Part I: Problem Solving Techniques

孫雅麗
台灣大學 資管所
Problem-Solving Techniques

- Chapter 1: Principles of Programming and Software Engineering
- Chapter 2: Recursion: The Mirrors
- Chapter 5: Recursion as a Problem-Solving Technique
- Chapter 3: Data Abstraction: The Walls
- Chapter 4: Linked Lists
Motivation

• Complexities of large programs/systems
  - size
  - system complexity (platform, utility software, etc.)
  - software itself (logic, interaction and relationship, etc.)
Motivation

• Requirements
  - Modifiability
  - Readability
  - Fail-safe programming (example)
  - Debugging
Software Engineering

• A branch of “Computer Science”
• How to apply *principles, techniques* and *tools* to the production of programs (software)?
• Sometimes it is an *art*.
• We make every effort to make it a *science and engineering work*!
“Principles” of Programming and Software Engineering

• Characteristics of a good solution!

• Ways to achieve a “good” solution
  - abstraction (modeling)
  - modularity
  - information hiding
    • readability, modifiability
Chapter 1

Principles of Programming and Software Engineering
Outline

Problem Solving and Software Engineering

• What is a problem solving?
• The life cycle of software.
• What is a good solution?

Achieving a Modular Design

• Abstraction and information hiding
• Object-Oriented design
• Top-Down design
• General design guidelines
Outline (cont’d)

Key issues in programming

• Modularity
• Modifiability
• Ease of use
• Fail-safe
• Programming style
• Debugging
Problem Solving and Software Engineering

• When did you begin when you wrote your program?

• Problem Specification, Design
  - vs. Writing codes

• Large software development project
  - a team of “software engineers”
  - a team of “programmers”
System Development

• Architect!
• System architecture
  – System configuration
  – Functional block diagram
  – Control flow/data flow
• Implementation architecture
• Example
Problem Solving and Software Engineering (cont’d)

- Team work requires an overall plan, organization, and communication!

- Software Engineering
  - to apply scientific knowledge on planning, designing and construction and management of software
Software Life Cycle

• As the size and complexity of computer software development increase, some disciplines of software engineering is needed.
  - Principles
  - Techniques (methods)
  - Tools
Software Development Life Cycle - Nine phases
The Life Cycle of Software (1/5)

- **Stage#1: Specification**
  - Problem analysis
  - To determine **what the program is to do**
  - Functional specification; Input/Output/User Interface
Stage#1: Specification
(more details)

• Aspects of the problem which must be specified:
  – What is the input data?
  – What data is valid and what data is invalid?
  – Who will use the software, and what user interface should be used?
  – What error detection and error messages are desirable?
  – What assumptions are possible?
  – Are there special cases?
  – What is the form of the output?
  – What documentation is necessary?
  – What enhancements to the program are likely in the future?

• Prototype program
  – A program that simulates the behavior of portions of the desired software product
The Life Cycle of Software
(2/5)

• Stage#2: **Design**
  
  - To develop *a validated program (algorithms and data structures)* to *solve the problem*
Stage#2: Design
(more details)

- Includes:
  - Dividing the program into modules
  - Specifying the purpose of each module
  - Specifying the data flow among modules

- Modules
  - Self-contained units of code
  - Could be
    - Loosely coupled
    - Highly cohesive

- Interfaces
  - Communication mechanisms among modules

- Specification of a function
  - A contract between the function and the module that calls it
  - Should not commit the function to a particular way of performing its task
  - Include the function's preconditions and postconditions
The Life Cycle of Software (3/5)

• Stage#3: Risk Analysis
  - All software projects have risks
  - Techniques exist to **identify, assess and manage risks**
  - Will affect the other phases

• Stage#4: Verification
• Stage#5: Program Implementation
  - coding
Stage#4: Verification

(more details)

• **Formal methods** can be used to prove that an algorithm is correct

• **Assertion**
  – A statement about a particular condition at a certain point in an algorithm

• **Invariant**
  – A condition that is always true at a particular point in an algorithm
Stage #4: Verification
(more details) (cont’d)

• Loop invariant
  – A condition that is true before and after each execution of an algorithm’s loop
  – Can be used to detect errors before coding is started
• The invariant for a correct loop is true:
  – Initially, after any initialization steps, but before the loop begins execution
  – Before every iteration of the loop
  – After every iteration of the loop
  – After the loop terminates
The Life Cycle of Software (4/5)

• **Stage#6: Testing**
  - bottom-up testing
  - both valid and invalid data testing
  - *fail-safe*
  - *Functional test and Stress test*

• **Stage#7: Refining the Solution**
  - to relax assumptions, e.g., input/output format, more error checks, etc.
The Life Cycle of Software
(5/5)

• **Stage#8: Production**
  - to distribute and install software product to its intended users

• **Stage#9: Maintenance**
  - error correction
  - feature enhancement
  - *good documentation is important!*
Software Development Life Cycle - Nine phases

Documentation!
What Is a Good Solution?

