Lecture 1

Principles of Programming and
Software Engineering

Outline

- Introduction
- 2.1 Life cycle of software
- 2.2 Achieving a modular design
- 2.3 Six key programming issues

Problem Solving and Software Engineering

- Coding without a solution, design increases debugging time

- Software engineering facilitates development of programs when a large amount of
  - Software
  - People (planning, organization and communication)

  are involved

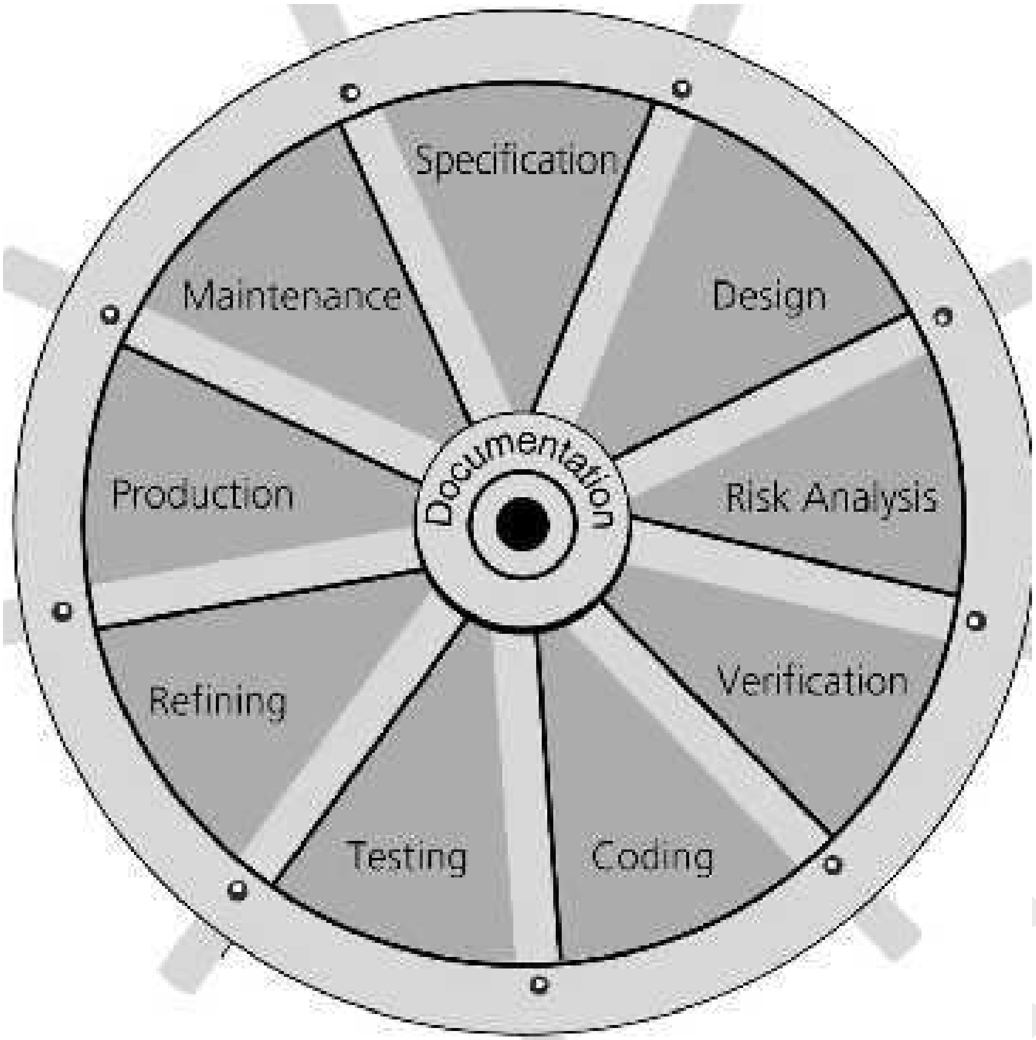
What is Problem Solving?

