Whys and Whynots of Software Engineering

Marsha Chechik
University of Toronto
Department of Computer Science
July 8, 1997
http://www.cs.toronto.edu/~chechik

Motivation

"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"

This talk...

- What is software engineering?
- Why do we need it?
- Where do we need it?
- A few techniques

Project Development (school projects)

- Teacher: well-defined problem

  "Write a program that takes a file, adds numbers in it and prints out result"

- Student: implement the solution, ask teacher for clarifications if needed, or make assumptions

```
open file
if cannot open file
   print "File does not exist"
else
  sum = 0
  while not end of file
     read number
     sum = sum + number
  print number
```

Project Development (school projects)

- Student: test the program on a couple of inputs

```
file doesn't exist
file contains 2, 3, 4, 5
```

- Teacher: test the program on a couple of inputs, give evaluation

```
file does not exist
file is empty
file contains 2
file contains 2, 3, 4
```

Grade: A

“Real” project development

- Customer - gives vague requirements

  “I want something that will fly, heat up my dinner, water my plants and balance my checkbook. Maybe...”

- Requirements analysis - attempts to clarify exactly what the system does, run it by customer, make sure everyone agrees what is being built
- Design - “plan” the solution

  data structures, architecture, hardware, major algorithms, interface characterization, etc.

- Code - putting this into machine-understandable language. Shouldn’t take more than 20-30% of development time

“Real” project development

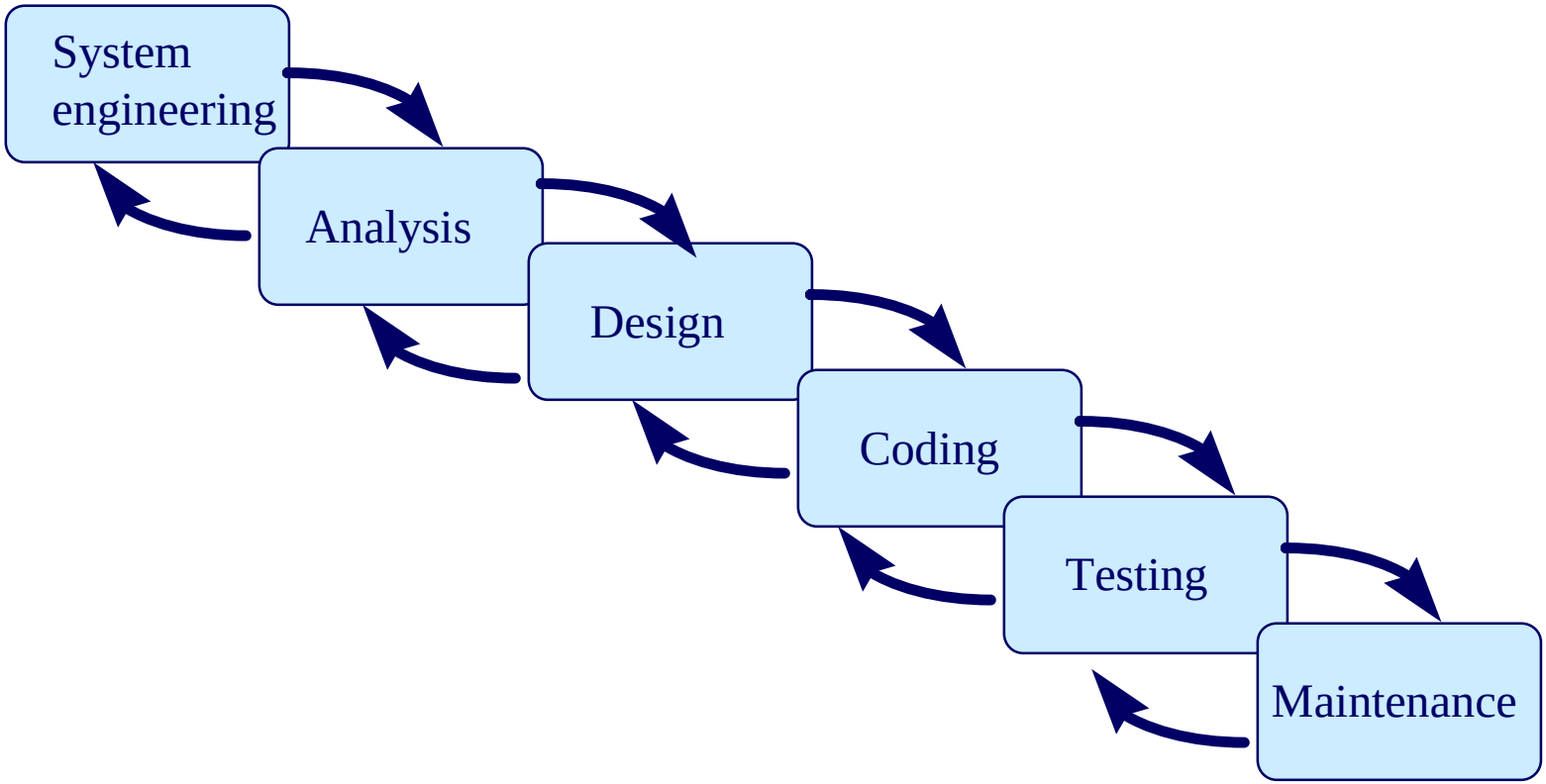
- Test code. Make sure that you built exactly what is specified in requirements
- Maintain code. Fix bugs, add functionality, etc.

