# Lecture #5

| Address | Memory | Binary | Instruction/Data |
|---|---|---|---|
| ⋮ | | | |
| 30B0 | 0 0 0 4 | | |
| 30B1 | 2 2 B 0 | | |
| 30B2 | E 0 B 7 | | |
| 30B3 | F 0 2 2 | | |
| 30B4 | 1 2 7 F | | |
| 30B5 | 0 2 B 3 | | |
| 30B6 | F 0 2 5 | | |
| 30B7 | 0 0 6 8 | | |
| 30B8 | 0 0 6 9 | | |
| 30B9 | 0 0 2 0 | | |
| 30BA | 0 0 0 0 | | |
| ⋮ | | | |

# Exercise

- Write out the memory image of a program loaded at location x3000, that:
  - computes the sum of the integers contained in the 12 locations x3100-x310B
  - prints this sum to the console

# Software Engineering

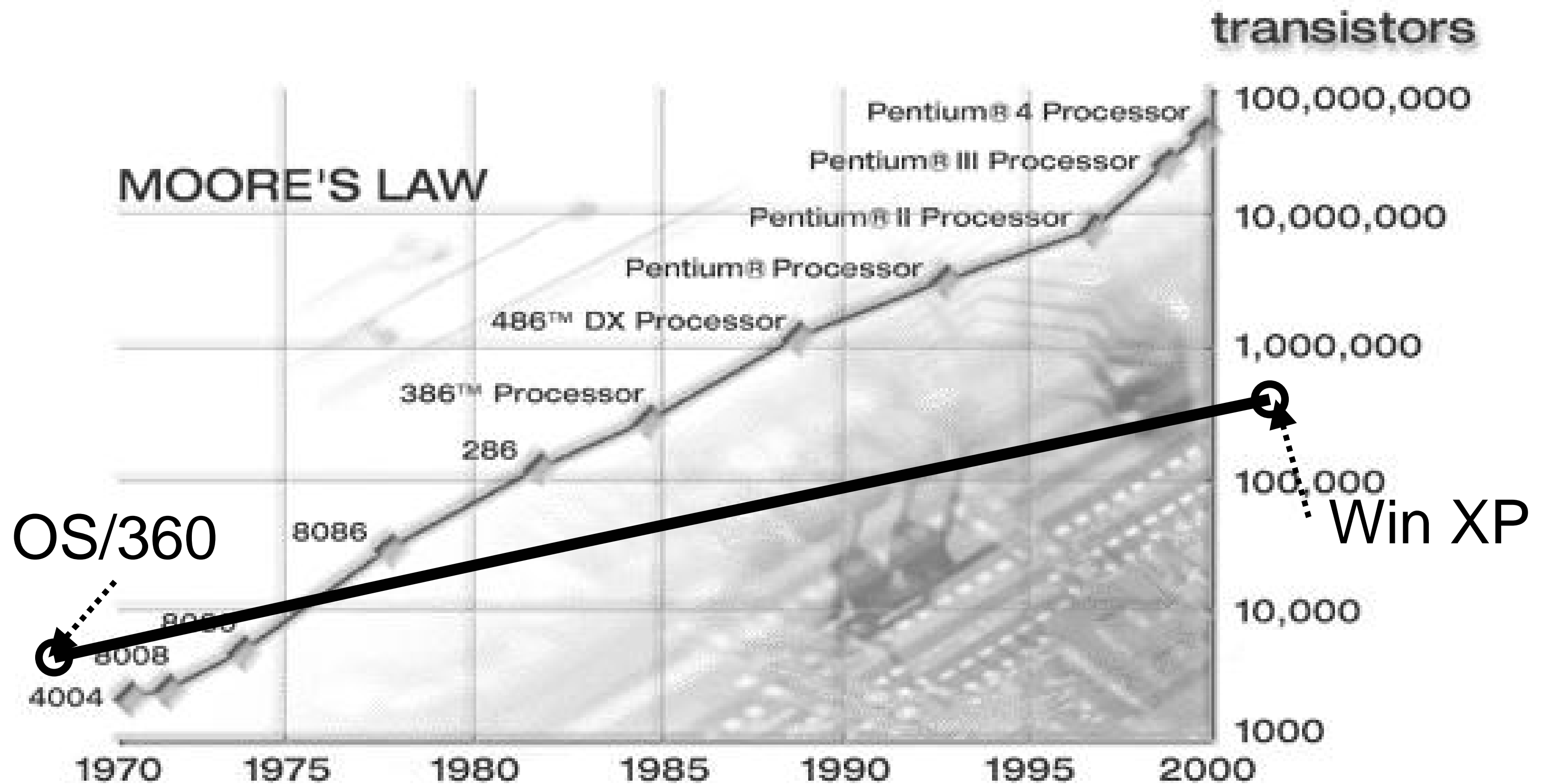
# The "Software Crisis"

- We're in the midst of a s/w *crisis*
  - and we've been there for 30+ years!
- Complexity continuously increasing:
  - machine
  - software
- Tools and techniques to manage complexity
  - tools: CASE, analysis, testing, …
  - techniques: languages, methodologies, …

