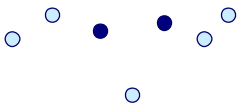
- Say we wanted a "smart car"
  - loudness of horn is proportional to speed of car
  - AC adjusts to the amount of weight present in the back seat
- Hard (requires design). In software -easy. No immediate and obvious costs

Analog vs. Discrete systems

- Analog - easier to test (can rely on continuity)



- Discrete - small change in circumstances might change the program behavior considerably



No historic usage information

- Software is always specially constructed
- What if every part of airplane or car were completely changed for each new model or version?
- Nobody kept track and evaluated how we are doing
  - mistakes
  - successful designs
  - we reinvent the wheel a lot
  - reuse helps somewhat (OO paradigm)

Additional problem with reactive systems

- Too many scenarios!
- Informal specifications often miss a few (10%?) essential scenarios.
- Implementors often do not know what the correct behavior is suppose to be:
  - Ask users ($$$)
  - Make assumptions (sometimes wrong!!!!!!!)

Example: auto-pilot

Problem:

"Design a part in auto-pilot that avoids collision with other planes."

Solution:

"When distance is 1km, give warning to other plane and notify pilot. When distance is 300m, and no changes in the course of other plane were noticed, go up to avoid collision"

Problem with solution

- Both planes have the same software. Both go up...

More such examples

- US aircraft went to southern hemisphere and … flipped when crossing the equator
- Air traffic controller: US to Britain. It never dealt with problem of 0 degrees longitude. Result: software "folded" Britain along Greenwich Meridian, plopping Manchester on top of Warwick
- Software written for US F-16 - accidents when reused in Israeli aircraft flown over the Dead Sea
  (altitude < sea level)
- Year 2000 problem

Yet more such examples

- NASA Space Shuttle software (in use since 1980)
  - 16 severity-level 1 software errors
  - 8 remained in code that was used in flights
  - none encountered during flights
  - total size - only 400,000 words

How to do quality assurance?

- "system always does the right thing"
  - Testing (what are problems with testing?)
  - Fault injections
- "system never does the bad thing"
  - Fault-tree analysis [Leveson95]

Group "essentially similar" behaviors together so that fewer cases need to be considered.

However, complex systems usually exhibit "essentially different" behaviors!

Other QA methods

- Code inspections
  - Read code with several people, attempting to find bugs
- Mathematical analysis (formal methods)
  - "prove" that software is correct
  - Ex: Darlington nuclear power plant
  - Very expensive ($7,000,000 for 2,500 lines of code)
  - Cannot be entirely automated (Halting problem)

Formal Methods

"The use of mathematical modeling, calculation and prediction in the specification, design, analysis, construction and assurance of computer systems and software."

Engineering - partial differential equations to model variations in physical quantities over time and space.

Software - model discrete quantities. Need concepts like sets, graphs, partial orders, finite-state machines.

"Calculation" is in terms of formal logic rather than numerical computation.

Where do we go from here?

"Learn realistic techniques for specifying, designing and analyzing large and complex systems"

- How to pose questions about global behaviors of computer systems
- How to answer such questions automatically
- How to refine specifications, via architecture, to code, to ensure that it is correct.
- Design patterns - reuse of common paradigms often occurring in software systems

The techniques will be examplified using an Elevator Controller System