Combined, this is software development life cycle.

Several ways to put these activities together:

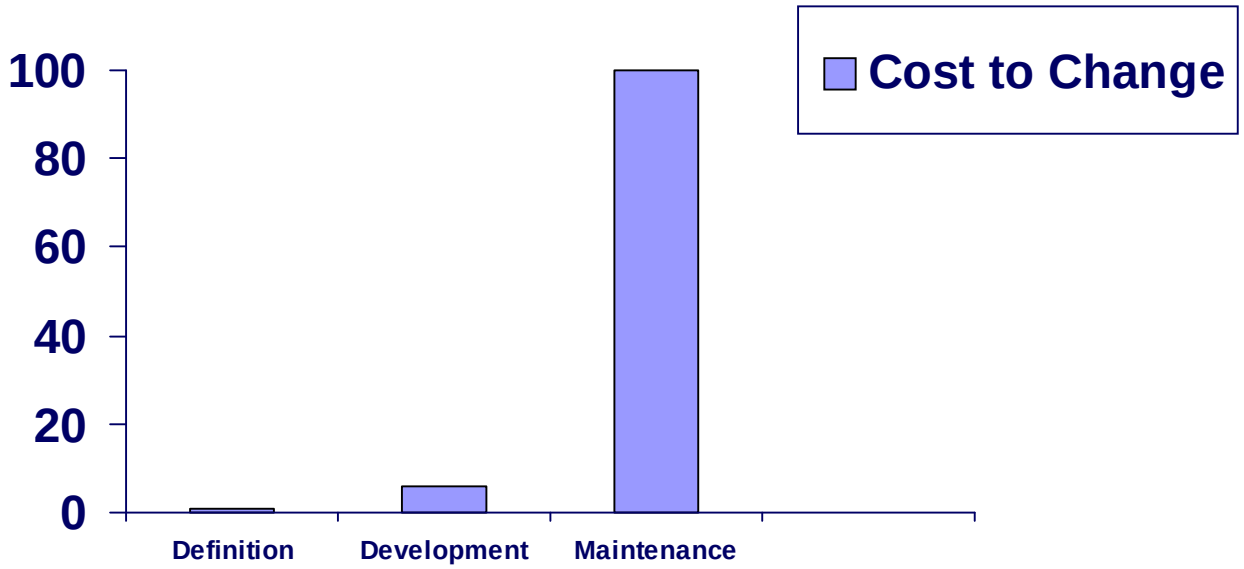
- Waterfall model
- Spiral model

Waterfall model (“old”)



Which part is most important?

- Requirements!



Problems

- Real projects almost never follow sequential flow
- Difficult for customer to state all requirements explicitly
- Customer must have patience. A working version of program might not be available until … late.
- And what if this is not at all what they wanted?

Spiral model (“new”)

- figure 1.9 of Pressman, third edition, p. 29

Safety-Critical Software

- where software failure might result in loss of human life or property
- Examples:

Want to get those right!

Easy to get right 90% of time, hard all the time since we cannot foresee all scenarios.