- The entire process of taking the statement of a problem and developing a computer program to solve the problem
  - Example: To solve a quadratic equation

Program = algorithm + data structure

  - Algorithm: a step-by-step specification of a method to solve a problem within a finite amount of time
  - Data structure: ways to store information

The Life Cycle of Software as a water wheel



We'll cover only aspects that play a crucial role in data structures

- Specification
- Design
- Verification
- Coding
- Testing

The other parts will be covered in later semesters, especially in Software Engineering

Phase 1: Specification

Make the problem statement precise and detailed

For example:

- What is the input data?
- What data is valid and what data is not valid?
- Who will use the software, what user interface should be used?

A prototype program can clarify the problem: a simple program that simulates the behavior and illustrates the user interface

Phase 2: Design

Divide a large problem to small modules:

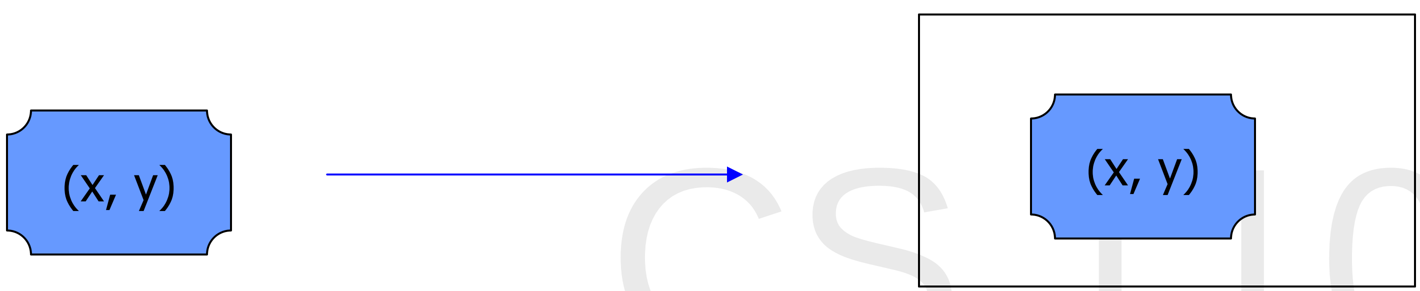
- Loosely coupled modules are independent
- Each module should perform one well-defined task (highly cohesive)
- Specify data flow among modules
  - E.g., purpose, assumptions, input, and output
  - It is NOT a description of what methods to use to solve the problem; just a decomposition into smaller tasks

Phase 2: Design (cont)

- View Specifications as a contract

  Example: To design a method for a shape object that moves it to a new location on the screen. Possible specifications:

  - *The method will receive an (x, y) coordinate.*
  - *The method will move the shape to the new location on the screen*



Phase 2: Design (cont)

- A module's specification should not describe a method of solution.
- Method specifications include precise *pre-condition* and *post-conditions ;* identify the method's formal parameter, etc.
- Incorporate existing software components in your design.

Phase 2: Design (cont)

First-draft specifications

move (x, y)
// Move a shape to a new location on the screen
// **Pre-condition:** The calling program provides an
// (x, y) pair, both integers.
// **Post-condition:** The shape is moved to the new location
// (x, y)

Phase 2: Design (cont)

Revised specifications

move (x, y)
// Move a shape to a new location on the screen
// **Pre-condition:** The calling program provides an
// (x, y) pair, both integers, where
// 0 <= x <= MAX_XCOOR, 0 <= y <= MAX_YCOOR,
// where MAX_XCOOR and MAX_YOOR are class
// constants that specify the maximum coordinate values.
// **Post-condition:** The shape is moved to the new location
// (x, y)