• A solution is good if the total *cost* it incurs over *all* phases of its life cycle is *minimal*.

• Programs must be *well structured and documented*.

• *Efficiency* is only ONE aspect of a solution’s *cost*.
軟體產品、系統開發

一項全球性的調查顯示(2000)，在軟體專案的開發

- 85% 的專案是在時程落後
- 大幅超出預算（平均成本大幅超出預算的190%）
- 功能不全
- 甚至未完全檢測瑕疵及完成除錯的情形下，就上線使用

- 30%的專案將會被半途撤銷

軟體開發要能依照預定時程、合乎預算幾乎是件『不可能的任務』，甚至根本就注定失敗？

成功的專案需要避免那些失敗因素，才能成為『僥倖』成功的範例呢？
談軟體產品、系統開發專案
失敗的六大關鍵

• 「過份樂觀」和「沒有經驗的主事者」
• 低估軟體開發的複雜度及忽視許多不可抗力因素
• 來自各方的不合理壓力
• 不能以專業態度來 say『no 』
• 有多加人手來解決問題的心態
• 收尾無法作到『收斂』的功夫
Program Design: Achieving a Modular Design

- Top-Down design
- Object-Oriented design
- Abstraction and Information Hiding
- General design guidelines
Program Design

• Developing ALGORITHM ...
  - Translate **WHAT** is to be done to **HOW** it will be done

• Two approaches
  - *top-down* design (aka. stepwise refinement)
  - *bottom-up* design
Top-Down Design

• Initial statement -> more explicit statement -> more explicit ... until the resulting list of statements constitutes an algorithm.

• Basic assumptions:
  - clear and frozen functional specifications
  - no or minimum reusability
Hierarchy of Modules: an example of Top-Down Design

Figure 1.4
A structure chart showing the hierarchy of modules
Bottom-up Design

- **Reusability** is the goal.
- A major feature/advantage in Object-Oriented design
- To build up algorithms from what has already been developed
- To add new algorithms to the algorithm library
Bottom-up Design (cont’d)

- e.g., class library in object oriented programming languages
Modeling Object-Oriented Design Using UML

- The Unified Modeling Language (UML) is a *modeling language* used to express object-oriented designs
  - *Class diagrams* specify the name of the class, the *data members* of the class, and the *operations*
Modeling Object-Oriented Design Using UML

- Relationships between classes can be shown by connecting classes with lines
  - Inheritance is shown with a line and an open triangle to the parent class
  - Containment is shown with an arrow to the containing class
UML Diagrams

Figure 1.6

UML diagram for a banking system
Unified Modeling Language (UML)

- It is a standardized general-purpose modeling language in software engineering
- It includes a set of **graphical notation** techniques to visualize, specify, construct, and document the artifacts of a software-intensive system.
UML: a graphical language

- A standard way to write a system's blueprints, including conceptual things, e.g.,
  - business processes
  - system functions
  - programming language statements,
  - database schemas, and
  - reusable software components.
Important Concepts ...

• **Data Encapsulation** or **Information Hiding** is the concealing of the implementation details of a data object from the outside world.

• **Data Abstraction** is the separation between the specification of a data object and its implementation.
Example: Ice maker in a refrigerator
A Summary of Key Issues in Programming (1/7)

1. Modularity has a favorable impact on
   - Constructing programs
   - Debugging programs
   - Reading programs
   - Modifying programs
   - Eliminating redundant code
A Summary of Key Issues in Programming (2/7)

- Modifiability is possible through the use of
  - Function
  - Named constants
  - The `typedef` statement
A Summary of Key Issues in Programming (3/7)

2. Ease of use

- In an *interactive* environment, the program should **prompt** the user for input in a clear manner
- A program should always **echo** its **input**
- The **output** should be well labeled and easy to read
A Summary of Key Issues in Programming (4/7)

