Data Structures

Lecture 01
Software Engineering Principles

Goals

- Describe the general activities in the software life cycle
- Describe the goals for "quality" software
- Explain the following terms: software requirements, software specifications, algorithm, information hiding, abstraction, stepwise refinement
- Explain and apply the fundamental ideas of top-down design
- Explain how CRC cards and UML diagrams can be used in software design

Goals

- Identify several sources of program errors
- Describe strategies to avoid software errors
- Specify the preconditions and postconditions of a program segment or function
- Show how deskchecking, code walk-throughs, and design and code inspections can improve software quality and reduce the software development effort
- Explain the following terms: acceptance tests, regression testing, verification, validation, functional domain, black-box testing, white-box testing

Goals

- State several testing goals and indicate when each would be appropriate
- Describe several integration-testing strategies and indicate when each would be appropriate
- Explain how program verification techniques can be applied throughout the software development process
- Create a C++ test driver program to test a simple class

Software Process

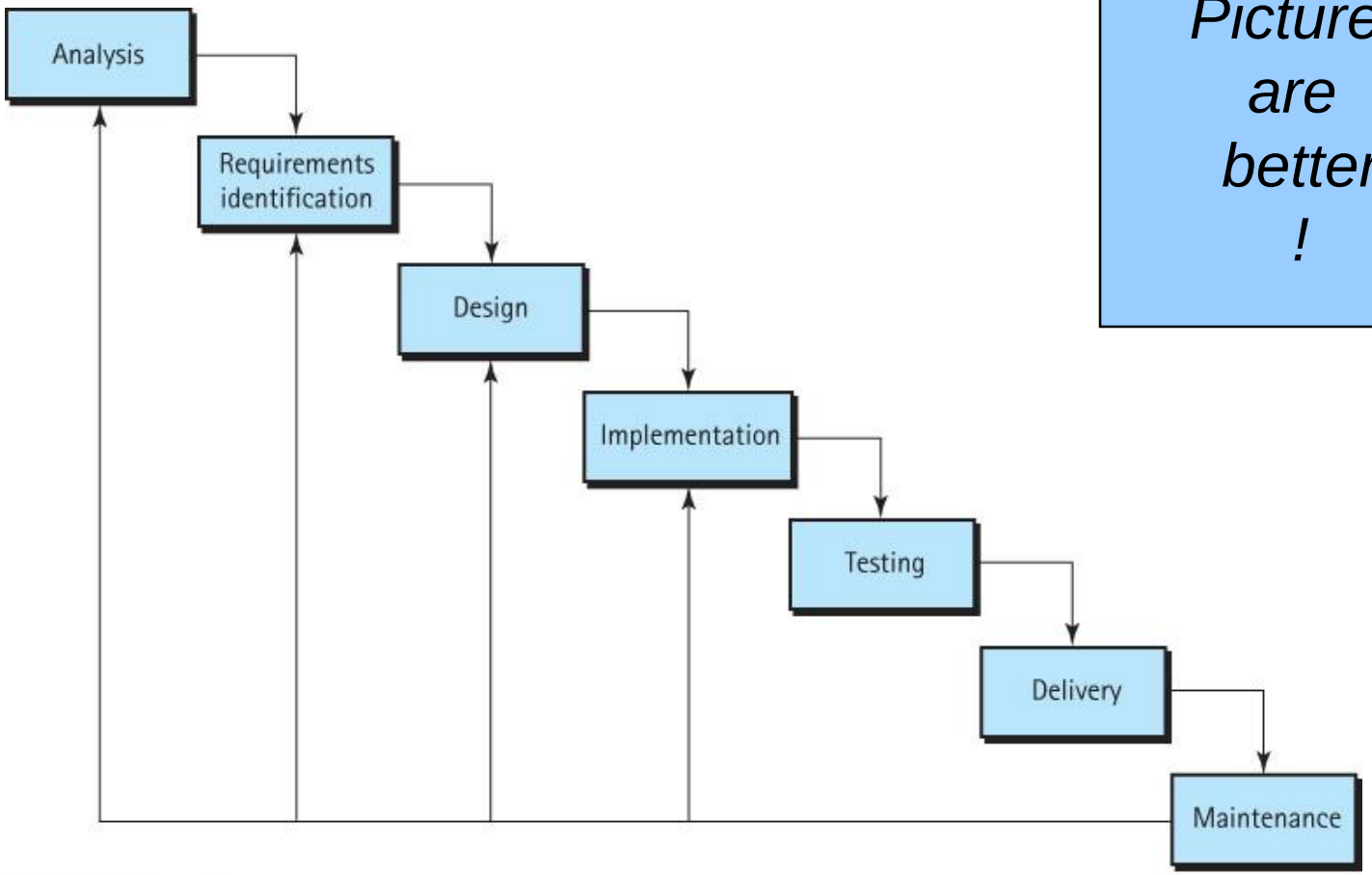
- **Software Engineering**
- A disciplined approach to the design, production, and maintenance of computer programs that are developed on time and within cost estimates, using tools that help to manage the size and complexity of the resulting software products
- **Software Process**
- A standard, integrated set of software engineering tools and techniques used on a project or b an organization

The Software Life Cycle

- Problem analysis
- Requirements elicitation
- Software specification
- High- and low-level design
- Implementation
- Testing and Verification
- Delivery
- Operation
- Maintenance

List are so dull !

