SOFTWARE ENGINEERING PRINCIPLES

The Software Process

- How do we write complex programs?
- Focus on the *software life cycle*
- Develop techniques of software engineering

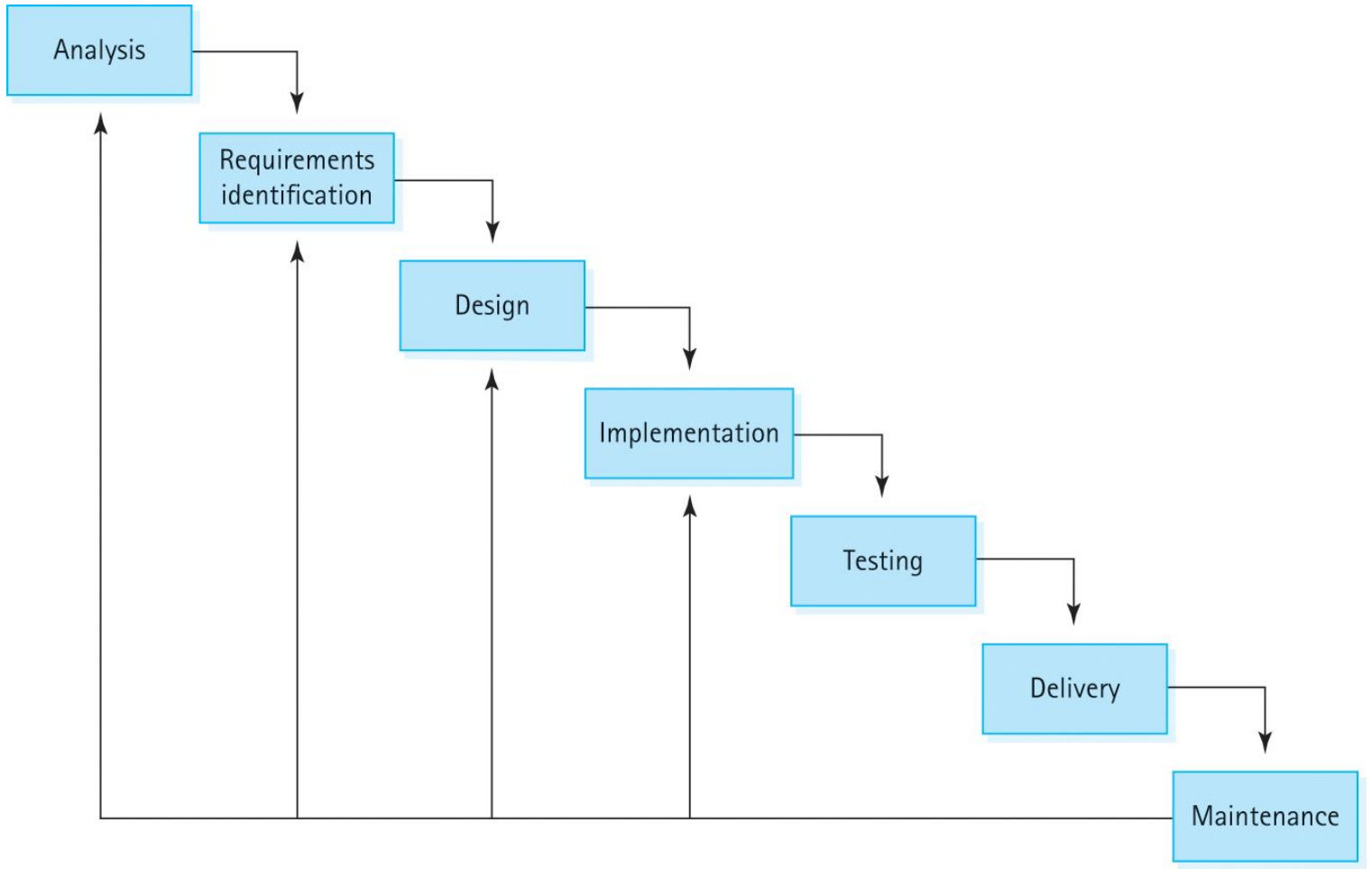
Software Engineering

- A disciplined approach to the design, production, and maintenance of computer programs
- Use tools that help manage the size and complexity of the resulting software
- Ensure that projects are developed on time and within cost estimates