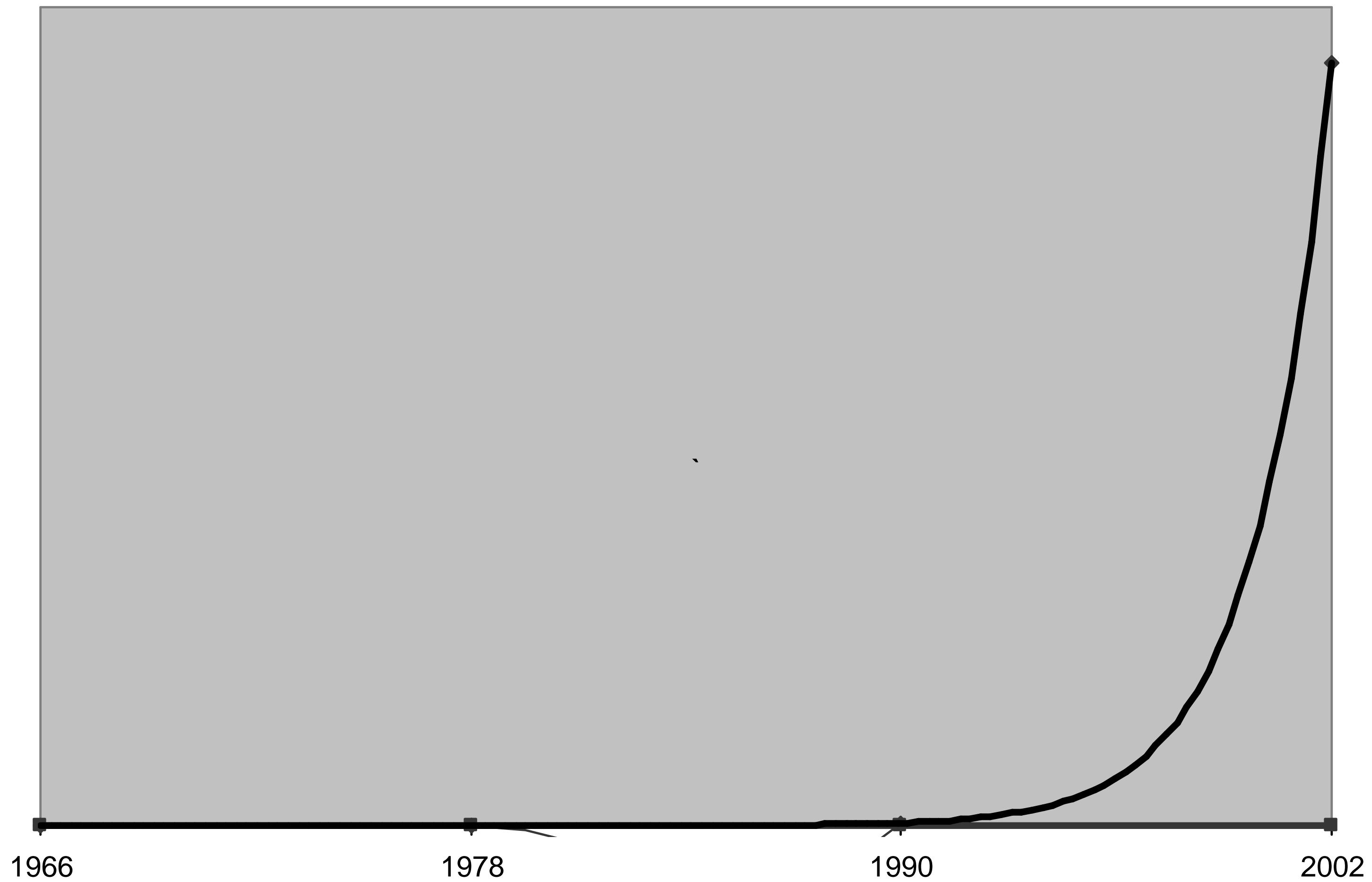
- However, system complexity frequently pushes the envelope…
- Net effect:

The support for building complex systems always seems to lag behind the systems we build (or want to build) !!

# Moore's Law



# The Great Software Crisis



# Characteristics of Well-Designed Code:

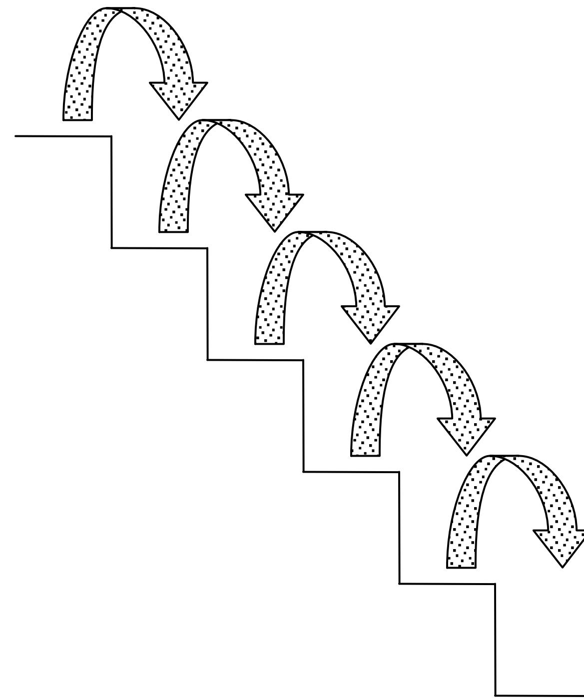
- Easy to _______
  - user-friendly
  - robust
  - efficient
  - flexible (and easy to configure)
  - easy to maintain