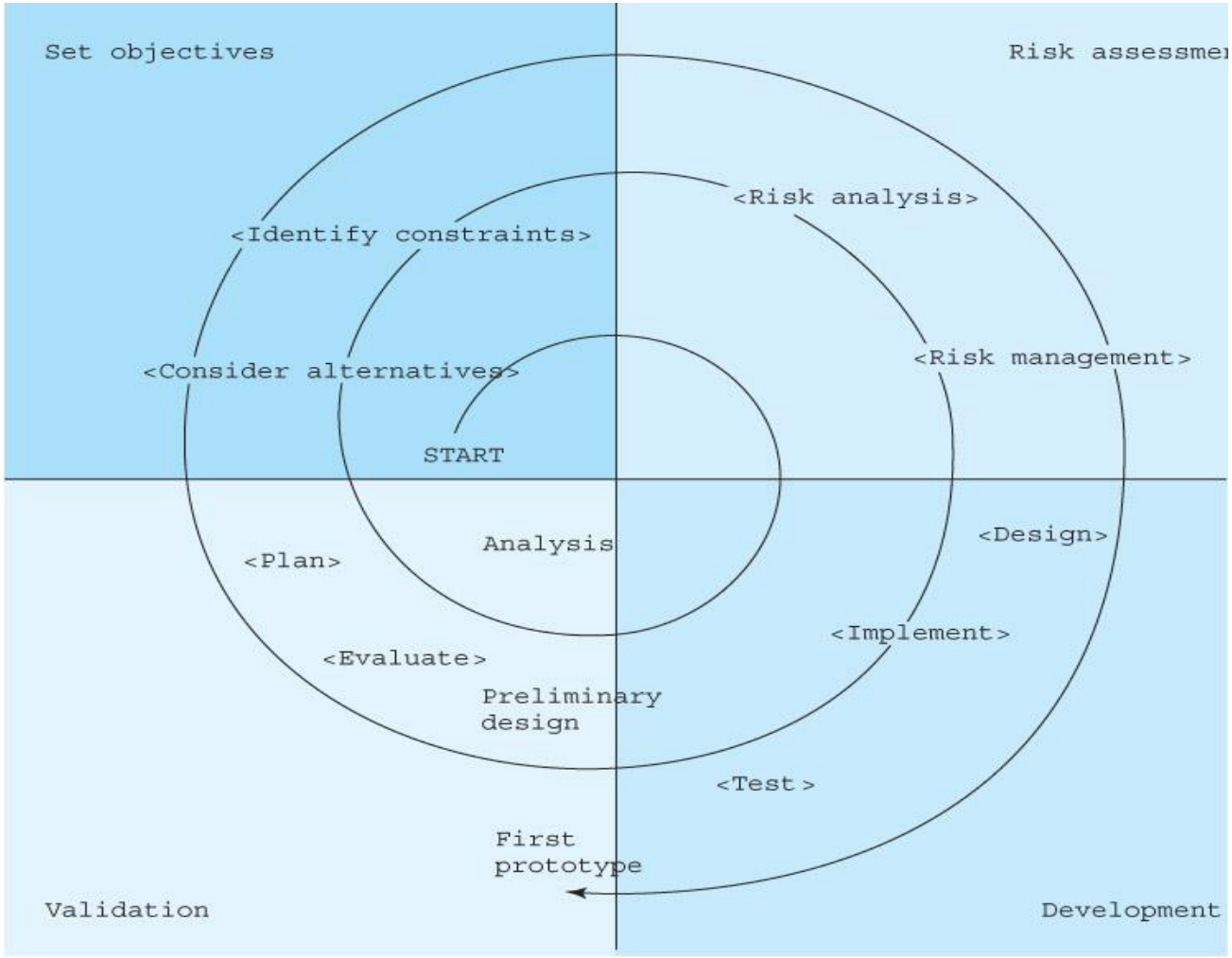
Software Life Cycle



(a) Waterfall model

Pictures are better !

Software Life Cycle



(b) Spiral model

Programmer's Toolbox

- **Algorithm**
- A logical sequence of discrete steps that describes a complete solution to a given problem computable in a finite amount of time

Programmer ToolBoxes

- **Hardware**
- The computers and their peripheral devices
- **Software**
- Operating systems, editors, compilers, interpreters, debugging systems, test-data generators, and so on
- **Ideaware**
- Shared body of knowledge

Including all you learn in this course!

Goals of Quality Software

- It works.
- It can be modified without excessive time and effort.
- It is reusable.
- It is completed on time and within budget.

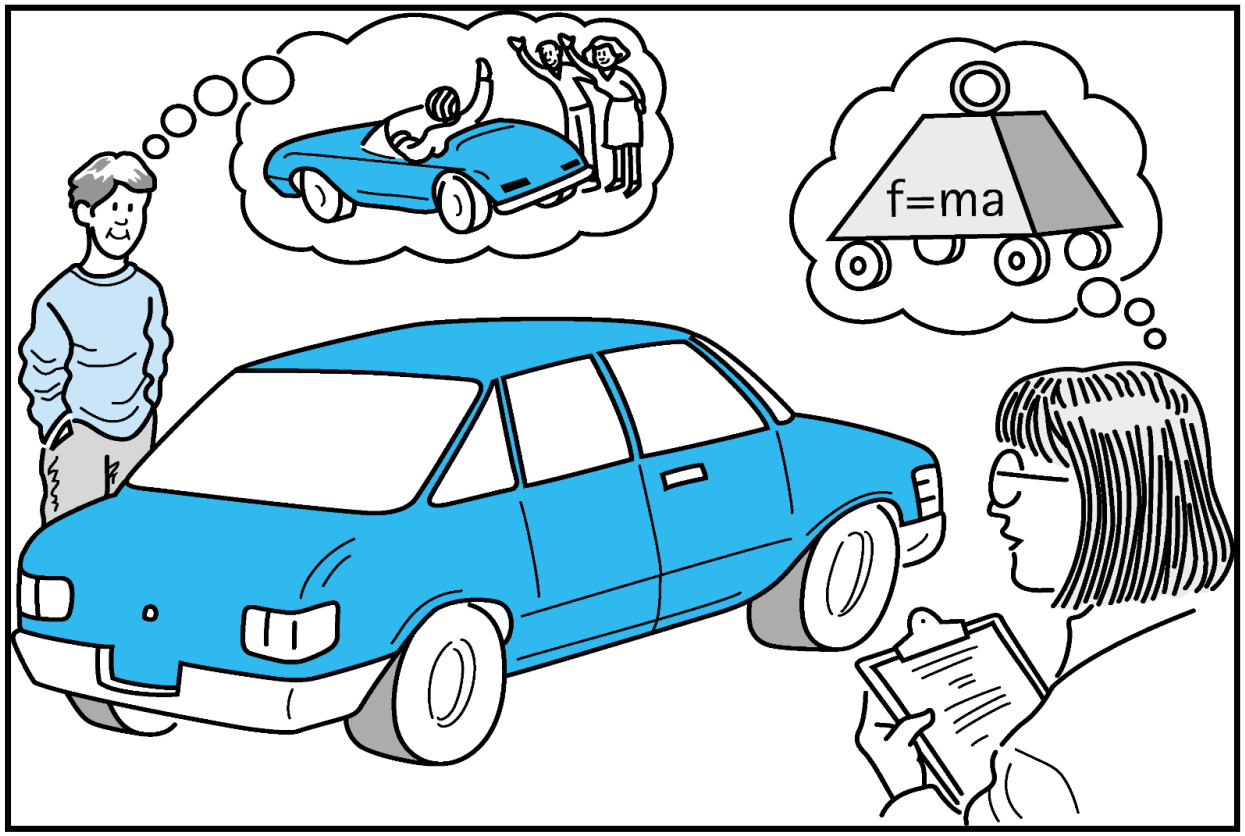
Goals of Quality Software

- **Requirements**
- A statement of what is to be provided by a computer system or software product
- **Software specification**
- A detailed description of the function, inputs, processing, outputs, and special requirements of a software product; it provides the information needed to design and implement the program

Program Design

- **Abstraction**
- A model of a complex system that includes only the details essential to the perspective of the viewer of the system
- Programs are abstractions
- **Module**
- A cohesive system subunit that performs a share of the work; an abstraction tool

Program Design



Program Design

- **Information Hiding**
- The practice of hiding the details of a function or data structure with the goal of controlling access to the details of a module or structure
- **Abstraction and information hiding are fundamental principles of software engineering**
-

Program Design

Abstraction is the most powerful tool people have for managing complexity!