  - ask users
  - make assumptions (sometimes wrong)

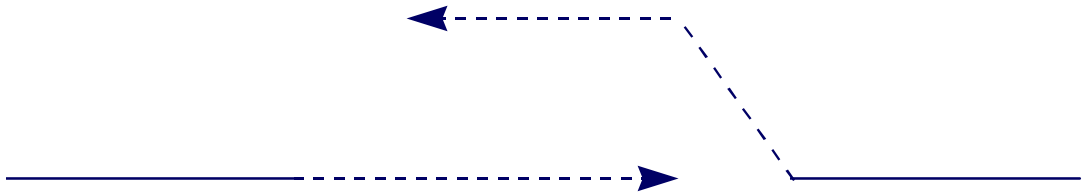
Example: auto-pilot

Problem:

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

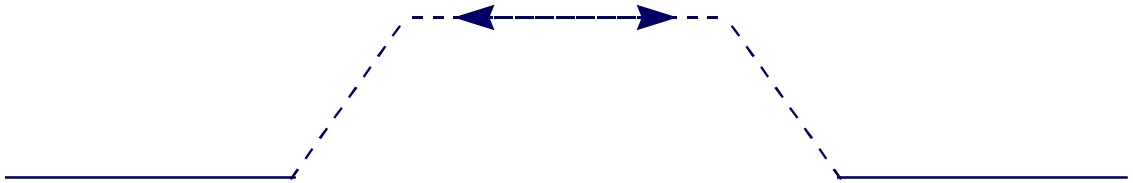
Solution:

"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 Dear 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

Good representation of specification

Figure 6.10, p. 197 of Pressman

Source: Gelileo’s Discorsi e Dimonstrazioni Matematiche ubtorno a due nuove science, Leyden, 1638

Requirements Analysis

- Attempts to check that the entire system satisfies some global properties
- Attempts to check system for consistency (cannot derive true and false)
- Attempts to check system for completeness (everything we wanted to say has been said for all scenarios)
- Attempts to “animate’ requirements

Sometimes is pure black magic

Sometimes can use mathematics rather than prose for specifications. Used in safety-critical systems.

How to tell that software is good?

- 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 fully automated (Halting problem)

Testing

- Run program with some inputs, see if outputs are correct

"If we were really good at programming, there would be no bugs to catch"

"Testing and test case design is an admission of failure"

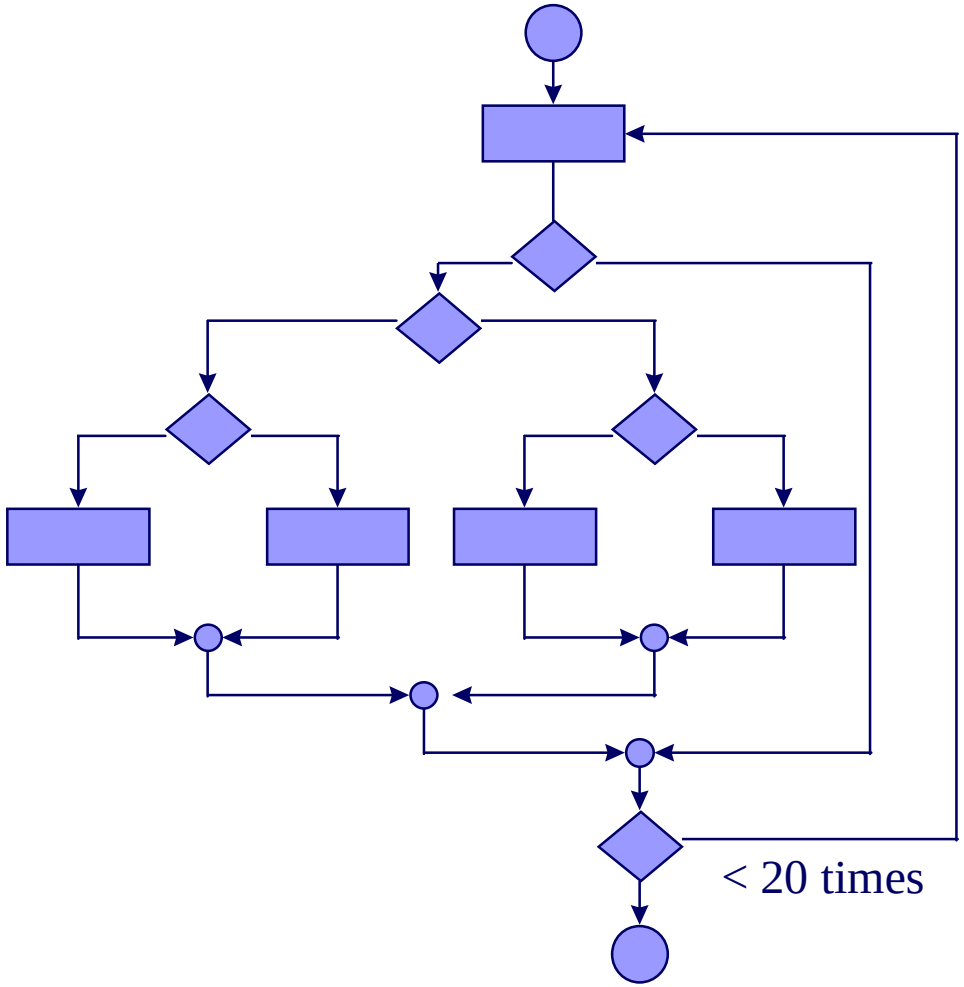
"Tedium of testing is just punishment for our errors"