# Characteristics of Well-Designed Code (II):

- Easy to _______
  - easy to understand and reason about
    - documentation and simplicity of design
  - easy to read
    - coding conventions and style
  - easy to modify
  - easy to extend

# Waterfall Model of Development

- Simple model of software development
- Occurs in stages:
  1. requirements analysis
  2. system specification
  3. design
  4. implementation
  5. testing
  6. maintenance / support



# Problems with this Model

- There is no “barrier” between steps
  - e.g. begin testing *before* implementation done
- Water flows uphill
  - e.g. working on design reveals gaps in requirements analysis
  - more like an Escher print than a real waterfall!

# Fundamentals

- Many alternatives to pure waterfall exist
  - spirals, matrices, …
- Concept of distinct *stages* is useful
  - helps structure the effort, like a “battle plan”
- Some basic stages:
  1. requirements / specification
  2. design

# Lecture #6

# Basic Stages in S/w Development: Req Analysis & System Spec

- Answers: “*What* does the system do?”
- Focus: *understanding* the problem
  - usually the software engineers are not experts in the problem domain (e.g., health care, insurance, banking…)!
- Deliverables:
  1. requirements document
     - written from the user’s point of view: functionality, cost, performance
     - *defines* and *limits* the scope of the system
     - forms a contract, of sorts, between the client and the developer
  2. specification document
     - written from the developer’s point of view
     - basis for design / implementation / testing
     - document for internal consumption

# Basic Stages in S/w Development: Design

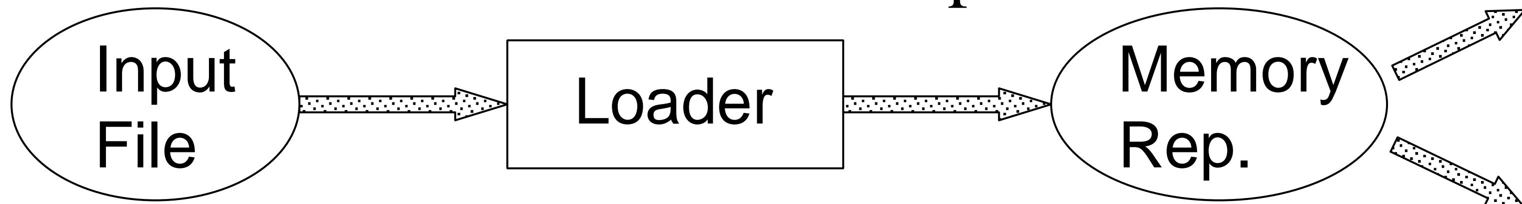
- Answers: “*How* does the system do what it does?”
- Focus: *architecting* the software artefact that will solve the client’s problem
  - usually the client is not an expert in software engineering!
- High Level Design
  - *identify* and *evaluate* possible solutions
  - possible metrics: simplicity, effort, cost, licensing, performance,…
  - refine the design
- Lower Level Design
  - functional description of components, interfaces, and interactions
    - abstraction is critical
  - given in terms of data structures, procedures, algorithms, …

# Basic Stages in S/w Development: Module Specification

- Answers: “What is the role of each element in the designed architecture?”
- Focus: *identifying* the abstractions
- Deliverables:
  1. Behavioural description of the modules
     - see the specification skeleton in the “Writing a Programmer’s Guide” handout
  2. Description of the interactions (interfaces)
     - see the data element dictionary section in the handout
     - other interactions include call graphs, inheritance trees, data flow, etc…
- Two common and broad classes of design:
  - procedural
  - object-oriented