Phase 4: Verification

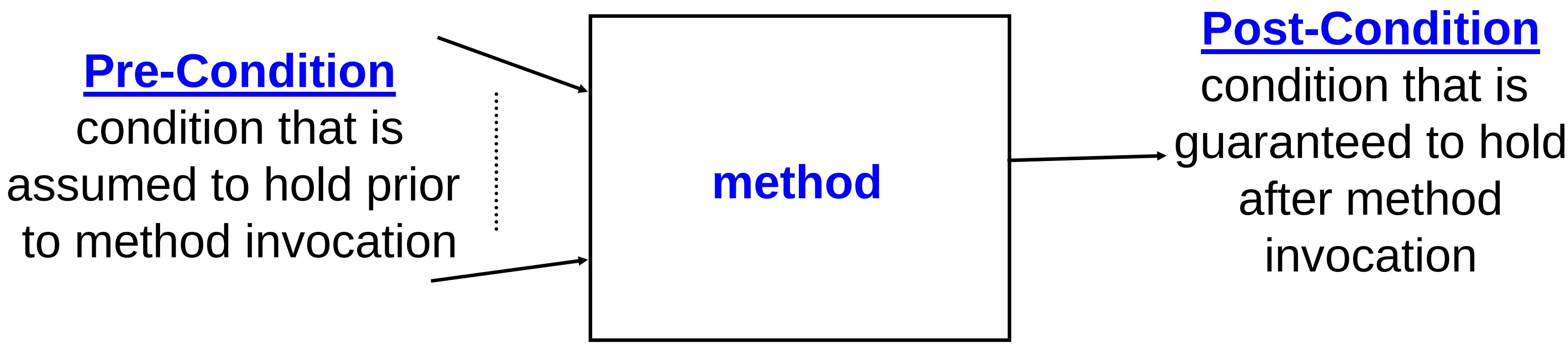
- Formal theoretical methods are available for proving the correctness of an algorithm
  - still a research subject
- Some aspects of the verification process
  - Assertion
  - Invariant

An assertion is a statement about a particular condition at a certain point in an algorithm

An invariant is a condition that is always true at a particular point of the algorithm

Phase 4: Verification - Assertion

- An assertion is a statement about a particular condition at a certain point in an algorithm.
  - special case: pre/post-conditions



Phase 4: Verification - Example

Revised specifications

move (x, y)
// Move a shape to a new location on the screen
// **Pre-condition:** The calling program provides an
// (x, y) pair, both integers, where
// 0 <= x <= MAX_XCOOR, 0 <= y <= MAX_YCOOR,
// where MAX_XCOOR and MAX_YOOR are class
// constants that specify the maximum coordinate values.
// **Post-condition:** The shape is moved to the new location
// (x, y)

Phase 4: Verification - Invariant

- An invariant is a condition that is always true at a particular point of the algorithm
  - For example, a loop invariant is a condition that is true before and after each execution of an algorithm's loop.

Example of Loop Invariant

```
// computes the sum of item[0], item[1],
// ... item[n-1] for n>=1
// Loop invariant: sum is the sum of the
// elements item[0] through item[j-1]
int sum = 0
int j = 0;
while (j<n) {
  sum += item[j];
  ++j;
} // end while
```

More on Loop Invariants

Steps to establish the correctness of an algorithm:

- The invariant must be true initially
- An execution of the loop must preserve the invariant
- The invariant must capture the correctness of the algorithm
- After the loop terminates
  - The loop must terminate!

Loop Invariant – Establish correctness

```
int sum = 0
int j = 0;
```

The invariant is true here

```
while (j<n) {
```

The invariant is true here

```
  sum += item[j];
  ++j;
```

The invariant is true here

```
} // end while
```

The invariant is true here

Phase 5: Coding

- Translating the design into a particular programming language

- Coding is a relatively minor phase in the software life cycle.

Phase 6: Testing

- Design a set of test data to test the program

- Testing is both a science and an art

Phase 7: Refining the Solution

- Often to make some simplifying assumptions during the design of the solution
  - develop a working program under these assumptions

- Add refining sophistication
  - do not require complete re-design

What is a good solution?

- When the total cost incurred over all phases of the life cycle is minimal
- Programs must be well structured and documented – the “hub” of our water wheel
- Efficiency is important
  - Using the proper algorithms and data structures can lead to significant differences in efficiency
  - In many instances, the specific style of coding matters less than the choice of data structures

Achieving a Modular Design

A few principles to respect:

- Abstraction and Information Hiding
- Object-Oriented Design
- Top-Down Design

Learn how to use interface classes; rely more on private attributes, etc.

