1. Specification and design

1. Object-oriented specification and design
2. Algorithm-design tools
3. The loop control structure

Inquiry

• What is a *problem*?
• How do *you* solve problems in your life?
• What does *specification* mean?
• What is a *design*?
• What *flowcharting* have you done?
• What are some steps in problem solving?
1. Describe and apply principles of system specification and design

- Identify the steps in software development*
- Explain the notion of an algorithm**
- Describe algorithm design tools*
- Trace a branching computation**
- Design a branching computation**
- Trace a loop**
- Design a looping computation**
- Explain the concept of debugging**
1. Specification and design

Subtopic outcomes

1.1a Write a UML class diagram*
1.1b Apply standards for system specification*
1.1c Explain principles of object-oriented design*
1.1d Design and test a two-threaded robotic process†
1.3a Describe the loop control structure
1.3b Describe the debugging process*
1.4 Evaluate an expression using a recurrence

The system development process

- Development process is iterative
- **Specification**
  - Understanding the problem
  - input; output; their relationship
  - Unified Modeling Language
- **Design**
  - Language-independence
  - Module hierarchies, flowcharts, pseudocode
  - Coding, testing come later

![Diagram of the system development process](image_url)
1. Specification and design

**Four CS I skills objectives**

- Vague specification $\rightarrow$ precise specification (analysis)
- Problem specification $\rightarrow$ flowchart (design)
- Flowchart $\rightarrow$ trace (design verification)
- Design $\rightarrow$ program (coding)
- Program $\rightarrow$ trace (testing)

1. Object-oriented specification and design

- What is an *object* in Windows, the Mac OS, or Linux?
- What does a database *field* represent? A *row*? A *table*?
- How may concepts be related to each other?
Standards for system specification

- Specification (requirements document) gives:
  - input
  - output
  - correspondence between the two
- User interface should be specified
- If input is via *file* or *port*, specify this
- Any repeated steps should be stated

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
Computations may be interactive

- Typical menu- or command-driven user environment:
  ```java
  Repeat
  input a command
  execute command
  until command = "quit"
  ```
- With interaction, input may depend on previous output
- Almost all computing today is interactive
- Internet computing presents new challenges related to interaction

Designing for interaction

- Most applications must respond to multiple inputs from a user in sequence, must store information reflecting past interactions
- A design stage follows the analysis (specification) stage
- Design precedes coding (e.g., HTML, JavaScript, Java, C++)
Unified Modeling Language

• A standard graphical notation for system specification and design
• UML (1990s) was motivated by need to depict interactions, initiated by external actors, between systems and their environments
• Diagrams include use-case, activity, class, state, interaction
• UML supports an object-oriented methodology

Use cases

• Use case: A typical interaction initiated by actor in its system environment
• Actor: A role, such as customer, manager, supplier, salesperson
• Example of two use cases where customer is actor (UML use-case diagram, right)
IBM user interface principles

- **Affinity**: good visual design
- **Assistance**: provide proactive assistance
- **Availability**: all objects at any time
- **Encouragement**: predictable and reversible
- **Familiarity**: build on user’s prior knowledge
- **Obviousness**: objects visible and intuitive
- **Personalization**: user customization of interface
- **Safety**: keep the user out of trouble
- **Satisfaction**: user feeling of achievement
- **Simplicity**: do not compromise usability
- **Support**: place the user in control
- **Versatility**: Support alternate techniques.

Features of *graphical user environments* (e.g., Windows)

- Program loading (Start menu; icons, Run)
- Task switching (task bar)
- Device control (Start / Settings / Control panels)
- Disk directory tree structure
  - program files
  - data files
  - folders
- Menus, icons, dialog boxes
- Drag and drop
- Double-click to execute commands
Objects in a graphical user interface

- A window is an object with position, size, contents
- A window opens, closes, moves, resizes
- Other objects: icons, menus, buttons
- In Windows, clicking on an object with right mouse button gives user access to object’s properties and methods (operations)

Event-driven applications

- Response to an event is an algorithm that generates from input and current state
  - an output and
  - a change of state
Object-oriented design

- Object-oriented design focuses on classes of objects as defined by their attributes (properties) and behaviors (methods)
- **Object**: An instance of a class
- In object-oriented programming, objects interact with each other via *messages*
- **Example**: Right-clicking an icon displays the properties and methods of the class of the object the icon represents