# Procedural Design

- Focus on the *functionality*
- Create a data-flow view of computation



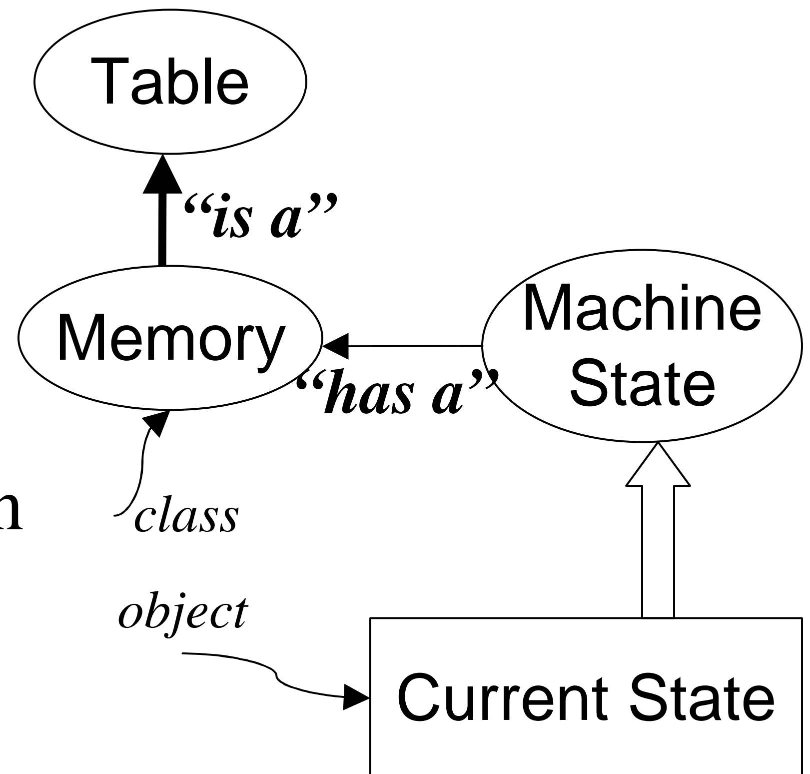

- Use this data-flow to decompose a large system into individual modules
  - hierarchical: modules can then be further decomposed
- Each module responsible for some transformation
  - line of ascii text in file → parsed data in buffer
  - 4 character string of hex digits → integer

# Procedural Design (II)

- When does this process stop?
- Look for modules that:
  - are *small* enough to be easily understood
  - are *large* enough to result in reasonable overall complexity
  - are *generic* (and flexible) enough to be reused
- Key activity: ***DEFINE INTERFACES***
  1. Syntactical structure
     - function name, argument types (i.e., signature)
  2. Behavioural contract
     - requirements and guarantees (i.e., requires/ensures specification)
- Fixing the interfaces allows work to proceed in parallel on the sub-parts

# Object-Oriented Design (OOD)

- Focus on the *data*
- Program = collection of interacting objects
- Sketch out the *types* needed and the *interactions* between these types



# OOD: Finding Classes

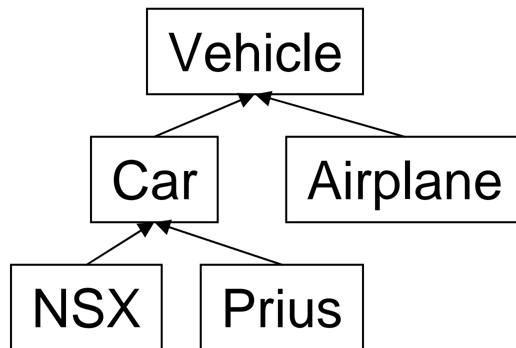
- Design is often based on reality
- Talk to field experts to understand the system being modeled in software.
- Write down scenarios
  - “use case” analysis
- Draw lots of pictures and refine model
- Decide on class invariants

# OOD: Specify Relationships

- Typical class relationships include:
  - inheritance (e.g. “a car *is a* vehicle”)
    - key principle: substitutability and polymorphism
      - a client designed to use type vehicle, can be given a car to use instead
    - therefore, all properties and behaviours of vehicles must apply to cars too
  - containment (e.g. “a car *has a* steering wheel”)
    - key principle: encapsulation
  - use (e.g. “a car *uses a* traffic light”)
- Determine responsibility of each class
  - delegate *where appropriate*
  - benefit of delegation: single point of control/modification
  - cost of delegation: can increase complexity and decrease performance
- Must strike a balance (small vs large) in the *size* and *functionality* of classes

# Digression on Inheritance

- Client designed to use "base class"
- But at run time, client can be given instance of a "derived class" instead




- Cars have different properties than airplanes, but both are vehicles

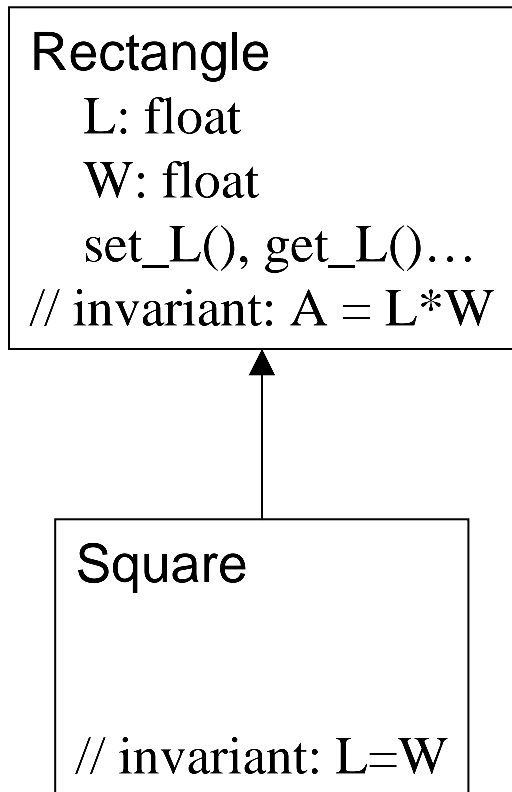
```
class Vehicle {…};
class Car : Vehicle {…};

transport (Person p,
               Place a, Place b,
              Vehicle v)  {
  … }

main()  {
  Car c = new Car();
  transport (paul, CMH, MSP, c);
}
```