For example, wish to deal with circle, triangle, rectangle; probably can think of dealing "shapes" first.

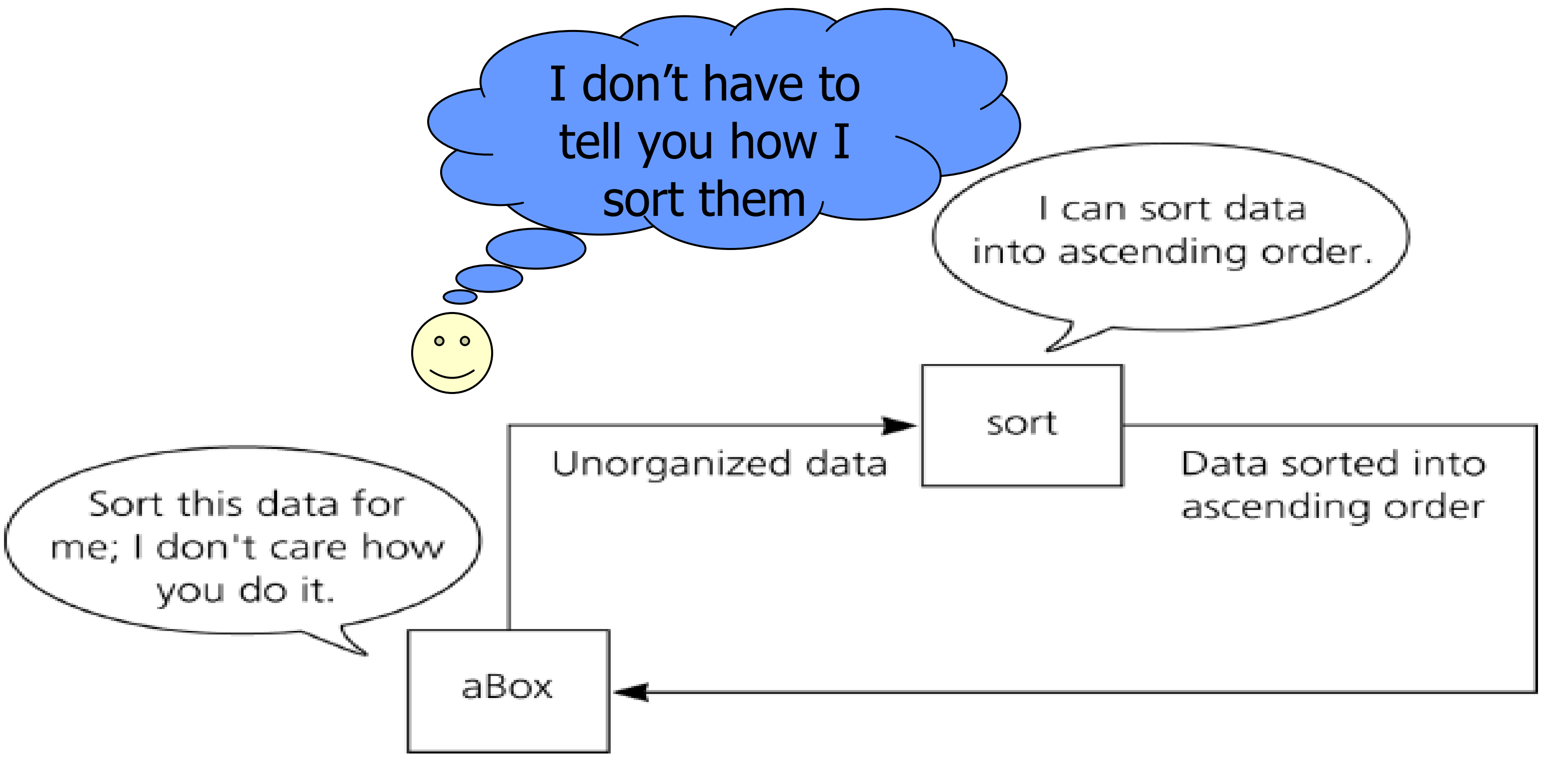
1st Principle: Abstraction and Information Hiding

