6. Problem solving and programming

1. Specification and object-oriented design
2. Algorithm design tools
3. The loop control structure
4. Event-driven software and JavaScript

Inquiry

- Why is programming considered really hard?
- Is programming addictive?
- Why does software not work?
- What steps and what tools enable building computer solutions efficiently?
6. Problem solving and programming

Software pioneers

Ada Lovelace wrote the first computer program

Grace Hopper invented the COBOL programming language

Topic objective

Explain and apply concepts of computing-system specification and design
Essential and priority objectives

6.0a Recall basic design concepts* $^M$
6.1 Describe the steps in system development*
6.2a Explain algorithm design tools*
6.2b Trace a branching computation*
6.3a Trace a looping computation**

1. Specification and object-oriented design

- What is a system?
- Do you use any systems?
- How do you fix them when they don’t work well?
- What is analysis?
- Do you design any systems?
6. Problem solving and programming

Subtopic objective

6.1 Describe the steps in system development*

Systems and system development

- **System**: a group of components working together toward a result
- **Information system**: hardware, software, staff, procedures, ways to generate and store data
- A **systems development life cycle** describes steps in analysis, design, implementation, testing, and maintenance
- The job categories of **systems analyst** and **software engineer** carry out these tasks
The system development process

The development process is iterative.

Phases:
- **Analysis:** specifies desired input and output
- **Program design:** prepares algorithms, plans
- **Coding:** implements design by writing a program in a language
- **Testing:** evaluates working program
- **Maintenance:** addresses errors and needs not found in previous stages

An example problem description

- **What service must the software provide?**
  *Example:* Process customer orders from catalog
- **What assumptions are made?**
  *Example:* Some customers may find Web access convenient
- **What risks are involved?**
  *Example:* Some users are inexperienced with Web access
### Aspects of system specifications

- Input
- Output
- Relationship between input and output

*Example*: Sum a column of numbers

*Designs tell how to satisfy the specifications:*

- Definiteness (deterministic sequence of operations)
- Finiteness and reasonableness of time

---

### Systems design

*Design of solutions is:*

- based on specification (analysis)
- language independent
- structured
- modular

*Tools*: flowcharts, pseudocode, module hierarchies, Unified Modeling Language diagrams

- Coding and testing come after design
System implementation and maintenance

- Purchase and install hardware, software
- Develop and document applications
- Test applications
  - Unit testing (of components)
  - Integration testing (of entire system)
- Train users and convert to new system
- Maintenance includes repairs and upgrades

Unified Modeling Language

- A standard graphical notation for system specification and design
- Motivated by need to depict interactions between systems and their environments, initiated by external actors like users
- Diagrams include use-case, activity, class, state, interaction
- Supports an object-oriented methodology
6. Problem solving and programming

Use cases and system specification

- **Use case**: A typical interaction between system and an *actor* in its environment
- **Actor**: A role, such as customer, manager, supplier, salesperson
- Actors initiate use cases
- Example of two uses cases (UML use-case diagram, right)

UML activity diagrams

- Depict order in which steps occur in a use case
- **Examples**:
  - A decision point (branching)
  - A *fork* (2 forms display at once)
  - A *join* (execution converges)

Schneider and Winters, 1996, p. 59
The object-oriented paradigm

- *Object:* An instance of a class, with attributes
- *Class:* A category defined by data attributes (properties) and operations (“methods”)
- In object-oriented programming, objects interact with each other via *messages*
- Objects may *contain* other objects
- Subclasses may *inherit* from other classes
- *Example:* Right-clicking an icon displays the properties and methods of the class of the object the icon represents

UML class diagrams

- Describe the compositions of objects
- *Form:* rectangle with 3 horizontal compartments

- *UML class association diagrams* show how instances of different classes are related
2. Algorithm design tools

- What processes are part of your activities?
- Do all processes take the same steps every time?
- Are some processes repetitive?
- Do some processes terminate? Is this useful?
- How do you add two three-digit numbers?