- But it isn't always obvious whether such substitution is sound…

# Square is a Rectangle (Restriction)




```
void f (Rectangle r)  {
    float a = r.get_L()*r.get_W();
    r.set_L(2*r.get_L());
    assert (r.area() = 2*a);
    …
}
```

*Rectangles can do things that squares can not!*

# Rectangle is a Square (Extension)

Square
L: float
// invariant: A = L*W

Rectangle
W: float

// invariant: L=W

```
void g (Square s)  {
    float a = s.get_L()*s.get_L();
    assert (s.area() = a);
    …
}
```

*Rectangles can do things that squares can not!*

# OOD: Specify Operations

- Important categories of operations:
  - construct, initialize, copy, assign, destroy
  - access, update, iterate
- Set should be small and independent
  - do not implement every possible use / extension
  - generality may promote reuse, sacrifice performance
- Focus on behavior, not implementation
  - confirm invariants
- Key activity: ***DEFINE INTERFACES***

# Lecture #7

# Testing

1. Philosophy
2. Example
3. “How to”s (including code) and caveats
4. Levels of Testing

# Testing: Philosophy

# Definition of Testing

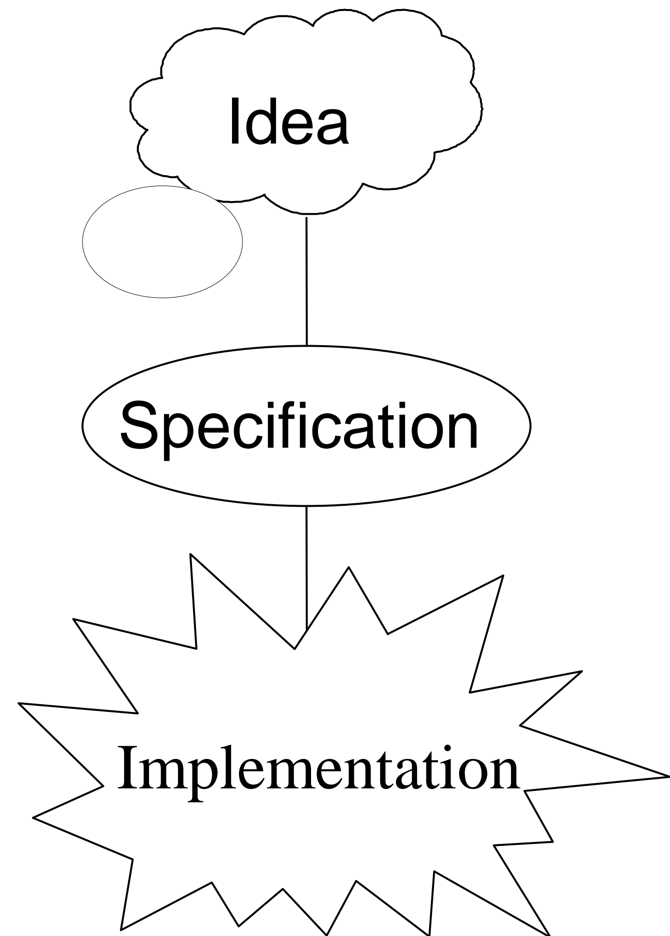
- What is “testing”?
  - A process whereby we *increase our confidence* in an implementation by *observing* its *behavior*
- Fundamental point:
  - testing can detect the *presence* of mistakes, never their *absence*!
- Fail a test case ==>
- Pass *all* test cases ==>

# Importance of Testing

- Despite limitations, testing is the most practical approach for large systems
- Knuth quotation:
  - "Warning: I've only *proven* this algorithm is correct… I haven't *tested* it!"

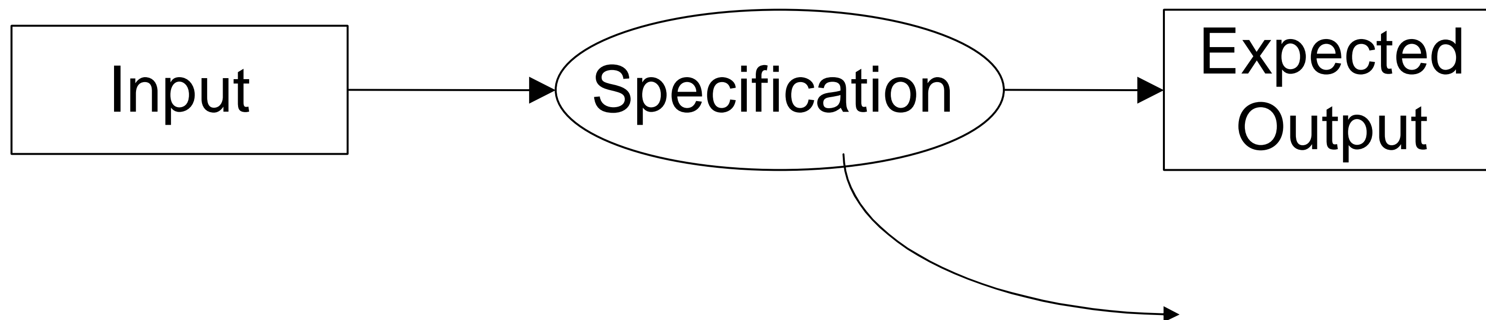
# Theory

- 3 levels of abstraction in functionality
- Want: the idea
- Have: implementation
- “Testing” requires comparing it against something, but what?