- Specify what to do, not how to do it

- Write specifications for each modules before implementing it
  - Specifications do not indicate how to implement a module
  - Specify what a method does, not how to do it
  - Specify what you will do to data, not how to do it

Sorting as an example

- The details of the sorting algorithm are hidden from other parts of the solution



2

2nd Principle: Object-Oriented Design

- Object encapsulate data and operations
- Encapsulation hides inner details

Example: A digital clock

- An object is an instance of a class

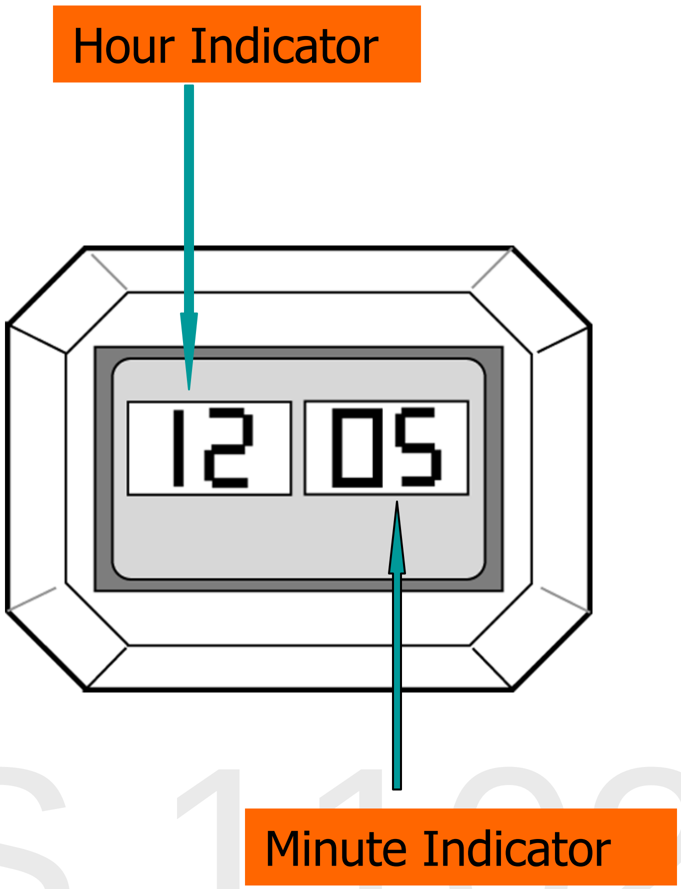
Example: A digital clock

Clock is an object and can perform operations such as

- Set the time
- Advance the time
- Display the time

The Hour indicator and minute indicator are also objects. Each indicator performs operations such as

- Set its value
- Advance its value
- Display its value



Object-Oriented Programming

Three principles of OOP:

- Encapsulation: Objects combine data and operations
- Inheritance: Classes can inherit properties from other classes
- Polymorphism: Objects can determine appropriate operations at execution time