Subtopic objectives

6.2a Explain algorithm design tools*
6.2b Trace a branching computation*
6.2c Design a branching computation
Algorithm:  
A precise plan to convert input to output in a finite number of steps

- Program designs use algorithms
- Much computation is algorithmic
- Flowcharts and pseudocode can express algorithms

Algorithmic thinking

Essential properties of algorithms:
- Specification of input, output, and relationship between them
- Definiteness: deterministic sequence of operations
- Effectiveness: problem is actually solved
- Finiteness: all algorithms terminate
Expressions an algorithm or process

- Natural language
- Language of mathematics
- Design notations
  - Flowchart
  - Pseudocode
  - Unified Modeling Language (UML)
- Programming language (JavaScript, Java, C++…)
- Machine and assembler languages

Two notations for low-level design

- Both notations show order of execution

- **Pseudocode**
  - informal
  - precise
  - text outline format

- **Flowcharts**
  - graphical
  - shapes denote steps
  - arrows show flow of control

Equivalence of values:

- \( a = b^2 + 5 \)
- Assignment:
  \[ y \leftarrow x_1 + x_2 \]
Control structures

All algorithms may be built from three basic control structures:

**Sequence**
- Cook dinner
- Eat dinner
- Clean up

**Branch**
- Begin
- Exercises are confusing?
  - N: Review chapter
  - Y: Continue
- End

**Loop**
- Begin
- Dial
- Got answer?
  - T: Talk
  - F: Restart
- End

The *sequence* control structure

**Problem:**
Accomplish dinner routine

**Pseudocode:**
1. Cook dinner
2. Eat dinner
3. Clean up

*Algorithms go step by step*
An algorithm to add numbers

1. Prompt for integers \( \text{input1, input2} \)
2. \( \text{sum} \leftarrow \text{input1} + \text{input2} \)
3. Display \( \text{sum} \)

- In pseudocode and flowcharts, the symbol \( \leftarrow \) stands for assignment of a value to a variable
- An algorithm is almost always *general purpose*: it works with *variables*, not only specific constant values

Example problems

- Performing binary operations (+, -, *, /)
- Evaluating algebraic expressions \((mx + b; \ ax^2 + bx + c; \ a^b)\)
- Operating on a sequence:
  - Is it in ascending order?
  - Are all values the same?
  - What is *mode* (vote winner)?
  - Does sequence contain a certain value?
The decision control structure

Problem:
Prepare to do homework

Pseudocode:
If exercises are confusing
review chapter

Flowchart

IF in spreadsheets

Excel has an IF function that yields a value conditional on a cell value
= if(c4 > 0, "yes", "no")
yields “yes” as cell value if cell c4 has a value greater than 0
Logical operations in spreadsheets

- **IF** (condition, value-if-true, value-if-false) returns value-if-true if condition is true, otherwise returns value-if-false
- **AND** (condition1, condition2) returns true if both conditions hold
- **OR** (condition1, condition2) returns true if either or both conditions hold

<table>
<thead>
<tr>
<th>Salesperson</th>
<th>Sales</th>
<th>Returns</th>
<th>Great?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>40000</td>
<td>5000</td>
<td>TRUE=AND(B3&gt;30000, C3&lt;10000)</td>
</tr>
<tr>
<td>Smith</td>
<td>40000</td>
<td>12000</td>
<td>FALSE</td>
</tr>
<tr>
<td>Doe</td>
<td>30000</td>
<td>5000</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

Finding absolute value

**Input n**
- If \( n \geq 0 \) display \( n \)
- otherwise display \( -n \)

- With the *branch* control structure, one and only one of the alternatives executes
- In pseudocode, the subordinate (conditional) steps are normally indented
Finding largest of three numbers

• Right, the values $a$, $b$, and $c$ are input
• The variable $y$ stores the largest input found so far
• Two comparisons are used to assign to $y$ the value of the largest of the three.