UML class diagrams

- Rectangle with 3 horizontal compartments

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>ID</td>
</tr>
<tr>
<td>salary</td>
</tr>
<tr>
<td>display()</td>
</tr>
<tr>
<td>input()</td>
</tr>
<tr>
<td>calc-paycheck()</td>
</tr>
</tbody>
</table>

- Class association diagram

`Roster` \(\rightarrow\) `Employee`

Association of *containment*
1. Specification and design

**Relationships between classes**

- Objects may *contain* other objects
- Subclasses may *inherit* from other classes

![Diagram showing relationships between classes](image)

**UML activity diagrams**

- Depict order in which steps occur
- *Example*: A decision point (branching)

![Activity diagram example](image)

*Schneider and Winters, 1996, p. 59*
**UML state diagrams**

- Depict a process that transitions from state to state; e.g., academic courses

```
Enrollment

Proposed → Open For Enrollment → Full → Closed to Enrollment

Scheduled

Closed

cancelled

student dropped [seminar size > 0]

student dropped [seminar size = 0]

Final Exams

Being Taught

term started

classes and

Closed

[Diagram showing various states and transitions related to enrollment process]
```

**Concurrent activity: fork and join**

- A *fork* (two forms display at once)
- A *join*, where execution converges

```
Log in

Order form displayed

Order selection form displayed

Log off

Order form closed

Order selection form closed
```

[Diagram showing concurrent activities with fork and join, including log in, order form display, order selection form display, log off, and order form and order selection form closures]
Concurrenty in robotics

- *Waiting for a spoken command* and responding is an event-driven process
- *Speaking and walking* at the same time is a case of concurrency
- [pic of NAO Choreograph screen]

Subtopic outcomes

1.1a Write a UML class diagram*
1.1b Apply standards for system specification*
1.1c Explain principles of object-oriented design*
1.1d Design and test a two-threaded robotic process†
2. Algorithm-design tools

- What happens in design in everyday life?
- What is a flowchart?
- What is pseudocode?
- Have you traced an algorithm?
- Have you designed an algorithm?

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
Design concerns

- Simplicity (via abstraction, structure)
- Performance (throughput, response time)
- Reliability (redundancy, recovery, integrity)
- Evolvability (adaptation to changes in function and scale)
- Security (access control privacy, authentication)

**Design principles**: abstraction, information hiding, modularity, packages, version control, divide and conquer, layering, hierarchy, reuse, interfaces, encapsulation, virtual machines

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
Expressing 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

Begin
Cook dinner
Eat dinner
Clean up
End

Branch

Begin
Exercises are confusing?
Review chapter
End

Loop

Dial
Got answer?
Talk

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
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

Counting to 10

\[ i \leftarrow 1 \]
while \( i \leq 10 \)
  display \( i \)
  \[ i \leftarrow i + 1 \]

- A counted loop; counter is \( i \)
Multiplication without “×”

Problem: Find the bug in this design

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 a flawed design

- Suppose we try to find the largest of three numbers as follows:

  \[
  \text{input } a, b, c \\
  y \leftarrow a \\
  \text{if } b > a \\
  \quad y \leftarrow b \\
  \text{if } c > a \\
  \quad y \leftarrow c \\
  \]

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

### Tracing an algorithm or process

- Allows designer to check result of algorithm, including internal (undisplayed) values

- Use one column per variable; one row per iteration.

- *Example* (See prev slide), assuming input 3, 2, 1, 0:

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

D. Keil 8/13
## Algorithm without input

```
Begin
qty ← 2

If qty > 9 then
    qty
Else
    Add 3 to qty

End
```

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>

---

## 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
Debugging using trace

- **Problem**: Why does the pseudocode below fail to add the numbers from 1 to 5?
  