# Theory (II)

- Ideal: test against our “idea”
  - but the idea is usually too fuzzy
- So make it concrete by writing specification
  - defines desired mapping from input to output



# Testing: An Example

# Example: Sorting a List

- Idea: function sorts a list in ______________ order
- Spec: `void sort (List x)`

  requires: |x| <= 100

  modifies: x

  ensures:

- Q: do we really need the "expected output"?
  - i.e. why not just look at actual output and see if it is sorted?

# Expected Output

- A:
- Specifications often relate *final* states to *initial* ones
  - but not necessarily true
  - e.g. `void f(int x)`

# Testing: “How To”s and Caveats

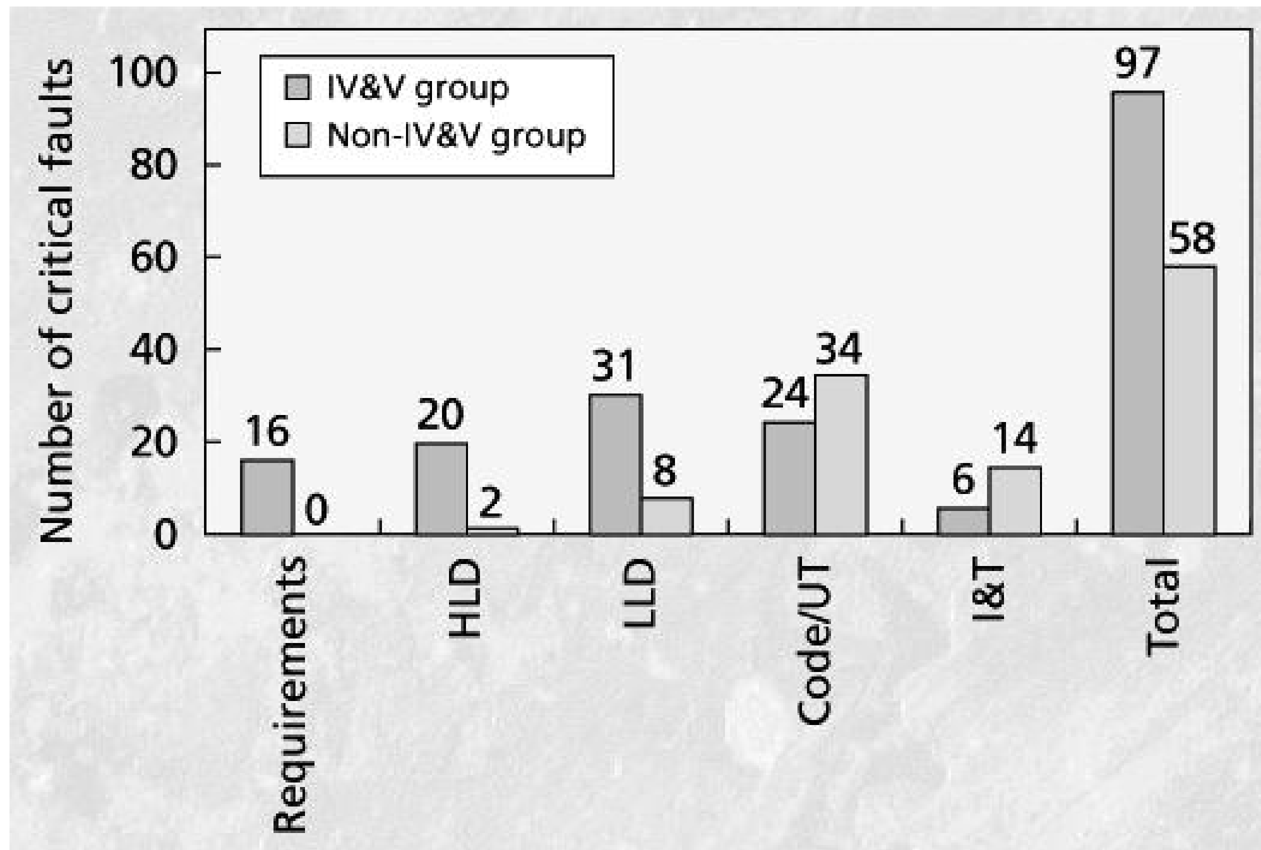
# The Right Frame of Mind

- Tests should be written to *break* a program
  - not to show it works!
- When a test reveals an error, that's success!
- Good approach: have someone else test your code