Debugging a flawed design

• Suppose we try to find the largest of three numbers as follows:
  
  ```plaintext
  input a, b, c
  y ← a
  if b > a
    y ← b
  if c > a
    y ← c
  ```

  Trace of this algorithm for $(a, b, c) = (2, 4, 3)$ is above; do you see the error?
Why trace?

- Computer programs don’t display all their internal workings
- To find and fix a car problem, the mechanic must look under the hood
- A trace displays the values of all variables as they change
- Tracing is crucial in debugging programs and systems

Algorithms, flowcharts, and pseudocode

- Every algorithm can be represented by a flowchart or pseudocode
- Some flowcharts and pseudocode represent algorithms
  - All input is at the start
  - All output is at the end
  - Always terminate
- There is no way to tell for sure whether a given flowchart or pseudocode depicts an algorithm
6. Problem solving and programming

Modular design

- A problem-solving strategy: *divide and conquer*
- All languages use modularity
- A *hierarchical* organization may be modular

Module hierarchy diagrams

- *Example:*
  *Main* invokes *Input, Calculate, and Output; Calculate* calls *Assign and Multiply*
  
  ```
  Main
  A. Input
  B. Calculate
     1. Assign
     2. Multiply
  C. Output
  ```

  - A module hierarchy chart shows module dependencies, whereas a flowchart shows order of execution.
3. The loop control structure

- What processes can the flowcharts we’ve seen be used to design?
- Is there any process they can’t represent?
- How do we know *when* a program has a bug?
- How do we know *why*?

Subtopic objectives

6.3a Trace a looping computation**
6.3b Design a looping computation
6.3c Explain the concept of debugging
6. Problem solving and programming

Problem: Telephone someone

**Pseudocode:**
Repeat
  Dial number until someone answers

**Flowchart:**

Can this be simplified?

input $x$
y $\leftarrow$ 0
$i$ $\leftarrow$ 0
$y$ $\leftarrow$ $y + x$
$i$ $\leftarrow$ $i + 1$
$y$ $\leftarrow$ $y + x$
$i$ $\leftarrow$ $i + 1$
$y$ $\leftarrow$ $y + x$
$i$ $\leftarrow$ $i + 1$
$y$ $\leftarrow$ $y + 2$

**Hint:** It computes
$y$ $\leftarrow$ $3x + 2$
Counting to 10

A counted loop; counter is \( i \)

- \( i \leftarrow 1 \)
- while \( i \leq 10 \)
  - display \( i \)
  - \( i \leftarrow i + 1 \)

A sentinel-controlled loop

Not an algorithm, because input alternates with processing

- Begin
- \( \text{total} \leftarrow 0 \)
- Input \text{quantity}
- \( \text{total} \leftarrow \text{total} + \text{quantity} \)
- \( \text{quantity} \leftrightarrow 0 \)
- Display \text{total}
- End
Tracing an algorithm or process

- Allows designer to check result of algorithm, including internal (undisplayed) values
- Use one column per value traced; one row per loop iteration.
- Example (See previous slide), assuming input are 3, 2, 1, 0:

<table>
<thead>
<tr>
<th>quantity</th>
<th>total</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Algorithm without input

Trace:

<table>
<thead>
<tr>
<th>qty</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>
Finding largest of \( n \) numbers

- A loop finds the maximum.
- Here, the value \( y \) is compared with each element of the input series: \( x_1, x_2, x_n \).
- Any values that exceed \( y \) are assigned to \( y \) as its new value.

Suppose input contains an arbitrary number of values, \( n \).

Multiplication without “\( \times \)”

Problem: Find the bug in this design
Combining control structures

**Branch within loop**

- **Problem:** Is a vote unanimous?
- **Solution:**
  
  ```
  input num_votes, vote
  is_unan ← true, prev ← vote
  for i ← 1 to num_votes do
    if prev ≠ vote
      is_unan ← false
      prev ← vote
    input vote
  if is_unan
    display “unanimous”
  else display “not unanimous”
  ```
6. Problem solving and programming

Debugging in problem solving

• IT often displays *faults* where system fails to produce desired results

• *Debugging*: “figuring out why a process or system doesn’t work properly” (L. Snyder, 2006) – and fixing the error!