The Software Life Cycle

- Problem analysis
- Requirements elicitation
- High- and low-level design
- Implementation of the design
- Testing and verification
- Delivery
- Operation
- Maintenance

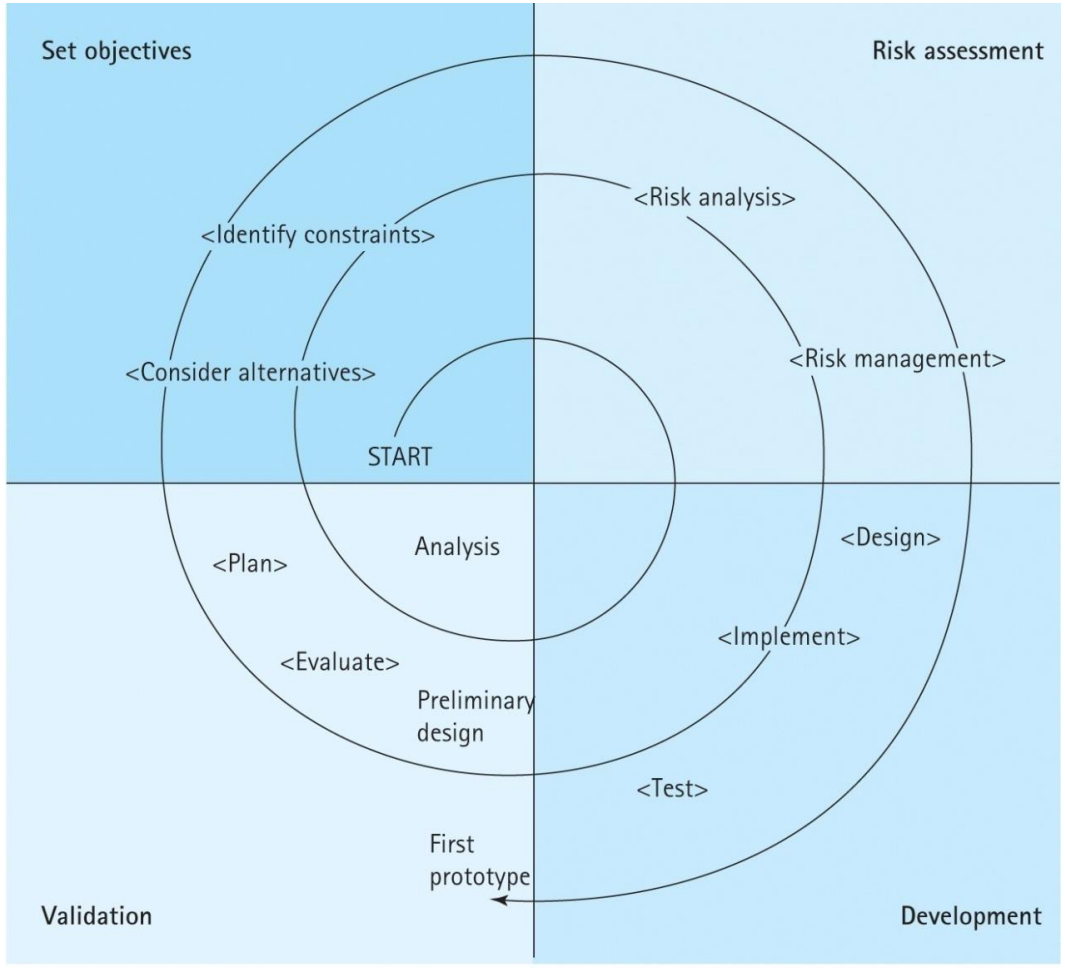
Waterfall Model



(a) Waterfall model

Figure 1.1a Life-cycle models: (a) Waterfall model (b) Spiral model

Spiral Model



(b) Spiral model

Figure 1.1b Life-cycle models: (a) Waterfall model (b) Spiral model

Programmer Toolboxes

- The various tools and techniques programmers use to create software
- **Hardware:** The computers and other devices
- **Software:** Operating systems, development environments, compilers, debuggers…
- **Ideaware:** The shared body of programming knowledge, such as useful algorithms and programming methodologies

An Algorithm Is…

- A logical sequence of discrete steps that describes a complete solution to a problem, computable in a finite amount of time.