Program Design

- **Stepwise Refinement**
- A problem-solving technique in which a problem is approached in stages and similar steps are followed during each stage, with the only difference being the level of detail involved
  - Top-down
  - Bottom-up
  - Functional decomposition
  - Round-trip gestalt design

Program Design

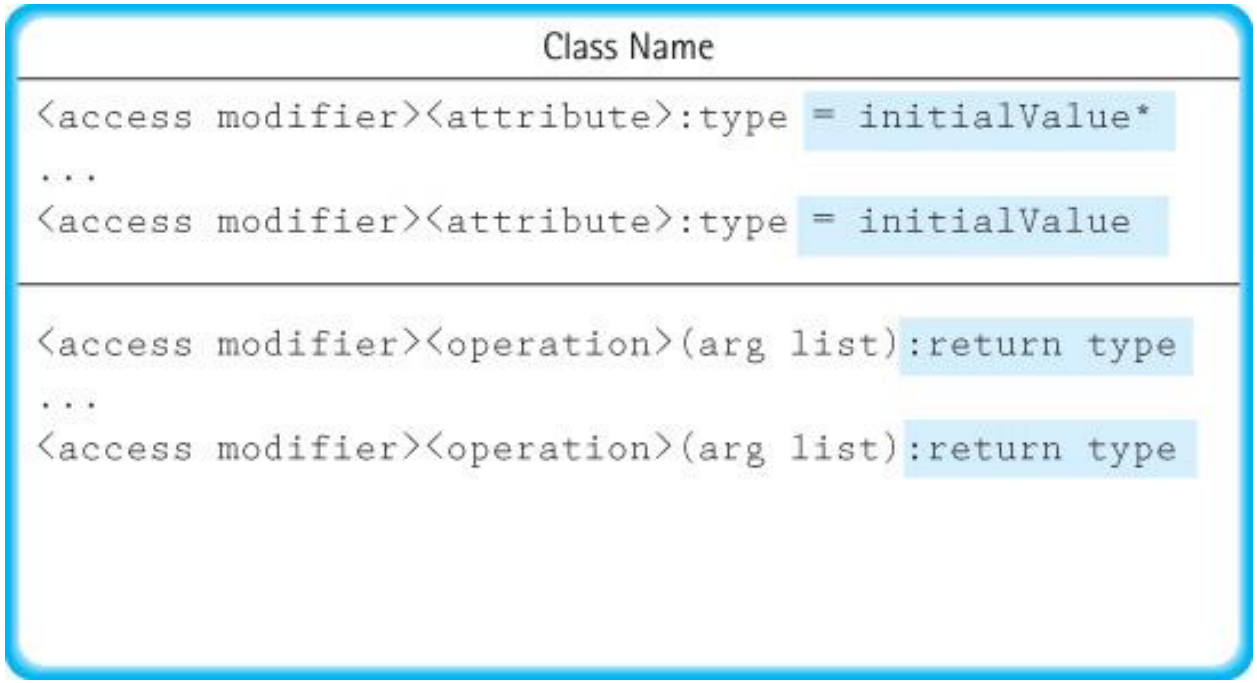
Visual Tools

Program Design

Class Name:	Superclass:	Subclasses:
Primary Responsibilities		
Responsibilities	Collaborations	

CRC card

Program Design

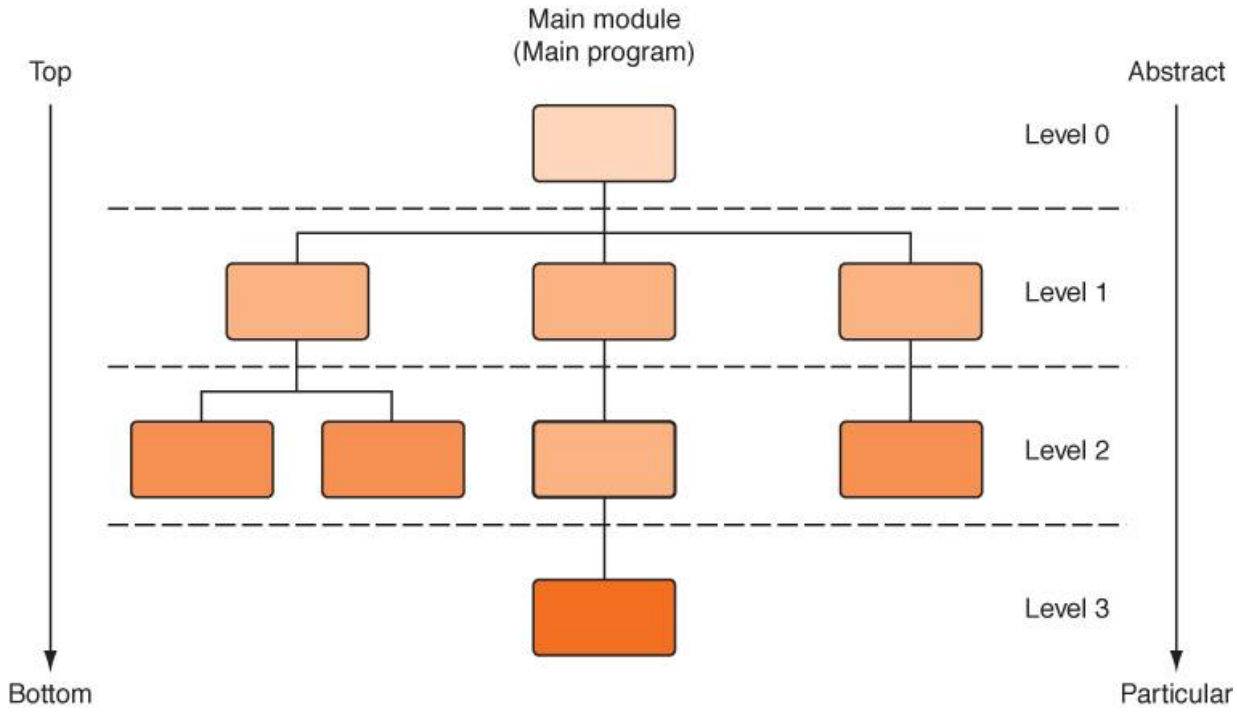


UML diagram

Design Approaches

- **Top-Down Design**
- Problem-solving technique in which the problem is divided into subproblems; the process is applied to each subproblem
- **Abstract Step**
- An algorithmic step containing unspecified details
- **Concrete Step**
- An algorithm step in which all details are specified