- What is a successful test case?
- What are problems with testing?

Types of testing

- Black box: knowing the specified function that a product has been designed to perform, tests can be conducted to demonstrate that each function is fully operational
- White box: knowing the internal workings of a product, tests can be conducted to ensure that the internal operation of the product performs according to specification and all internal components have been adequately exercised

Problems with exhaustive testing



- 100 lines of Pascal code

- 10^{14} possible paths

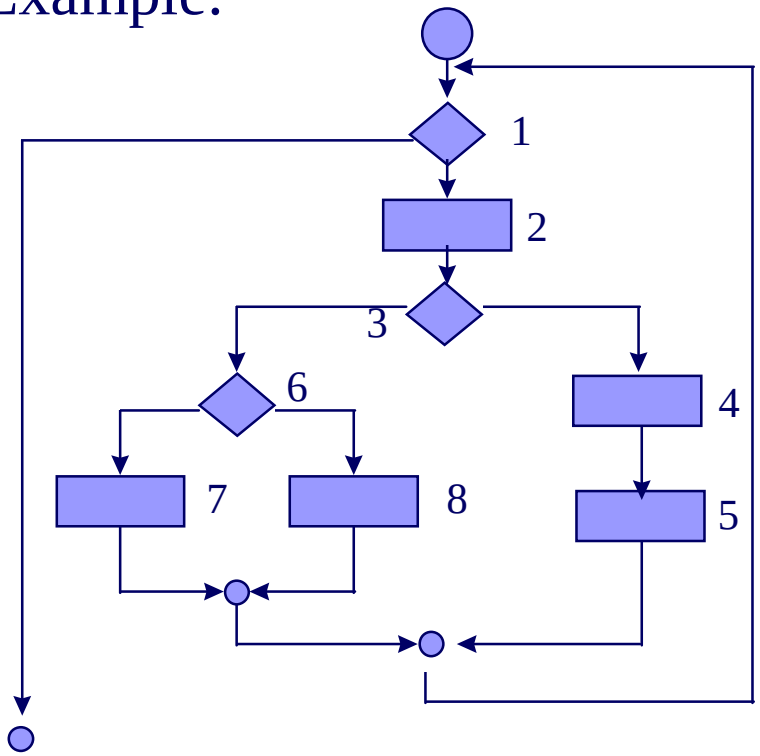
- Have a magic test processor that can develop, execute and evaluate results of testcase in 1 ms

- Will take 3170 years (working 24 hours a day, 365 days a year) to test this program!!!!!

Next best thing: Statement Coverage

- Goal: execute each statement once
- Hard: usually only 70-80% of code gets executed

Example:



Independent paths - have a new statement or new condition.