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

Quality Software Works

- **Requirements:** Describe what service the software should provide
- **Software Specification:** A written document that tells *what* a program does but not *how*
- Quality software must be complete and correct, and do everything as explained in the specification

Quality Software Can Be Modified

- Changes may occur during the design, coding, and testing phases
- Bug fixes and other small changes are made during the maintenance phase
- Easily modified programs are readable, well documented, and easily understood by humans

Quality Software Is Reusable

- Spend extra effort to make components more general now in order to save time later
- Using code that is already designed, implemented, and tested allows programmers to focus on the rest of the program
- This does require extra effort during specification and design phases

Quality software is completed on time and within budget

- Programmers must be able to work efficiently
- Missing a deadline can have significant impact on other projects, people, and companies
- “Time is money”

Software Specifications

- The **specification** describes what services a program should provide, based on the users' requirements
- **Scenario:** Sequence of events describing one complete execution of the program
- Useful questions:
  - What is the expected input and output?
  - How will errors be handled?
  - How will the program be used?

Abstraction

- A model of a complex system that includes only the essential details
- Programs are abstractions
- Simplify work by hiding complex details

Information Hiding

- The practice of hiding the details of a module with the goal of controlling access to the details of a module
- **Module:** A cohesive system subunit that performs some share of the work
- Information hiding allows programmers to focus on one module at a time
- Each module should serve a single purpose

Stepwise Refinement

- Approaching a problem in stages, step by step
- **Top-down:** Break problem into pieces, deferring details as long as possible
- **Bottom-up:** Focus on details first and build up to the high-level components
- **Functional Decomposition:** Approach a program as a set of cooperating functions
- **Round-trip Gestalt:** Top-down with emphasis on objects and data

Visual Tools

- Abstraction, information hiding, and stepwise refinement help manage complexity
- **Visual Tools:** Help us visualize the design
- **UML Diagrams:** Define components and the relationships between them

Design Approaches

- How do we define the modules of a project?
- Or, how do we break up a project into modules to work on?

Top-Down Design

- Starts with a “big-picture” view of the program
- Then, break the picture down into tasks
- Continue breaking down the tasks until they can be easily described in code
- **Functional decomposition** breaks the program into functions that form a complete solution to the problem when used together