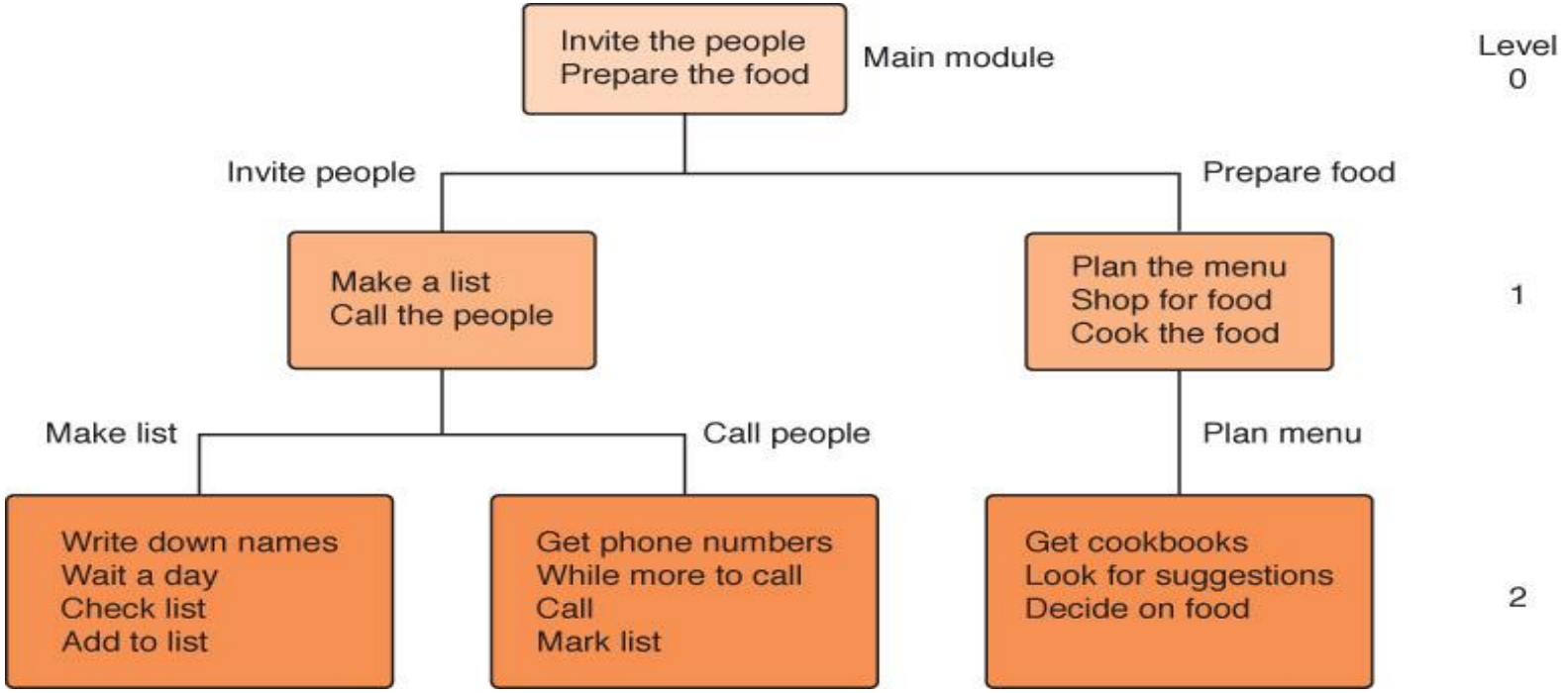
Top-Down Design



- Process continues for as many levels as it takes to make every step concrete
- Name of (sub)problem at one level becomes a module at next lower level

A General Example

- Planning a large party



A Computer Example

- **Problem**
- Create a list that includes each person's name, telephone number, and e-mail address
  - This list should then be printed in alphabetical order
  - The names to be included in the list are on scraps of paper and business cards

A Computer Example

Main Level 0

Enter names and numbers into list
Put list into alphabetical order
Print list

Enter names and numbers into list Level 1

While (more names)
    Enter name
    Enter telephone number
    Enter email address
    Insert information into list

Which steps are abstract? Which steps are concrete? What is missing?

A Computer Example

Enter names and numbers into list (revised) Level 1

```
Set moreNames to true
While (moreNames)
        Prompt for and enter name
        Prompt for and enter telephone number
        Prompt for and enter email address
        Insert information into list
        Write "Enter a 1 to continue or a 0 to stop."
        Read response
        If (response = 0)
                Set moreNames to false
```

Which steps are concrete? Which steps are abstract?

A Computer Example

Prompt for and enter name Level 2

Write "Enter last name; press return."
Read lastName
Write "Enter first name; press return."
Read firstName

Prompt for and enter telephone number Level 2

Write "Enter area code and 7-digit number; press return."
Read telephoneNumber

Prompt for and enter email address Level 2

Write "Enter email address; press return."
Read emailAddress

A Computer Example

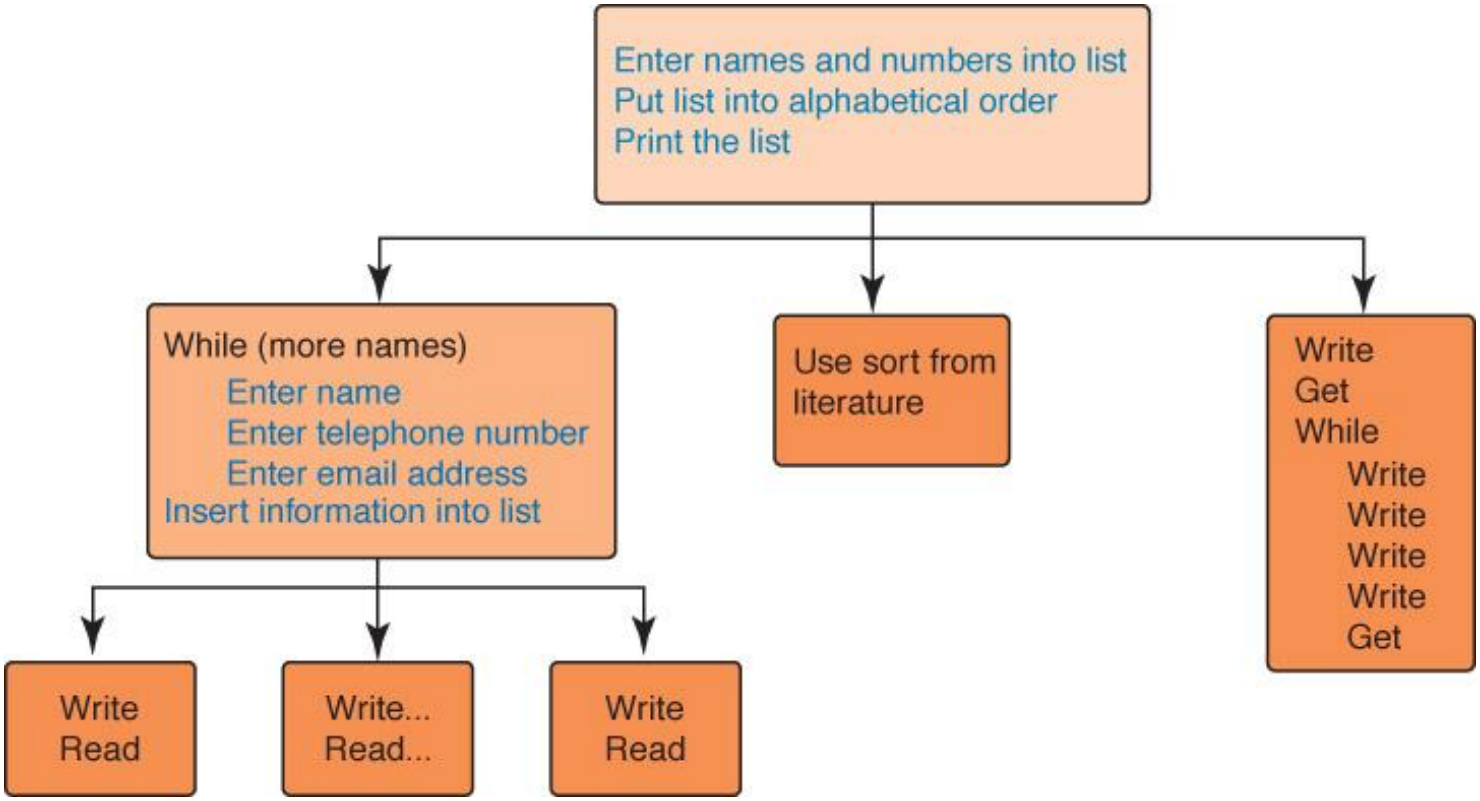
Put list into alphabetical order

Concrete or abstract?

Print the list Level 1

```
Write "The list of names, telephone numbers, and email
      addresses follows:"
Get first item from the list
While (more items)
      Write item's firstName + " " + lastName
      Write item's telephoneNumber
      Write item's emailAddress
      Write a blank line
      Get next item from the list
```

A Computer Example



Note: Insert information is within the loop

Object-Oriented Design

- **Object-oriented Design**
- A problem-solving methodology that produces a solution to a problem in terms of self-contained entities called *objects*
- **Object**
- A thing or entity that makes sense within the context of the problem

  For example, a *student*, a *car*, *time*, *date*

Object-Oriented Design

- **World View of OOD**
- Problems are solved by
  - isolating the objects in a problem,
  - determining their properties and actions (responsibilities), and
  - letting the objects collaborate to solve a problem

What? Say again!