• Three causes of error:
  – Faulty input *data* (e.g., typo)
  – Faulty *command* (e.g., user misunderstands command syntax)
  – Faulty *system* (e.g., program bug)

Debugging skills

Debugging data, commands, or system requires analytical and logical thinking and precision

– Is error reproducible?
– Localize the problem;
  e.g., to one device
– Avoid just “trying something”
6. Problem solving and programming

Debugging using trace

• **Problem:** Why does the pseudocode below fail to add the numbers from 1 to 5?

\[\begin{align*}
\text{count} & \leftarrow 0 \\
\text{sum} & \leftarrow 0 \\
\text{while} \; \text{count} < 5 \\
& \quad \text{input} \; x \\
& \quad \text{count} \leftarrow \text{count} + 1 \\
& \quad \text{sum} \leftarrow \text{sum} + 1 \\
\text{display} \; \text{sum}
\end{align*}\]

• **Solution:** Trace the algorithm, see where an unexpected value occurs

Modular decomposition: case study

Separate modules are easier to understand.
4. Event-driven software and JavaScript

- Who has written a program?
- What can happen when you visit a web page?
- Can HTML support user interaction?

Subtopic objectives

6.4a Describe features of JavaScript
6.4b Write and test JavaScript code†
6.4c Create an event-driven web page†
Event-driven design

• Browsers and most other apps are interactive, alternating input and output
• Command-line environments: URL line in browser, Google prompt, DOS or UNIX prompt
• Features of graphical user interfaces: windows, icons, menus, dialog boxes, buttons
• Common feature: User generates events, e.g., clicks, drags, keystrokes, timeouts
• Browser interacts via hyperlinks; embedding of event-handling JavaScript in HTML files

JavaScript: A procedural language

• The JavaScript text between the HTML tags <script language = "Javascript"> and </script> will execute when browser displays HTML file
• Motivation:
  • Working with IT means thinking abstractly and concretely about data and operations
  • Design, coding, and testing of solutions are part of learning problem solving
Generations of programming languages

- First generation: Machine languages (binary)
- 2nd generation: Assembler languages, processor specific
- 3rd generation: Procedural, high-level, hardware independent (C, BASIC, JavaScript)
- 4th generation: Nonprocedural query or report-generation languages (SQL, RPG)
- Declarative languages (Prolog)
- Object-oriented languages (C++, Java)
- Functional languages (Lisp)

Problem specifications and user interfaces

- Designer must consider *assumptions* about
  - Problem domain (e.g., business, education, personal, healthcare)
  - User needs and expectations
- *Interface* refers to how application (e.g., at web site) appears and responds to user
- Most user interfaces today are *graphical*
- Implementation (coding) is partly independent of interface
### JavaScript example

```html
<html> <!-- hello.htm -->
<head><title>DK-Hello</title></head>
<body>63.120 says Hello!
    <script language="JavaScript">
        alert("hello");
    </script>
</body>
</html>
```

- Displays “hello” in an alert box (a kind of dialog)
- `alert` is a JavaScript procedure
- JavaScript may be used after `script` tag

### Button

```html
<html> <!--button.htm-->
<head><title>63120 Hello</title></head>
<body>
    <input type=button value = "Hello"
    onClick = 'alert("Hi")'>
</body></html>
```

- This code displays “Hi” when “Hello” button is pressed
- `<input>` tag defines an input button object
- *Event handler*: code that specifies application’s response to a particular event, such as user click on a button
6. Problem solving and programming