Top-Down Design: Cake

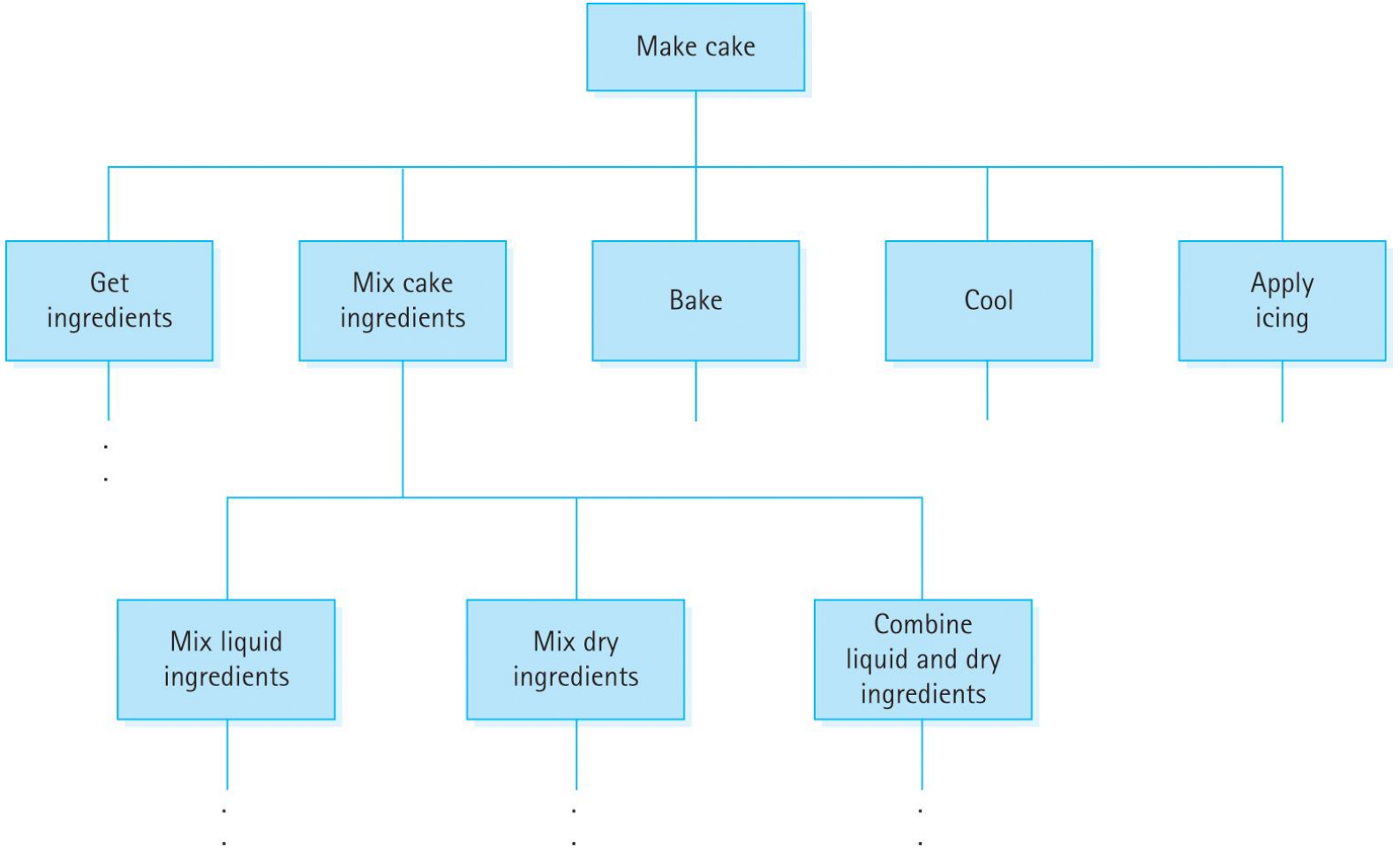


Figure 1.4 A portion of a functional design for baking a cake

Objected-Oriented Design

- Divide-and-conquer, but break down the program into *things* instead of *tasks*
- The design consists of objects, which are defined by classes
- Objects combine data and operations

Classes for Baking a Cake

Table 1.1 Example of Object Classes That Participate in Baking a Cake

Class	Attributes	Responsibilities (Operations)
Oven	Energy source	Turn on
	Size	Turn off
	Temperature	Set desired temperature
	Number of racks	
Bowl	Capacity	Add to
	Current amount	Dump
Egg	Size	Crack
		Separate (white from yolk)

Table 1.1 Example of Object Classes That Participate in Baking a Cake

Procedural vs. Object Oriented

“Read the specification of the software you want to build. Underline the verbs if you are after procedural code, the nouns if you aim for an object-oriented program.”

Grady Booch, “What Is and Isn’t Object Oriented Design,” 1989.

Verification of Software Correctness

- **Testing:** Executing a program with input designed to find errors
- **Debugging:** Investigating and removing known errors from a program
- **Acceptance Test:** Testing in a real environment with real data
- **Regression Test:** Testing that a program is still correct after being modified

Verification and Validation

- Is testing enough?
- **Program Verification:** Checking that a program fulfills its specification
  - “Are we doing the job right?”
- **Program Validation:** Checking that the program fulfills its intended purpose
  - “Are we doing the right job?”

The Origin of Bugs

- Where do bugs occur in a program?
- Is there a way to avoid bugs entirely?
- What “detective work” is needed to find bugs?