Object-Oriented Design

- An analogy: You and your friend fix dinner
- Objects: you, friend, dinner
- Class: you and friend are people
  - People have name, eye color, …
  - People can shop, cook, …
- Instance of a class: you and friend are instances of class People, you each have your own name and eye color, you each can shop and cook
- You collaborate to fix dinner

Object-Oriented Design

- **Class** (or object class)
- A description of a *group* of objects with similar properties and behaviors; a pattern for creating individual objects
- **Object** (instance of a class)
- A concrete example of the class
- **Classes** contain fields that represent the properties (name, eye color) and behaviors (responsibilities) (shop, cook) of the class
- **Method**
- A named algorithm that defines behavior (shop, cook)

Object-Oriented Design

- Top-Down Design
- decomposes problems into **tasks**
- Object-Oriented Design
- decomposes problems into
- collaborating **objects**

More on OOD in Chapters 2 & 3

Verification of Software Correctness

- A few words to be going on with…
  - Testing
  - The process of executing a program with data sets designed to discover errors
  - Debugging
  - The process of removing known errors

Are these definitions what you expected?

Verification of Software Correctness

- **Testing after you are sure the program works…**
- Acceptance test
- The process of testing the system in its real environment with real data
- Regression testing
- Reexecution of program tests after modifications have been made to ensure that the program still works

Verification of Software Correctness

- **Program verification**
- The process of determining the degree to which a software project fulfills its specifications
- **Program validation**
- the process of determining the degree to which software fulfills its intended purpose

Verification of Software Correctness

Program verification asks,

"Are we doing the job right?"

Program validation asks,

"Are we doing the right job?"

B.W. Boehm, Software Engineering Economics, 1981

Verification of Software Correctness

- Dijkstra decries the term “bugs”

- Ever since the moth was found in the hardware, computer errors have been called *bugs*. Edsger Dijkstra chides us for the use of this terminology. He says that it can foster the image that errors are beyond the control of the programmer—that a bug might maliciously creep into a program when no one is looking. He contends that this is intellectually dishonest because it disguises that the error is the programmer’s own creation.

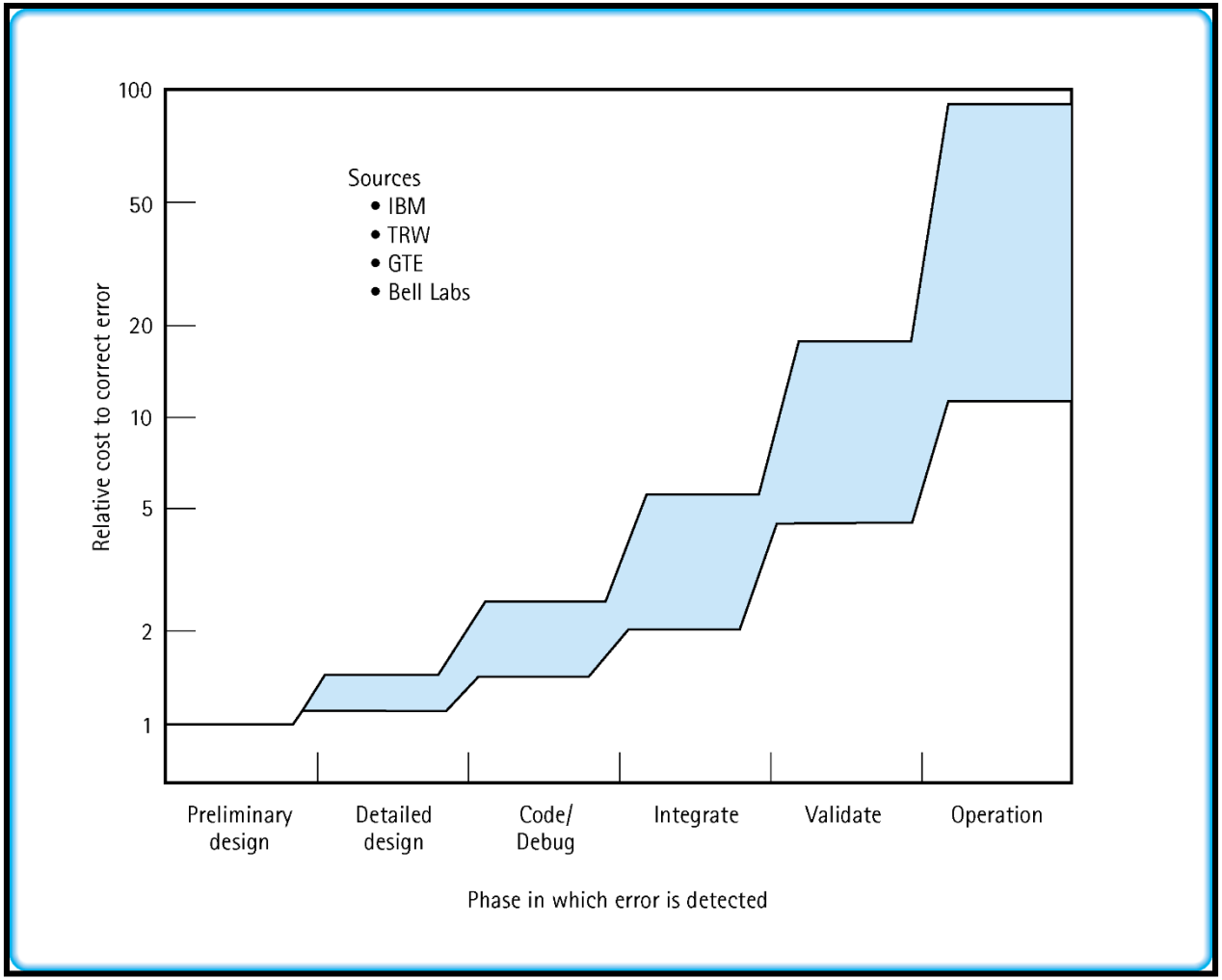
Verification of Software Correctness

Where do you begin to look for errors?

Verification of Software Correctness

- Specification and design errors
  - *Can you give an example?*
- Compile-time errors
  - *Can you give an example*
- Run-time errors (data)
  - *Can you give an example?*
- **Robustness**
- The ability of a program to recover following an error; the ability of a program to continue to operate within its environment

Verification of Software Correctness



Cost of errors goes up the later in the process you find them !

Verification of Software Correctness

- **Preconditions**
- Assumptions that must be true on entry into an operation or function for the postconditions to be guaranteed.
- **Postconditions**
- Statements that describe what results are to be expected at the exit of an operation or function, assuming that the preconditions are true

Preconditions and postconditions are warning signs!

Verification of Software Correctness

- **Desk checking**
- Tracing an execution of a design or program on paper
- **Walk-through**
- A verification method in which a team performs a manual simulation of the program or design
- **Inspection**
- A verification method in which one member of a team reads the program or design line by line and the other members point out errors

Verification of Software Correctness

- **Exceptions**
- An unusual, generally unpredictable event, detectable by software or hardware, that requires special processing; the event may or may note be erroneous
- Three parts to an exception me
  - **Define** the exception
  - **Generate** (raising) the exception
  - **Handling** the exception

Way to take care of possible problems in advance in the code

Program Testing

Remember the definition of testing?