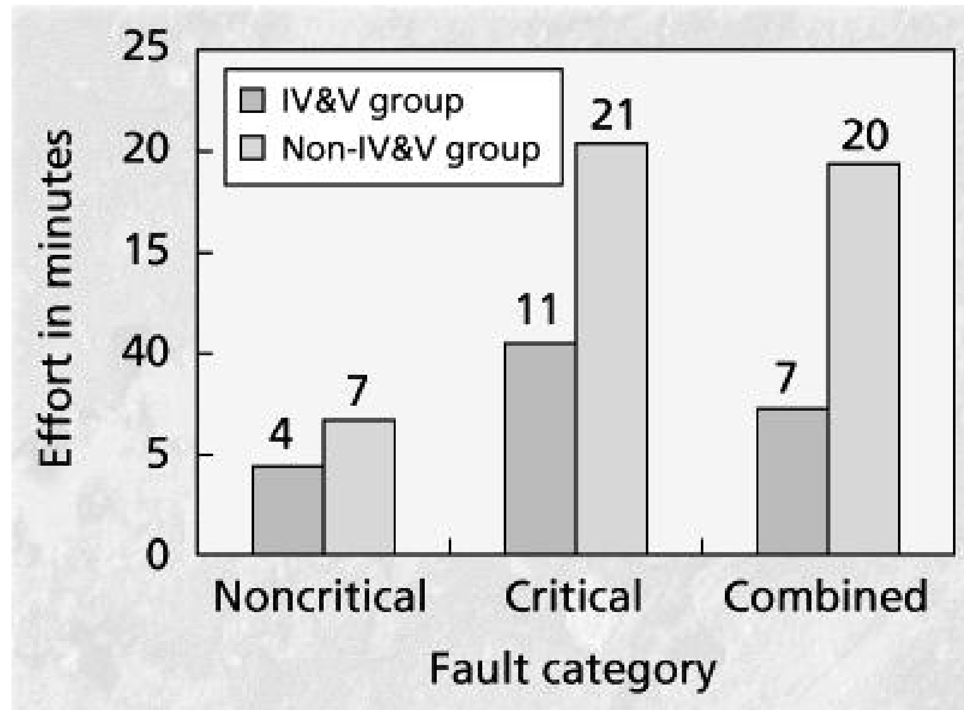
# Importance of Independent Testing

- See IEEE Computer, Oct '99
  - study at NASA Langley
  - had two groups working in parallel
- The group with independent testers found:
  - *more* faults overall (critical and non-critical)
  - found these faults *earlier* in the process
  - fixed these faults with *less effort*

# Figure 1 from Arthur article



# Figure 2 from Arthur article



# Lecture #8

# How To Choose Test Input

- Too many possible inputs to test them all
  - space of possible inputs defined by *requires*
- Important kinds of test input:
  - extremes (e.g. empty list, |x| = 100)
  - trivial / degenerate (|x| = 1, x is already sorted)
  - error-generating
  - different categories (+ve number, -ve numbers)
  - typical input (random list)

# How To Generate Expected Output

1. By hand
   - error-prone and tedious
2. With another program
   1. also error-prone
   2. often just redoing the implementation, and making the same mistakes!
3. Work backwards
   1. an inverse may be easier to calculate
   2. e.g. start with a sorted list, and permute it

# Alternate: Validating Output

- Steps:
  1. Keep a copy of the input
  2. Run the program
  3. Validate the actual output against input
- Example: sorting
  - write two functions:
  - copy the input
  - run program and check:
- Checking functions may be simpler than the full implementation

## *INSERT: Code for Testing Harness*

Insert 1: "Test Driver for Sort"

Insert 2: "Test Suite for Sort"

Insert 3: "Validating Form of Test Driver"

# Dangers with Testing

- “Expected output” is wrong
- Testing program is wrong
  - extra code means more chances to mess up
  - e.g. permute(A,B) always returns true
- With these errors, there are 2 dangers:
  1. spurious test failures
  2. false positives
- Which is worse?

# Dangers with Testing (II)

- A third, more subtle, potential error: The specification is “wrong”
  - how can this be?
  - may not be exposed during testing
  - to increase the chances of finding these problems, have *someone else* test your code!