Specification and Design Errors

- There may be errors in the specification itself
- This is a result of poor communication or a misunderstanding of users' needs
- Bugs are much cheaper to fix while in the specification stage
- Once the program has been written and deployed, fixing any errors is very expensive

Cost of Specification Errors

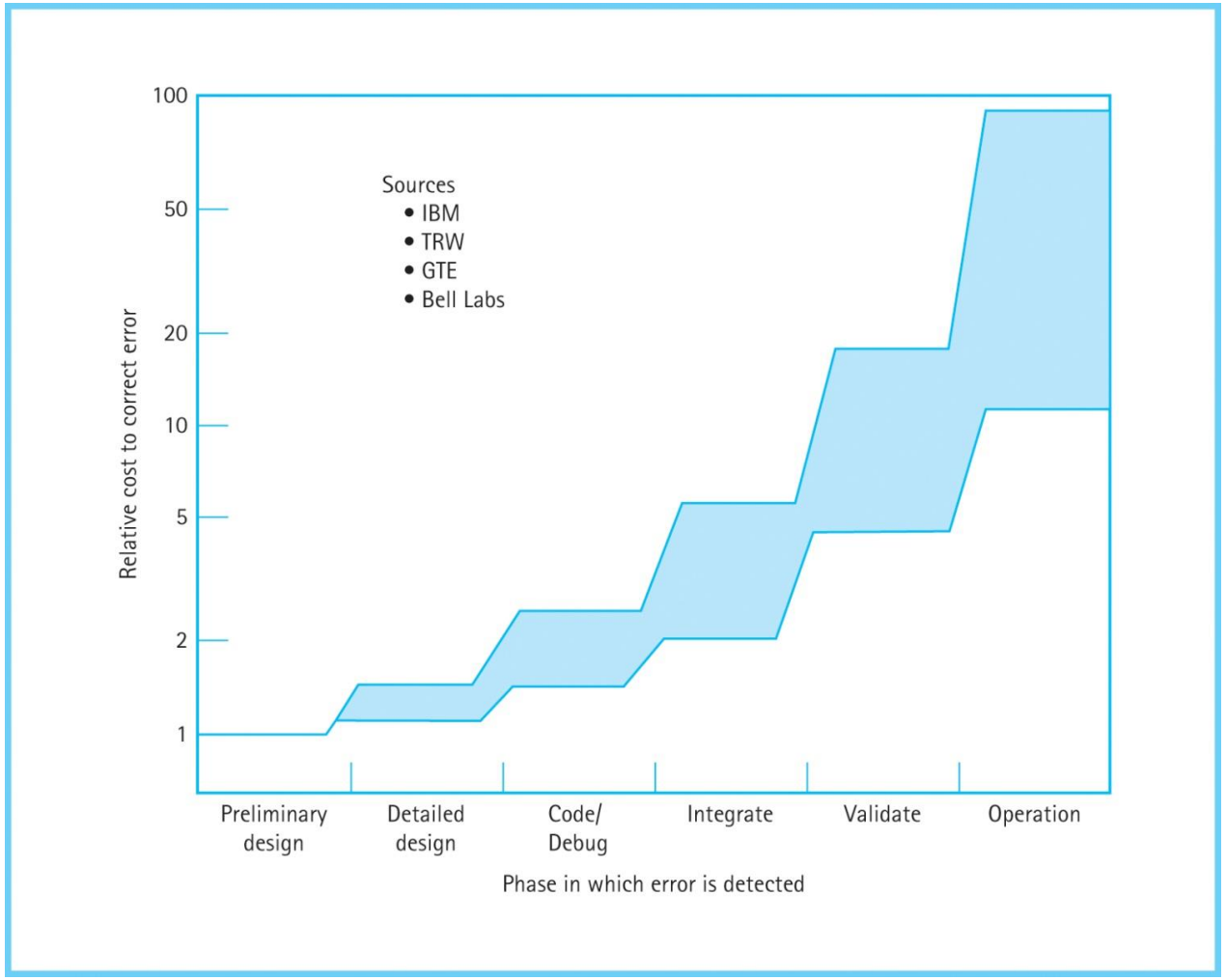


Figure 1.5 This graph demonstrates the importance of early detection of software errors

Compile-Time Errors

- It’s not impossible to write clean, syntactically correct code on the first try
- These errors are generally easier to handle than other types of errors
- Knowing your tools (programming language, editor, etc.) will help avoid these errors

Run-Time Errors

- An error that occurs during execution, often causing the program to crash
- **Robustness:** The ability of a program to recover from an error
- Run-time errors can often be found with sufficient testing
- Errors should be handled gracefully instead of allowing the program to crash

Designing for Correctness

- We would like to prove a program is correct
- Make **assertions** (logical propositions) about what a program should do
- Then, prove the design fulfills the assertions