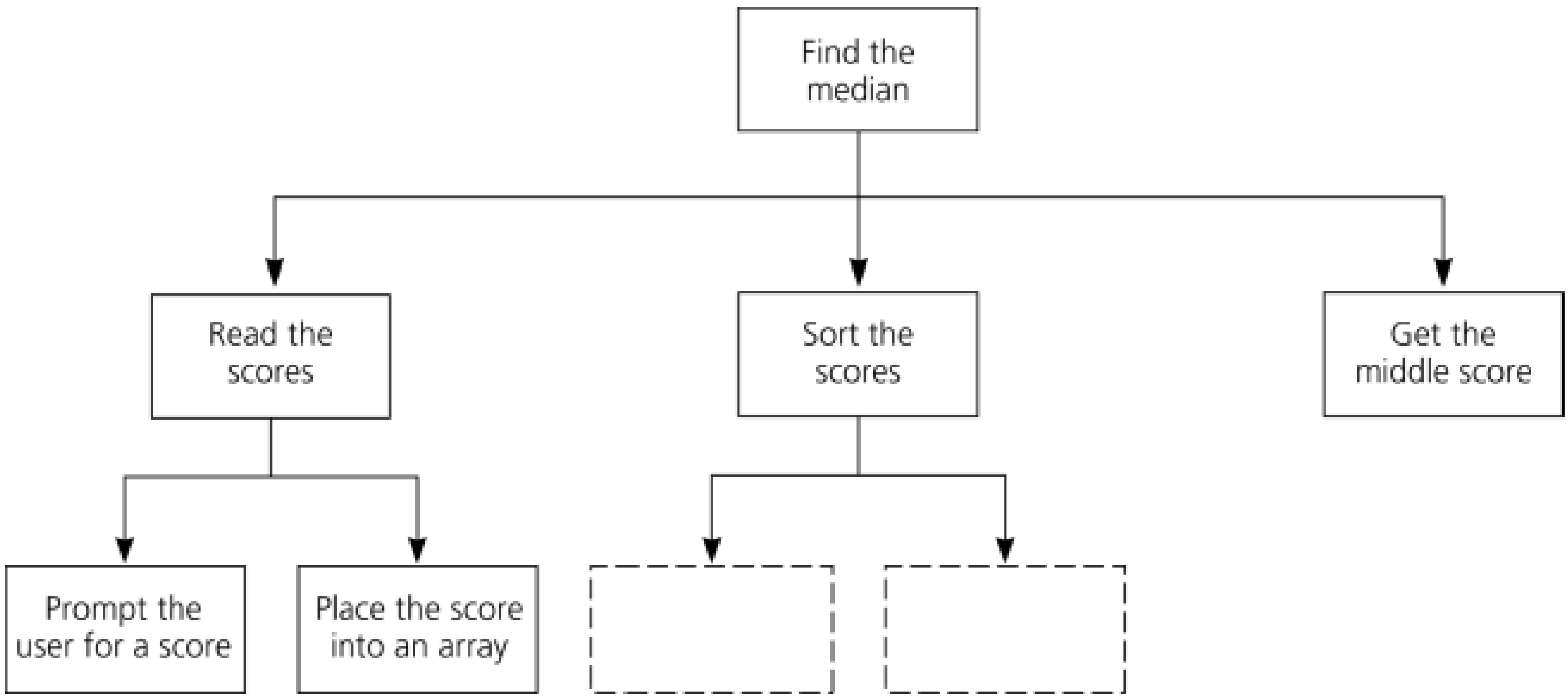
3rd principle: Top-Down Design

Use it:

- When designing an algorithm for a method
- When the emphasis is on algorithms and not on the data.

- A structure chart shows the relationship among modules.
- A solution consisting of independent tasks.

Example: Find the Median Score



Six Key Programming Issues

1. Modularity
2. Modifiability
3. Ease of use
4. Fail-safe programming
5. Style
6. Debugging

Modularity

- Facilitates programming
- Isolates errors
- Programs are easy to read
- Isolates modifications
- Eliminates redundancies

Modifiability

- Methods make a program easier to modify
- Named constants make a program easier to modify

Ease of Use

- A good user interface, for example, prompt user for input
- A good manual

Fail-Safe Programming

A fail-safe program is one that will perform reasonably no matter how anyone use it:

- Check for errors in input
- Check for errors in logic
- Methods should check their invariants
- Methods should enforce their preconditions
- Methods should check the values of their arguments

Style

- Extensive use of methods
- Use of private data fields
- Error handling – In general, methods return a value or throw an exception but do not display a message.
- Readability – meaningful identifiers, indentation, etc.
- Documentation

Debugging

- Use either watches, assertions or temporary System.out.println statements to find logic errors
- Systematically check a program's logic to determine where an error occurs

Summary

Today we've examined:

- Software engineering
- The life cycle of software
- Modular solution
  - Object-oriented and Top-down design