  ```
  count ← 0
  sum ← 0
  while count < 5
      input x
      count ← count + 1
      sum ← sum + 1
  display sum
  ```

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

---

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.
The loop control structure

- Loops enable repetitive processes
- A loop has a body and an exit condition

**Problem:**
Telephoning someone

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

**Structured flowcharts**

- Have one entrance, one exit to each step

- Loops in structured pseudocode observe this condition
Three control structures suffice

• Three control structures: *sequence*, *branch*, and *loop* are used in structured programming; may be combined

• *Any solvable problem* can be solved using only these three

• *Note*: Some problems are not solvable

The story at a glance

• A flowchart should show what happens, at a glance

Not too clear

More readable
**Non-algorithmic computations**

- The process diagrammed at right will never terminate if user continues to input a non-zero value

- *Interaction*, alternating input and output, is non-algorithmic

---

**Modular decomposition**

- *One strategy*: divide and conquer
- All programming languages support modularity
- An *organization* is modular
- Modular design may be *top-down*
- *Subprograms* implement modular designs
1. Specification and design

Module hierarchy charts

- Tell which subprograms use which
- Module do_problem here uses modules calculate, get_input and display_result; calculate uses add and divide
- Stepwise refinement further breaks down problem
- Module hierarchy chart shows calling relationships rather than time sequence

3. The loop control structure

- Do you know any algorithms?
- How is a series of numbers added?
- Can all problems be solved with branches?
- For what problems would the branch control structure alone be inadequate?
Can this be simplified?

Can this be simplified?

```plaintext
input x
y ← 0
i ← 0

y ← y + x
i ← i + 1

y ← y + x
i ← i + 1

y ← y + x
i ← i + 1

y ← y + 2
```

**Hint:** It computes

\[ y ← 3x + 2 \]

Problem: Display all values from 1 to 10

- A *counted* loop; counter is \( i \)
A sentinel-controlled loop

**Sum**\((a, b)\)

\[
y \leftarrow a \\
\text{while } b > 0 \\
a \leftarrow \text{succ}(a) \\
b \leftarrow b - 1
\]

**Problem: binary to decimal**

- **Problem**: Convert binary string to decimal value
- **Analysis**: A numeral represents a sum of place values, which are each \(b^i\), where \(b\) is base (2, 10,...) and \(i\) is the number of places from right
- **Solution** (see next slide):
  - Let \(x\) be the input bit sequence, \(y\) be output
  - Start at rightmost bit (counter \(i = \# \text{ bits}\))
  - Loop through bits, R to L
  - At each 1-bit, add a power of 2 (place-value)
  - At each iteration of loop, double place-values, decrement \(i\)
**1. Specification and design**

---

**Binary-to-decimal algorithm**

*Note* $x_i$ is the $i$th bit of string $x$.

**Example**: If $x = '10'$ and $i = 2$, then $x_i = '0'$.  

---

**Tracing a flowchart**

- Allows designer to check progress of algorithm or process, including internal (undisplayed) values.
- Use one column per value traced; one row per loop iteration.
- **Example** (See previous slide):

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

---

D. Keil 8/13
Debugging using trace

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

  ```plaintext
  count ← 0
  sum ← 0
  while count < 5
    count ← count + 1
    sum ← sum + 1
  display sum
  ```

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

Loop invariants

- A loop invariant is a logical (true/false) assertion about values and outputs at the beginning of a loop iteration

- **Example:** A loop invariant for the previous slide is that all values in \{1 \ldots i – 1\} have been displayed

- Loop invariants help to show *correctness* of algorithms
1. Specification and design

Nested control structures

Sequences (arrays, strings)
- A sequence $A$ with $n$ elements, denoted $A[1..n]$, is a series of values of the same type, possibly with duplication. Ordering is part of the definition of the sequence.
- Examples:
  - $A = (4, 1, 2, 5)$ is a sequence of natural numbers
  - $S = \text{“Hello”}$ is a sequence (strong) of characters
- Elements of a sequence are denoted by their indexes, ranging from 1 to $n$
  
  \[
  A[1] = 4 \quad S[1] = \text{‘H’}
  \]
Example: calculator

- **Specification:** repeatedly display a list of mathematical operations: +, −, ×, ÷
- **Solution:**
  Repeat
  Show menu
  input operator, operands a, b
  if operator = ‘+’ display a + b
  else if operator = ‘−’ display a − b
  until operator = ‘Q’

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
**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” 
  ```

**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”

Debugging using trace

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

```
count ← 0
sum ← 0
while count < 5
    input x
    count ← count + 1
    sum ← sum + 1
display sum
```

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

1.3a Describe the loop control structure
1.3b Describe the debugging process*

References

D. Keil and R. Johnson. Problem solving and program design. www.framingham.edu/~dkeil/