Testing is the process of executing a program with data sets designed to discover errors

How do we create appropriate data sets?

Program Testing

- For each data set
  - Determine inputs that demonstrate the goal of test case
  - Determine the expected behavior of the program
  - Run the program and observe the resulting behavior
  - Compare the ex
    the actual behav

Yes, but how do we determine the goals?

Program Testing

Unit testing

Testing a module, class, or function by itself

Functional domain

The set of valid input data for a program or function

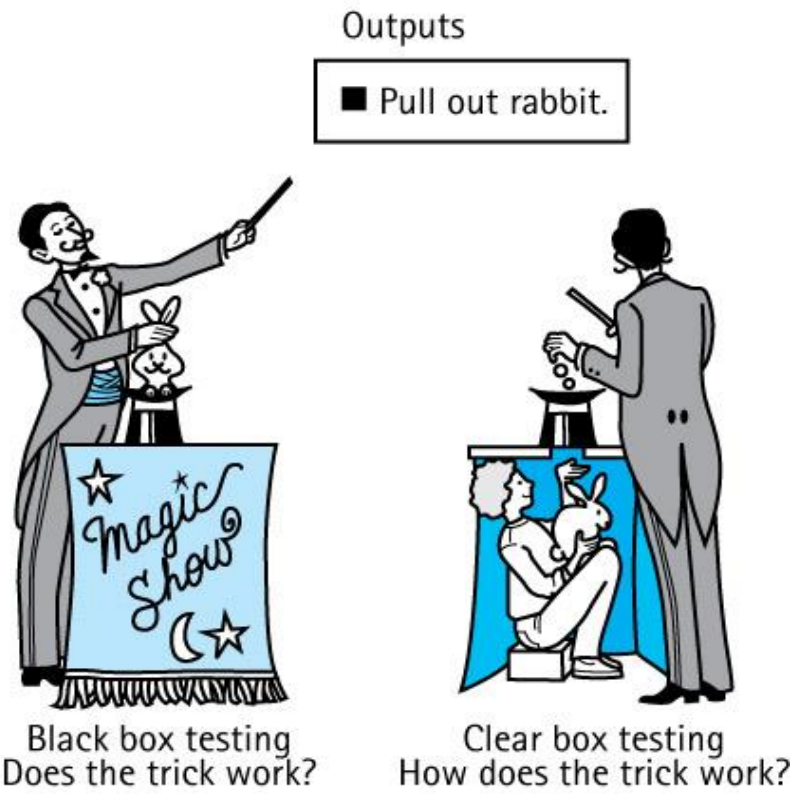
Black-box testing

Testing a program or function based on the possible input values, treating the code as a “black box”

Clear (white) box testing

Testing a program or function based on covering all of the statements, branches, or paths of the code

Program Testing



See the difference?

Program Testing

Statement coverage

Every statement in the program or function is executed at least once

Branch

A code segment that is not always executed; for example, a *switch* statement has as many branches as there are case labels

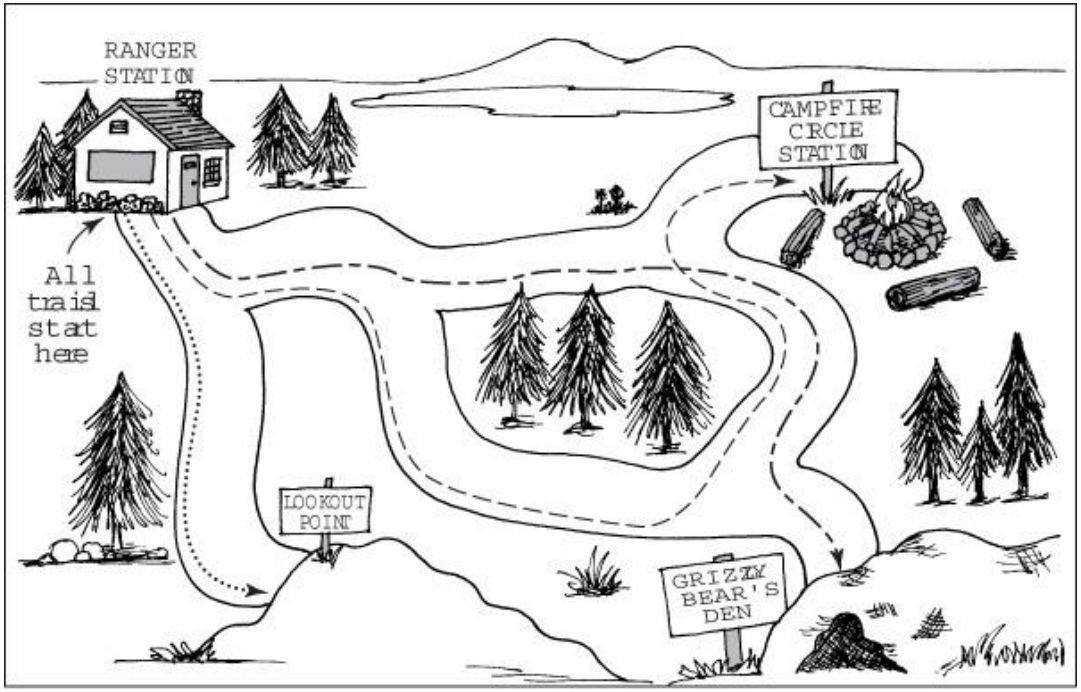
Path

A combination of branches that might be traversed when a program or function is executed

Path testing

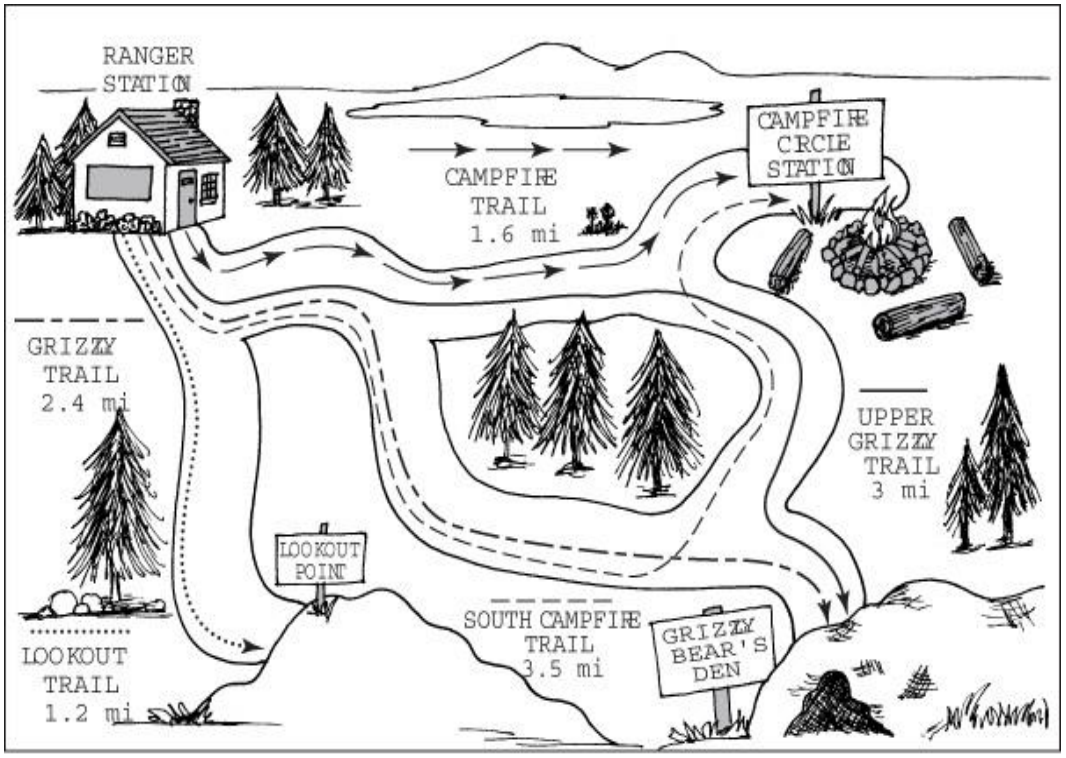
A testing technique whereby the tester tries to execute all possible paths in a program or function

Program Testing



Checking out all the branches

Program Testing



See the difference between branch checking and path checking ?