3. Fail-Safe Programming

- Fail-safe programs will perform reasonably no matter how anyone uses it
- Test for invalid input data and program logic errors
- Handle errors through exception handling
A Summary of Key Issues in Programming (5/7)

4. Eight issues of personal style in programming
   – Extensive use of functions
   – Use of private data fields
   – Avoidance of global variables in functions
   – Proper use of reference arguments
A Summary of Key Issues in Programming (6/7)

• Eight issues of personal style in programming (Continued)
  – Proper use of functions
  – Error handling
  – Readability
  – Documentation
Readability

• For a program to be easy to follow, it should have
  - A good structure and design
  - A good choice of identifiers
  - Good indentation and use of blank lines
  - Good documentation
  - Should avoid clever programming tricks that save a little computing time at the expense of much human time
Naming Conventions

- **Function names** within a class are **verbs**, with the first letter lowercase and subsequent internal words capitalized, e.g., `solveTowers`
- **Variables** begin with a lowercase letter, with subsequent internal words capitalized, e.g., `newPointer`
- **Data types** declared in a `typedef` statement and names of structures and enumerations each begin with an uppercase letter, e.g., `ListNode`
- **Named constants and enumerators** are entirely uppercase and use underscores to separate words.
A Summary of Key Issues in Programming (7/7)

• 5. Debugging
  – Must learn how to systematically track down errors
  – Fail-safe techniques help to guard against certain errors and report them when occur, also a great aid in debugging
  – Programmer must systematically check a program’s logic to determine where an error occurs
  – Tools to use while debugging:
    • Watches
    • Breakpoints
    • cout statements (e.g., variables)
    • Dump functions

The real trick is to use these aids in an effective way!
Difficulties in debugging

• The persons who code the program seem to be totally baffled by errors in their programs and have no idea how to proceed.

• Programmers tend to have a desire to “believe” that their program is really doing what it is supposed to do.
Summary (1/3)

• Software engineering: techniques to facilitate development of programs
• Software life cycle consists of nine phases
• Loop invariant: property that is true before and after each iteration of a loop
• Evaluating the quality of a solution must consideration a variety of factors
Summary (2/3)

- A combination of object-oriented and top-down design techniques leads to a modular solution
- The final solution should be as easy to modify as possible
- A function should be as independent as possible and perform one well-defined task
Summary (3/3)

- Functions should include a comment: purpose, precondition, and postcondition
- A program should be as fail-safe as possible
- Effective use of available diagnostic aids is one of the keys to debugging
- Use “dump functions” to help to examine and debug the contents of data structures
The end. 😊
Homework

• Pages 61-63 #2, #5, #11, #12
• Page 63 #4
Denial Of Service (DOS) Attacks - Smurf

- Internet Control Message Protocol (ICMP) Echo/Reply

- Spoofing Source address
DOS Attacks - Smurf

- Sending a large amount of ICMP echo traffic to a set of IP broadcast address with a specified spoofing source address.
DOS Attacks-SYN Flood

- Sending a barrage of initial SYN's without sending the corresponding ACK's, essentially leaving the server waiting for non-existent ACK's.

all available operating system resources are consumed, and the computer can no longer process legitimate user requests.
Call Scenario

1. Direct PC - to - PC

2. Routed by Call Server

1. Direct PC - to - PC

2.1 QoS AP Router

2.2 Internet

Call Server

SIP Softphone

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
Functional/Implementation Architecture

- Application Manager
- MIB
- SIP Manager
- SIP Session Table
- Codec Table
- SIP FSM
- Extract SIP Info
- RTP Rules & Aging Mgmt
- Layer 3/4 Packet Classifier
- QoS Manager

Control flow: ...
Data flow: →
Interface:  
Module:  

Diagram elements:
- Modules represented by rectangles
- Interfaces represented by circles
- Flow directions indicated by arrows
Module Relation

1. parse call_ID

Packet Classifier → SIP Manager

SIP FSM

1. update state

SIP Session Table

2. get current state

3. get next state

4. process packet

SIP Session Table

call-ID not in table → call-ID in table

NONINVITE → INVITE

call-ID not in table

call-ID in table

Not follow FSM

follow FSM

retransmission

Not retransmission

1. release

2. log

3. reserve

Bye

ACK

others

Extract SIP Info

Codec Table

QoS Manager

MIB

Implementation Architecture
E-Home: Wireless Multimedia Communication

- Up to three MPEG-4 (6Mbps) streams for 802.11g
- Support unicast and multicast