Conditions on Functions

- Assertions can be applied at the module level
- **Precondition:** A condition that must be true before an operation is executed
- **Postcondition:** A condition that will be true after an operation completes
- The postcondition will be true if the precondition is true

Design Review Activities

- **Deskchecking:** Tracing an execution of a design or program on paper
- **Walk-through:** A *team* performs a manual simulation of the program or design
- **Inspection:** One member of a team reads the program or design line by line while others point out the errors

Exceptions

- **Exceptions** allow programs to interrupt normal control flow to handle exceptional situations
- Handling them should be part of the design
- Different languages have different facilities for handling exceptions

Program Testing

- Testing is used to find remaining errors
- **Test Case:** A single test used to check a particular program feature. For each case:
  - Determine inputs
  - Determine the expected behavior of the program
  - Run the program and observe its behavior
  - Compare the expected behavior and the actual behavior

Unit Testing and Coverage

- **Unit Testing:** Testing a program module or function individually
- It’s hard to say how many unit tests are needed
- Two coverage approaches:
  - Cases based on possible inputs
  - Cases based on aspects of the code
- **Metric-based Testing:** Goals are based on measurable factors like coverage

Data Coverage

- Use a wide range of inputs to test the behavior of a module
- Exhaustive coverage means to test all possible inputs, which is usually impossible for most function domains
- The general approach is to cover typical cases and the edge cases
- **Black-box Testing:** Test based on the inputs, ignoring the actual code

Code Coverage

- Use tests that evaluate as many parts of the module as possible
- **Branch:** A code segment that is not always executed
- **Path:** A sequence of branches that is traveled during execution
- **White-box Testing:** Testing based on covering all statements, branches, or paths in a module

Test Plans

- A document showing test cases for a module, their purposes, inputs, expected outputs, and the criteria for success
- For program testing to be effective, **it must be planned**
- Start planning for testing before writing a single line of code
- **Test Driver:** Program that runs the test plan

Planning for Debugging

- We can plan for debugging from the start
- Provide a global output file to write debugging information to
- Use a global Boolean flag to control when debugging information will be printed
- More structured than adding print statements
- Too many debugging statements can obscure code!

Integration Testing

- Unit testing does not guarantee modules will be error-free when used together
- **Integration Testing**: Testing that ensures modules will work when used together
- “Divide-and-conquer” approach, similar to design phase
- Two approaches: top-down and bottom-up

Top-Down Integration Testing

- Ensure the overall logical design works and interfaces between modules are correct
- Assumes low-level details are correct
- Low-level subprograms are replaced with **stubs** or placeholder modules
- Stubs may print debugging information and return values, but do not perform actual work

Bottom-Up Approach

- Uses a test driver to use individual modules together
- Starts with the lowest-level modules in the design and works up to the top-level program
- Useful when different authors wrote and tested separate modules

Testing C++ Data Structures

- **Goal:** Test the various operations of a data structure in an efficient manner
- **Solution:** Describe the test cases in a file, use a test driver to read and execute the test cases, writing the output to another file
- This allows for generalized testing
- Requires a test driver for each structure

Testing C++ Data Structures (cont.)