# Testing: Levels

# Levels of Testing

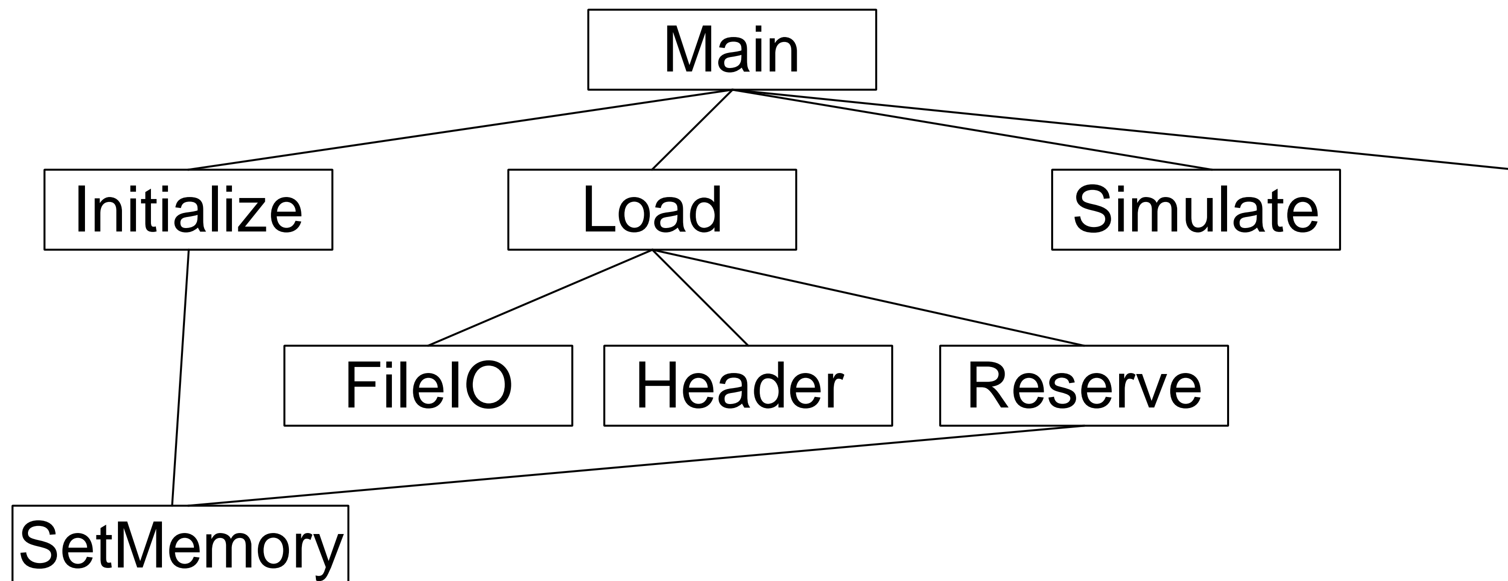
- Typical testing path:
  1. Unit tests
  2. Integration tests
  3. System tests

# Unit Tests

- Individual modules tested in isolation
- Two flavors:
  1. Black box: testing based *only* on specification (tester doesn't even look at code)
  2. White box: testing based on code structure (e.g. tester makes sure every branch of a switch statement is followed)

# Integration Tests

- Modules tested in combination in order to check the *interfaces*
- Best done incrementally



# Bottom-up vs Top-down Testing

Bottom-up

- start with most basic modules
- easy to exercise all the features
- write a “driver” for higher-level modules

Top-down

- start at top (main)
- test interfaces early
- write “stubs” for lower level modules

Often these two occur simultaneously, in tandem

# System Tests

- Verify that system as a whole meets the requirements and specifications
- Three flavors:
  1. alpha: by developers, before release
  2. beta: by “friendly customers”, before general release
  3. acceptance: by end customer, to decide whether or not to hire you next time!

# A Little About Documentation

- See "common errors" link on web page
- Resource: OSU Technical Writing Center
  - see link on web page
- Clarification of some elements of the Programmer's Guide:
  - Data Structures / Types
  - Data Element Dictionary
- Especially important: *shared elements*

# Documenting Data Structures

- Consider a structure representing a cell:

```
struct MemoryCell  {
     char Bit[CellSize];
  }
```

- Give name, declaration, description, invariant, purpose, …

  *invariant*: $(\forall i : 0 \leq i < \texttt{CellSize} : \texttt{Bit}[i] = \text{‘0’} \vee \texttt{Bit}[i] = \text{‘1’})$

- Also use pictures and English to help in the description

# Data Structures (II)

- These are the *shared* data structures
  - used in the declaration of shared variables

```
variable active_cell: MemoryCell;
```

  - used in the declaration of other types

```
type MemoryType: array [256] MemoryCell;
```

  - used as parameter types in function signatures

```
function decode (MemoryCell m): integer
```

- For more OO designs, you have *types*

# Documenting Data Types

**Type**: MemoryCell

*contents*: array of 16 characters

*description*: used to represent a ...

*invariant*: every character is ‘0’ or ‘1’

*operations*:
```
set_bit (int)
read_bit (int) : character
initialize (string)
```

- Then need to specify *each* operation

# Documenting Functions

**Name**: CheckHeaderSyntax()

*description*: This function checks whether or not a given string conforms to the required syntax for header records (see section 2.3.4)

*calling sequence:*

input: char *s - header record to be checked
returns: boolean - true iff header syntax ok

*requires:*

*ensures:*

# Visible State

- Basic principle: “information hiding”
  - hide implementation details from client
    - simplifies interface
    - client shouldn’t rely on these details
- Same principle applies to specifications
  - given in terms of visible (abstract) state
- Ideally, for each shared type, there are *two* kinds of specification: internal & external
  - Often we get lazy and write specs in terms of internal (concrete) state
  - Dangerous, but may be better than no spec at all

# Data Element Dictionary

- Should contain all declared data elements
  - i.e. variables *and* constants
- Non-local elements (shared)
  - group locals based on declaring module
  - highlight global elements in a special section
- Give pertinent information:
  - name, type, declaring module, local/global, description of use, any invariant, value (for constants)