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

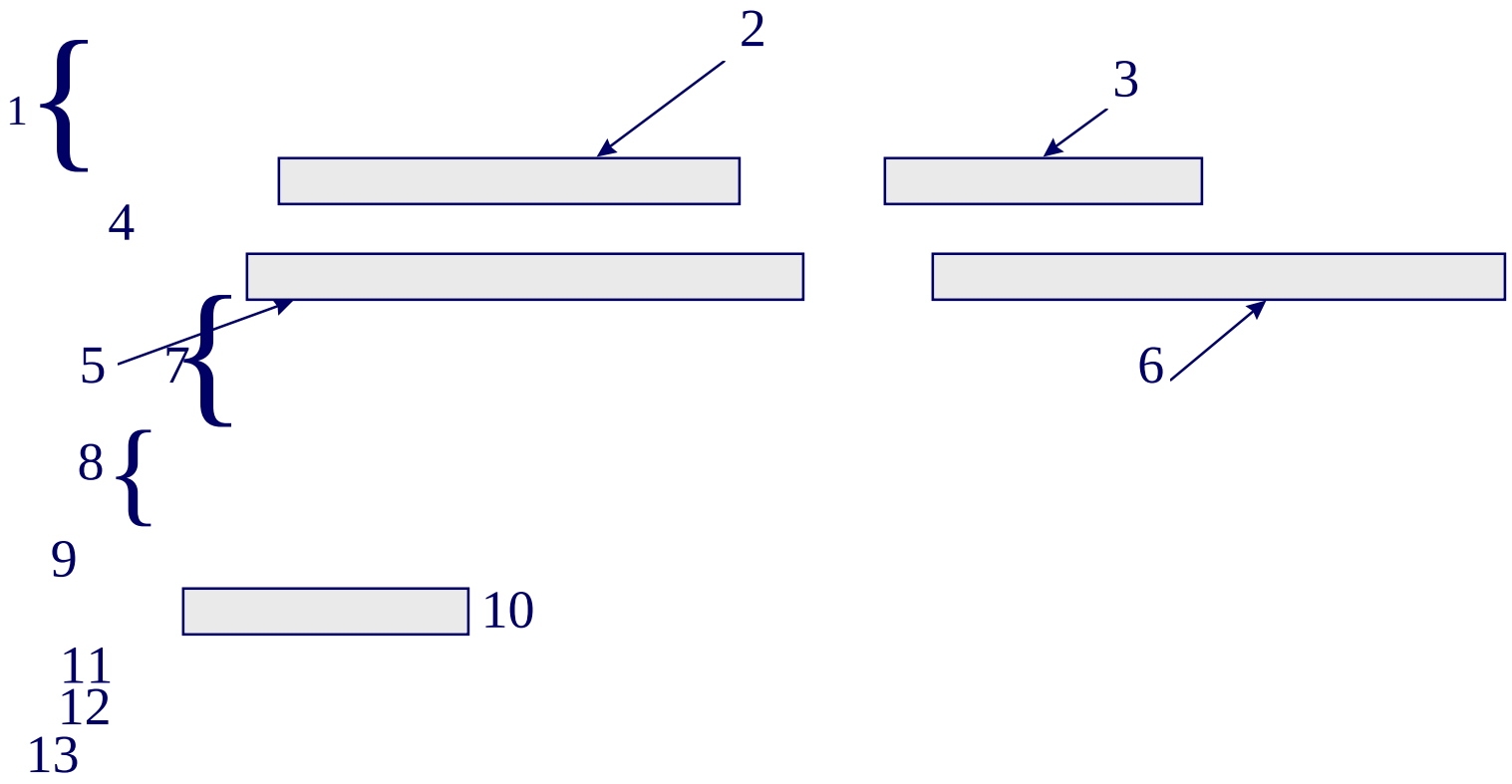
Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Then design test cases to execute these paths

Another example

```
Procedure average;
  value : array [1..100] of integers;
  average, input, valid : integer;
  minimum, maximum, sum, i : integer;
  i = 1;
  input = valid = 0;
  sum = 0;
  while value[i] <> -999 and input < 100
    input = input + 1;
    if value[i] >= minimum and value[i] <= maximum
      then valid = valid + 1;
           sum = sum + value[i];
      else skip
    endif
    i = i + 1;
  endwhile
  if valid > 0
    then average = sum / valid;
    else average = -999;
  endif
end
```


1
2
3
4
5
7
6
8
9
10
11
12
13

Example (Cont'd)

Notes:

- When two conditions are combined with AND, the second condition is evaluated only if the first is true

```
value[i] <> -999 and input < 100
```

- When two conditions are combined with OR, the second condition is evaluated only if the first is false

```
value[i] > 0 or value[i] <= maximum
```

Paths

Path 1: 1-2-3-10-11-13

Path 2: 1-2-3-4-5-8-9-2-…

Path 3: 1-2-3-4-5-6-8-9-2-…

Path 4: 1-2-3-4-5-6-7-8-9-2-...