Checking out all the paths

Program Testing

- **Test plan**
- Document showing the test cases planned for a program or module, their purposes, inputs, expected outputs, and criteria for success
  - Goals
  - Data to test goals
  - Expected output
- **Implementing a test plan**
- Running the program with the test cases listed in the test plan

Program Testing

- **In summary,**
- For program testing to be effective, it must be planned
- Start planning for testing before writing a single line of code
- A program is not complete until it has been thoroughly tested
- Your test plan should convince a reader that the program is correct

Program Testing

- *Write a black box test plan for a function that takes three*
- *integer values that represent the sides of a triangle and*
- *returns "isosceles," "equilateral", or "scalene"*

- *How many test cases must there be?*
- *Write a driver for your function*

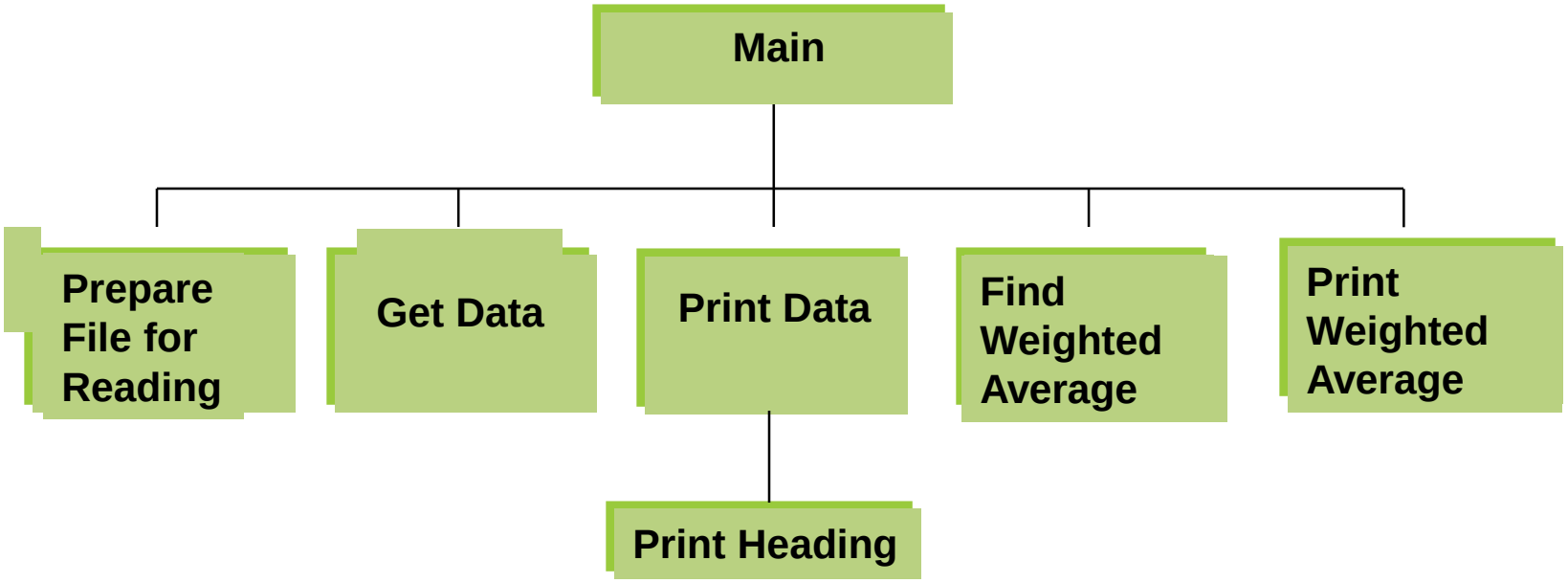
Test driver

A program that sets up the testing environment by declaring and assigning initial values to variables, then calls the program to be tested

Program Testing

Integration testing

Testing performed to integrate program modules that have already been independently unit tested



Program Testing

Top-Down Integration

Ensures correct overall design logic

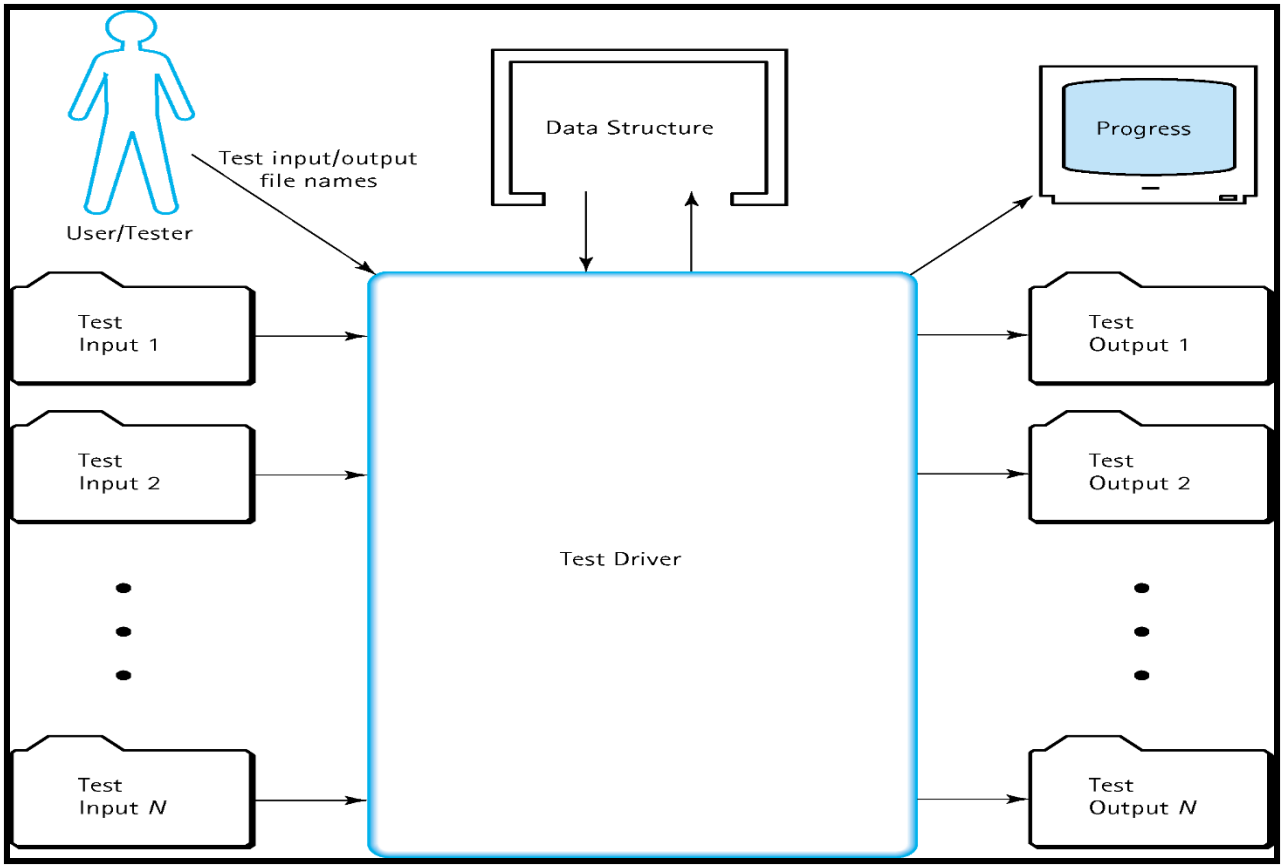
USES: placeholder module "stubs" to test the order of calls

Bottom-Up Integration

Ensures individual modules work together correctly, beginning with the lowest level.