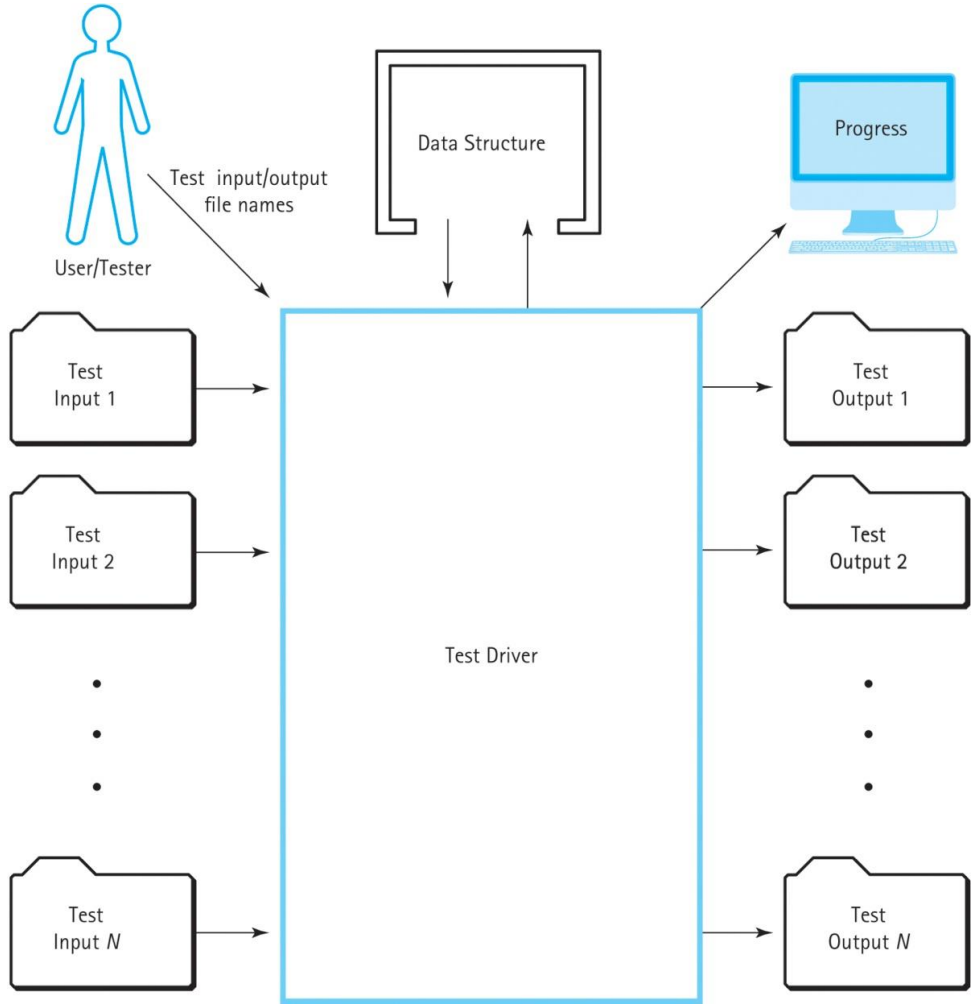


Figure 1.9 Model of test architecture

Testing C++ Data Structures (cont.)

- The input file lists commands for the test driver to execute
- Commands are specific operations for the structure, along with commands like “Quit”
- Input file also specifies the input for each command to use

Errors Rates in Software

Table 1.3 Error Rates upon Delivery by Application Domain[1]

Application Domain	Number of Projects	Error Range (Errors/ KESLOC[2])	Normative Error Rate (Errors/ KESLOC[2])	Notes
Automation	55	2 to 8	5	Factory automation
Banking	30	3 to 10	6	Loan processing, ATM
Command & Control	45	0.5 to 5	1	Command centers
Data Processing	35	2 to 14	8	DB-intensive systems
Environment/Tools	75	5 to 12	8	CASE, compilers, etc.
Military—All	125	0.2 to 3	< 1.0	See subcategories
• Airborne	40	0.2 to 1.3	0.5	Embedded sensors
• Ground	52	0.5 to 4	0.8	Combat center
• Missile	15	0.3 to 1.5	0.5	GNC system
• Space	18	0.2 to 0.8	0.4	Attitude control system
Scientific	35	0.9 to 5	2	Seismic processing
Telecommunications	50	3 to 12	6	Digital switches
Test	35	3 to 15	7	Test equipment devices
Trainers/Simulations	25	2 to 11	6	Virtual reality simulator
Web Business	65	4 to 18	11	Client/server sites
Other	25	2 to 15	7	All others

[1] Source: Donald J. Reifer, Industry Software Cost, Quality and Productivity Benchmarks, *STN* Vol. 7(2), 2004.
[2] KESLOC: thousands of equivalent source lines of code.

Table 1.3 Error Rates upon Delivery by Application Domain[1]

Practical Considerations

- How much verification is necessary?
- Must balance between time constraints, the effort required, and the impact of errors
- Verification should be started as early as possible
- Critical software must have lower error rates

The End!


A Software Engineer
Was Smoking ...
A Lady Standing Near
By Asked Him ,
Can't you See The
Warning?? Smoking is
injurious To health !
He Replied, "We are
bothered only
about Errors Not
Warnings !!
www.yohyoh.com