Path 5: 1-2-10-11-13

Path 6: 1-2-10-12-13

Designing testcases

Path 1 test case:

attempt to process 101 or more values
first 100 values should be valid
expected result: correct average based on 100 values and proper totals

Path 2 test case:

value(i) = valid input where i < 100
value(k) < minimum where k < i
expected results: correct average based on k values and proper totals

Path 3 test case:

value(i) = valid input where i < 100
value(k) > maximum where k <= i
expected results: correct average based on k values and proper totals

Designing testcases (Cont'd)

Path 4 test case:

value(i) = valid input where i < 100
expected result: correct average based on 100 values and proper totals

Path 5 test case:

value(k) = valid input where k < i
value(i) = -999 where 2 <= i<= 100
expected results: correct average based on k values and proper totals
Note: cannot be tested alone; must be with path 2, 3, 4 tests

Path 6 test case:

value(1) = -999
expected results: average = -999; other totals at initial values

Why is this not enough?

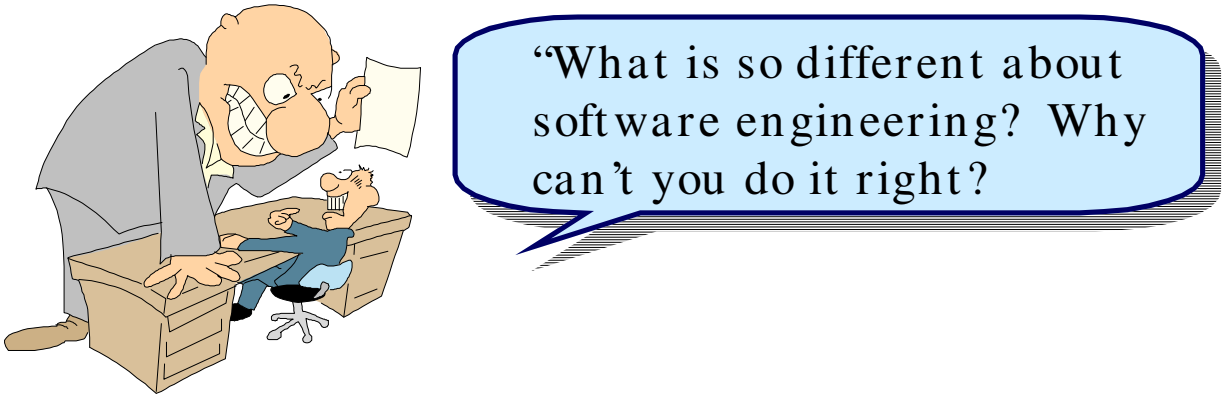
Example:

```
procedure Calculator
  x, y, z : real;
  read(x);
  read(y);
  x = x - 2;
  y = y + x;
  if x < 3
    then z = x / y;
    else z = x * y;
  endif;
end;
```

Black box testing

- Applied during later stages of testing
- In safety-critical systems:
  - Sometimes have redundant software, developed by separate engineering teams
  - System behaves based on majority vote
- Apply redundancy to software testing
  - Develop several versions of software (only one to be used)
  - Use majority votes to determine if testcase has been successful (no need to *know* output beforehand)

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 ...

Program design

picture on p. 35 of Leveson

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"

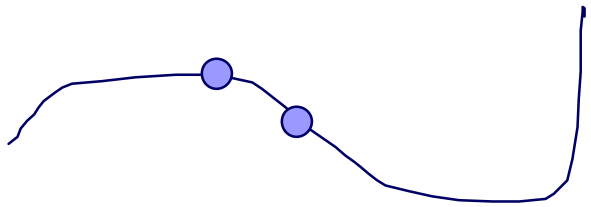
- What *can* be done vs what *should* be done
- Attempt to do too much
- Example: Denver airport automatic baggage handling system

Complexity and invisible interfaces

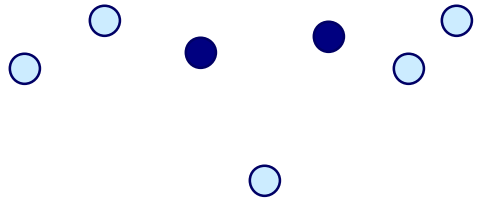
- Say we wanted a “smart car”
  - loudness of horn is proportional to speed of car
  - AC adjusts to the amount of weight present in the pack seat
- Hard (requires design). In software -easy. No immediate and obvious costs

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)

Conclusion

- Different models of software development in practice
  - Waterfall model
  - Spiral model
- Safety-critical systems
  - accidents associated with use of software
- Requirements analysis
- Testing
  - white-box
  - black-box
- Why is software engineering so hard?

A Final Thought

"Computers have a tremendous impact on our lives. And whenever a technology has a broad impact – an impact that can save lives or endanger them, build businesses or destroy them, inform government leaders or mislead them – it must be "handled with care"..."

- Roger Pressman