USES: a test driver to call the functions being tested.

Program Testing



Testing architecture for data structures

Algorithm for test driver

- *Declare an instance of the class being tested*
- *Prompt for, read the input file name, and open the file*
- *Prompt for, read the output file name, and open the file*
- *Prompt for and read the label for the output file*
- *Write the label on the output file*
- *Read the next command from the input file*
- *Set numCommands to 0*
- *While the command read is not 'quit'*
- *Execute the command by invoking the member function of the same name*
- *Print the results to the output file*
- *Increment numCommands by 1*
- *Print "Command number" numComands "completed" to the screen*
- *Read the next command from the input file*
- *Close the input and output files.*
- *Print "Testing completed" to the screen*

Life-Cycle Verification Act

Analysis	Make sure that specifications are completely understood. Understand testing requirements.
Specification	Verify the identified requirements. Perform requirements inspections with your client.
Design	Design for correctness (using assertions such as preconditions and postconditions). Perform design inspections. Plan the testing approach.
Code	Understand the programming language well. Perform code inspections. Add debugging output statements to the program. Write the test plan. Construct test drivers and/or stubs.
Test	Unit test according to the test plan. Debug as necessary. Integrate tested modules. Retest after corrections.
Delivery	Execute acceptance tests of the completed product.
Maintenance	Execute regression test whenever the delivered product is changed to add new functionality or to correct detected problems.

C++ Tips

- `<iostream>` is header file for a library that defines three stream objects
- **Keyboard**
- an `istream` object named `cin`
- **Screen**
- an `ostream` object named cout
- an `ostream` object named cerr used for error reports

C++ Tips

- **Insertion Operator (<<)**
- An output (`ostream`) operator that takes 2 operands
-
  - The left operand is a stream expression, such as `cout`
  - The right operand is an expression describing what to insert into the output stream

cout << "The book costs" << cost;

C++ Tips

- **Extraction Operator (>>)**
- An input (`istream`) operator that takes 2 operands
  - The left operand is a stream expression, such as **`cin`**
  - The right operand is a variable of simple type
- Operator `>>` attempts to extract the next item from the input stream and store its value in the right operand variable
- cin >> cost;

The value being keyed in for `cost` must be the same type as that declared for variable `cost`

C++ Tips

- **Whitespace characters**
- Characters such as blanks, tabs, line feeds, form feed, carriage returns, and other characters that you cannot see on the screen
- Extraction operator >> “skips” leading whitespace characters
- before extracting the input value from the stream
- Use function `get` to read the next character in the input stream
- `cin.get(inputChar);`

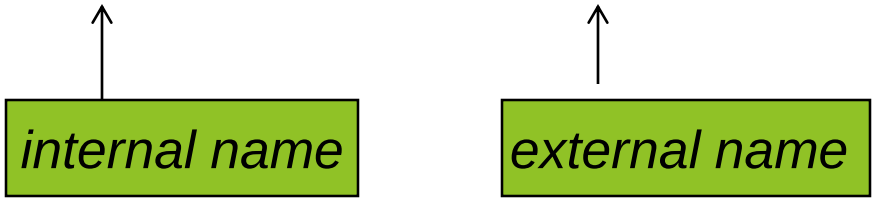
`get` returns one character

C++ Tips

```
#include  <iostream>
int  main( )
{
  using namespace std;
  int   partNumber;
  float unitPrice;
  cout << "Enter part number followed by return: "
  <<  endl ;                    // prompt
  cin  >> partNumber ;
  cout << "Enter unit price followed by return: "
  <<  endl ;
  cin  >>  unitPrice ;
  cout <<  "Part # "  <<  partNumber  // echo
  <<  "at Unit Cost: $ " << unitPrice << endl ;
  return 0;
}
```

C++ Tips

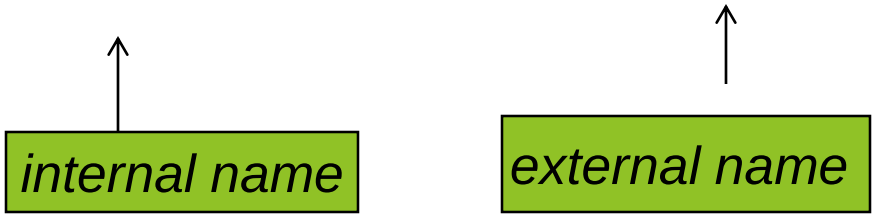
- **File input/output**
- `#include <fstream>` provides
- ofstream (output file) uses <<
- ifstream (input file) uses >>
- `ifstream myFile;`
- `myFile.Open("Data");`



`Open` binds internal name to external name

C++ Tips

- `ofstream yourFile;`
- `yourFile.Open("DataOut");`



`yourFile.Close();`

closes the file properly

C++ Tips

- Statements for using file I/O
- `#include <fstream>`
- `using namespace std;`
- `ifstream  myInfile;  // declarations`
- `ofstream  myOutfile;`
-
- `myInfile.open("A:\\myIn.dat"); // open files`
- `myOutfile.open("A:\\myOut.dat");`

- `myInfile.close( );  // close files`
- `myOutfile.close( );`

C++ Tips

- What does opening a file do?
  - associates the C++ identifier for your file with the physical (disk) name for the file
  - if the input file does not exist on disk, open is not successful
  - if the output file does not exist on disk, a new file with that name is created
  - if the output file already exists, it is erased
  - places a *file reading marker* at the very beginning of the file, pointing to the first character in it

C++ Tips

- **Stream failure**
- File enters the fail state
  - further I/O operations using that stream are ignored
  - Computer does not automatically halt the program or give any error message
- Possible reasons for entering fail state include
  - invalid input data (often the wrong type)
  - opening an input file that doesn't exist
  - opening an output file on a diskette that is already full or is write-protected.

C++ Tips

```
#include  <fstream>
#include  <iostream>
using namespace std;
int  main( )
{// CHECKS FOR STREAM FAIL STATE
  ifstream inFile;
  inFile.open("input.dat"); // try to open file
  if ( !inFile )
  {
    cout  <<  "File input.dat could not be opened.";
    return  1;
  }
    // rest of the program
  return   0;
}
```

C++ Tips

- Reading in file names
- `ifstream inFile;`
- `string fileName`
- `cout << "Enter the name of the input file"`
- `     << endl;`
- `cin  << fileName;`
- `inFile.open(fileName.c_str());`

Why not just fileName?