## Counting button clicks

```html
<html>  
<head><title>yes-no counter</title></head>  
<body>  

/* count-yes.htm  Displays Yes, No buttons for user to click, counts # clicks on each. Event-handlers specify response to input events: Yes, No, Stats, Reset. Variable track yeses and nos. */
<script language="JavaScript">
var num_yes=0, num_no=0; // Variables
</script>

<table>
<thead>
<tr>
<th>Button</th>
<th>JavaScript Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td><code>num_yes = num_yes + 1</code></td>
</tr>
<tr>
<td>No</td>
<td><code>num_no = num_no + 1</code></td>
</tr>
<tr>
<td>Stats</td>
<td><code>alert(&quot;Yes: &quot; + num_yes + &quot; No: &quot; + num_no)</code></td>
</tr>
<tr>
<td>Reset</td>
<td><code>num_yes = num_no = 0</code></td>
</tr>
</tbody>
</table>

</body> </html>
```

## Text input/output

```html
<head><title>Input echo</title></head>  
<body>  
<form name="Input"><table>
<!-- Display prompt and get input: -->
<td>Enter your user name: </td><input type=text name=user value="" size = 15>

<!-- At button-press, display message: -->
<td><input type=button value="Done" onclick="alert("Hello " + user.value)"></td>

<!-- Assigning a value to onClick defines JavaScript response to button click -->
</form></body>
```
JavaScript statements

- Statements are executable (or are variable declarations)
- Statements include
  - assignment \( a = 4; \)
  - compound statements \{ in braces \}
  - if, if ... else
  - while
- Simple statements, e.g., assignment, terminate with “;”

JavaScript variables

- Variables have name, type, value
- Must declare a variable to use it
  \[ \text{var } a; \]
- Valid types (interpretation of bits):
  - Number: \( \text{var } a=2, b=3; \)
  - String: \( \text{var } name = "Bill"; \)
  - Truth value: \( \text{var } greater = (a > b); \)
- Assignment can change variable’s value:
  \( a = b \times 4; \)
Expressions

• *Expressions* have *values* and are used in statements
• Expressions are *literals*, *variables*, or *function calls*, or may be formed with *arithmetic operators* (+, -, *, /, %)
• *Boolean* expressions have truth values and may be built with *relational operators* (==, !=, <, >, <=, >=) or *logical operators* (!, &&, ||)

Branches and loops in JavaScript

• *Branch:*
  ```javascript
  if (a > b)
    max = a;
  else
    max = b;
  ```

• *Loop:*
  ```javascript
  i = 1;
  while(i < 10)
  {
    sum = sum + i;
    i = i + 1;
  }
  ```

  See example code from D. Keil, C. Breuning
Testing and debugging

- Software and web sites require testing before deployment
- Testing is often done by quality assurance departments
- All software writing entails error and debugging
- JavaScript is easy to test on a browser, but the browser does not supply error locations or other diagnostics

Example with branch control structure

```html
/* remember.htm Displays 2 buttons for user to click, tells which was clicked last */
<var most_recent="A", message="Last you clicked was ">
<script>
<td><input type=button value = "A"
onClick = 'most_recent="A"'></td>
<td><input type=button value = "B"
onClick = 'most_recent="B"'></td>
<td><input type=button value = "Check"
onClick = 'if (most_recent == "A")
alert(message+"A");
else alert(message+"B")'>
</td>
</script>
[remember.htm]
```
Example with loop

```html
<! power.htm Display prompt and get input:>
<td>Enter a base:  <!-- Generate input-box:-->
<input type=text name=x1 value="0" size = 8>  </td>
<td>Enter an exponent:  <!-- Generate input-box:-->
<input type=text name=x2 value="0" size = 8></td>
<td><input type=button  value="Done" onClick = '  
    a = i = parseInt(x2.value);
    b = parseInt(x1.value)
    y = 1;
    while(i > 0) {
        y = y * b;
        i = i - 1;
    }
    alert(a + "^" + b + " = " + y); '>  
```

Modularity and subprograms

- *Divide and conquer*: a problem solving strategy that breaks problems down into subproblems, each with a simpler solution
- In this scenario, *subprograms* ("functions" in JavaScript terms) solve subproblems
- JavaScript functions, once *defined* (spelled out), may be *called* (invoked)
- Interfaces between subprogram and rest of program: *parameters; return values*
6. Problem solving and programming

References


