Lecture 1: Principles of Programming & Software Engineering

Read: Chpt.1 & Appendix A, Carrano.

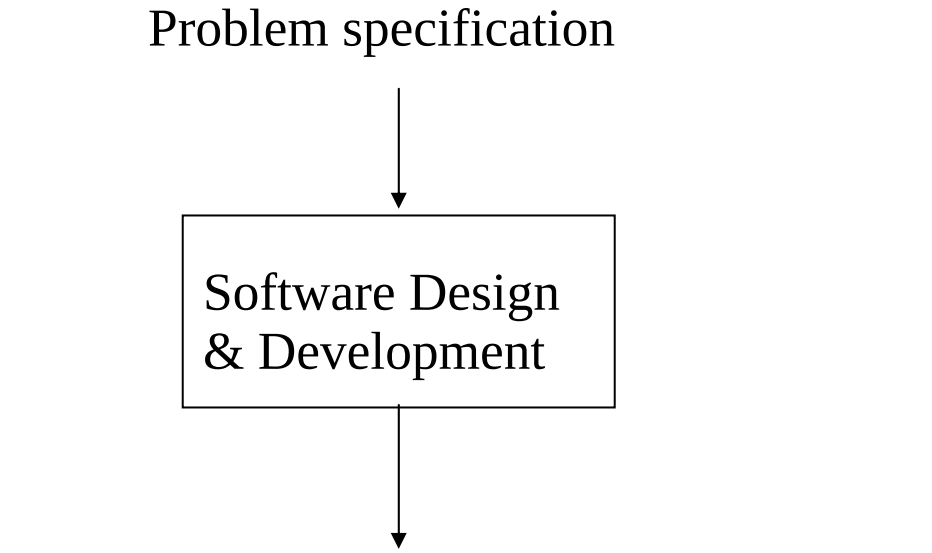
Q: What is EECS268?

A: This is *not* (the continuation of) a course in learning C++ syntax and semantics. This course is about learning problem solving using a computer:

- What kind of problems is solvable with computers?
- What is the cost in solving particular type of problems?
- How do we design computer-based solutions to a given problem?
- How do we implement the design?
- How do we debug and tune the resulting program?
- How do we manage complexity in large software systems?

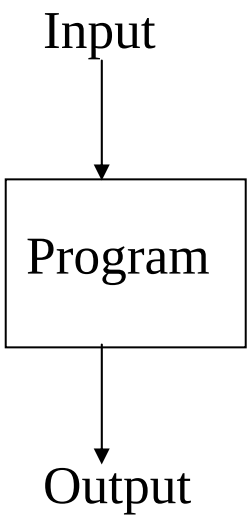
Tool: Object oriented design using C++.

Problem Solving:



Solving a Problem:

To develop a computer program that can generate correct output for ***all*** possible inputs to the problem.



Program consists of

- Algorithm
- Data structures

Algorithm: A sequence of step-by-step unambiguous instructions (of a method) that can be used to solve a given problem within a finite amount of time.

Data Structure: A collection of data objects organized in a "useful" way.

Program: Computer-based realization of solutions or algorithms to problem.

Program = $\mathbf{S_1;}$ $\mathbf{S_2;}$ **…** $\mathbf{S_m;}$

Statements:

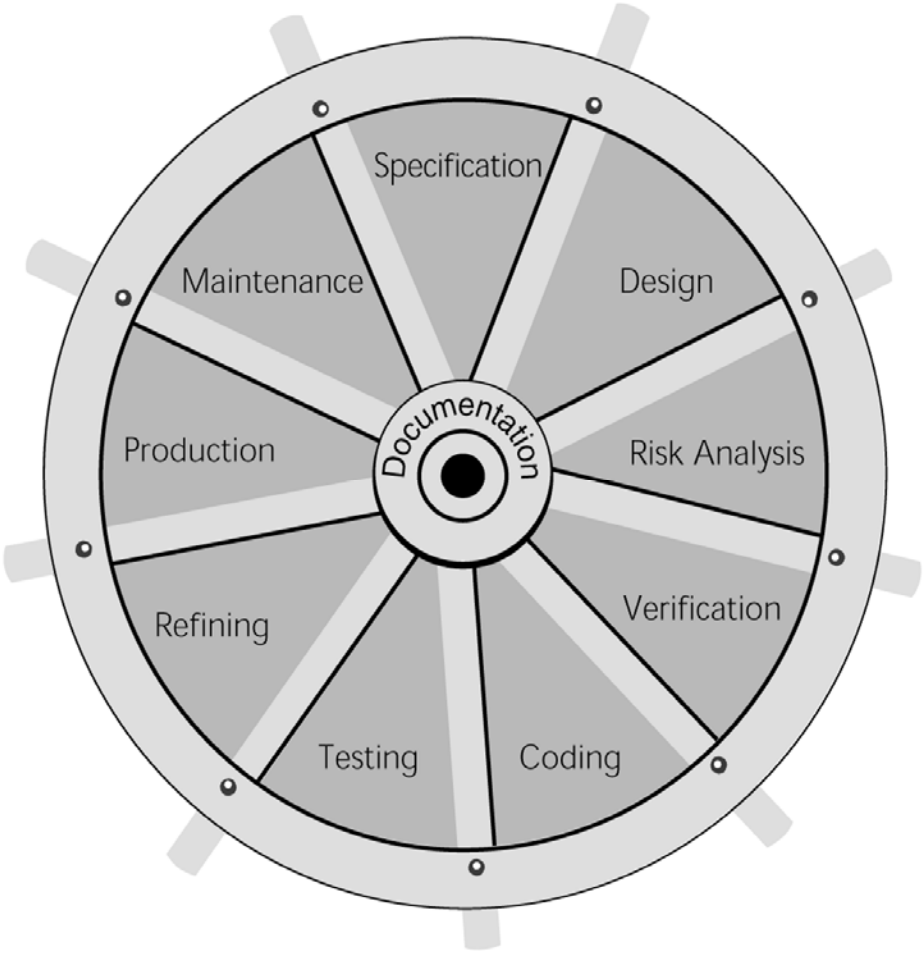
Simple statements
Compound/Structured statements
(loops, functions, …)

Software (system) = Collection of programs

Q: How do we develop and maintain a large software system for specific application?

- Using efficient algorithms and data structures
- Employ software engineering techniques to facilitate the development of computer programs.

Carrano's Water Wheel (Software Cycle):



Software Life Cycle:

1. **Specification:** Understand problem & I/O spec.
    - What is needed?
    - Who are the users?
    - What are the data?
    - What enhancements to the program are likely in the future?

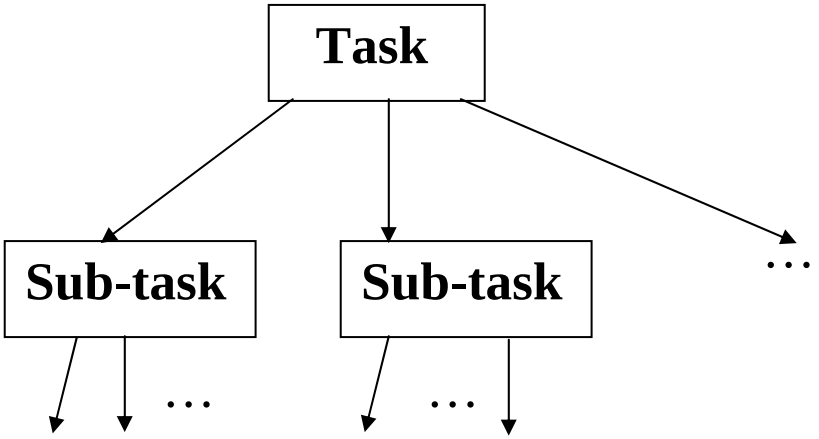
      …
2. **Design:** Algorithmic and software development.

   Algorithmic Design Paradigms:
    - Divide-and-conquer
    - Greedy approach
    - Dynamic programming
    - Backtracking
    - …

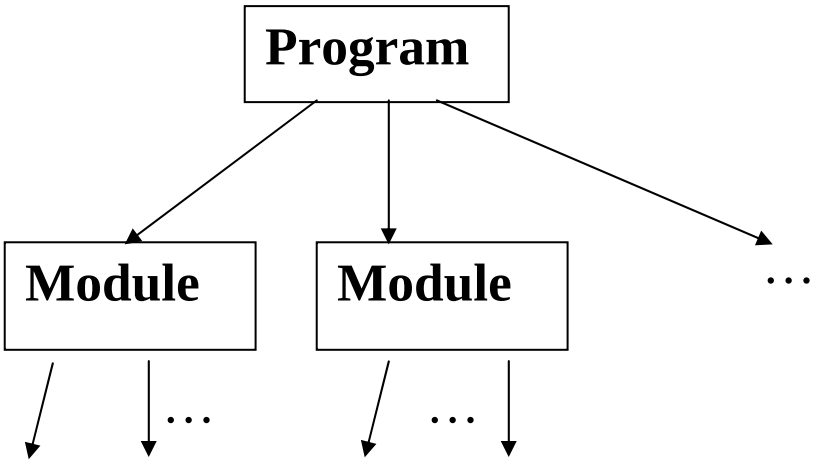
   Software Development:

   Top-Down Design with Module Programming:
    - Dividing the program into modules
    - Specifying the purpose of each module
    - Specifying the data flow among modules

Top-Down Design:



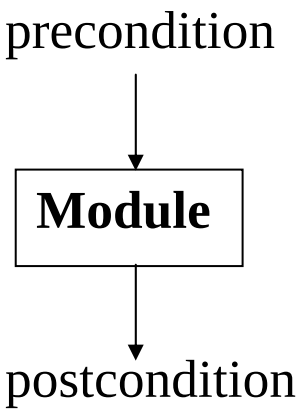
Module Programming:



Modules

- Self-contained units of code
- Should be designed to be:
  - Loosely coupled
  - Highly cohesive

Using Pre- and Post- conditions:



Precondition: Condition before execution of module.

Postcondition: Condition after execution of module.

3. **Risk Analysis:** Risk assessment.

   Can we get the tools we need? Will we have a sufficient budget? What happens if we fail?

   Techniques exist to identify, assess, and manage the risks of creating a software product.

4. **Verification:** Correctness of algorithm.

   Will the algorithm generate a correct output for each and every possible instance of the problem? There are formal techniques, but this field is in its infancy (assertions, invariants, loop-invariants, inductive arguments, …).

5. **Coding:** Program development.

   Select/Develop "good" data structures to support the implementation of the design using a chosen programming language.

6. **Testing:** Program verification.

   This is hard, but vitally important.

   Basic Techniques:

   - Test individual modules *before* assembling.
   - Test first vs. test last.
   - Removing logical error.
   - Using valid, invalid, random, and actual data for testing.

7. **Refining:** Improving accuracy and efficiency.

   Clean-up; more rigorous input testing; replace "dumb" implementations with "cleverer" one to improve performance. ***But*** need to make sure you use performance analyzers and the like to know where performance problems *really* lie.

8. **Production:**

   Distribute to intended users.

9. **Maintenance:**

   Fix previously undetected bugs and correct user-detected errors; add new features; improve performance and memory requirements, etc.

Remark:

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

The cost of a solution may include:

- Computer resources that the program consumes
- Difficulties encountered by users
- Consequences of a program that does not behave correctly

Using *Enduring Principles* (EP) in Problem Solving:

EP correspond to some of the fundamental concepts/techniques that have been evolved rather slowly over the years but have long lasting effects/impacts in the process of problem solving and software development.

Most Important EP in Problem Solving:

- Abstraction
- Algorithm

EP 1: Abstraction.

Abstraction helps us design, develop, and maintain software systems. It helps us to:

- Understand fundamental concepts in software development.
- Develop reusable software.
- Replace functionality and/or alter implementations of sub-systems without affecting clients.
- Manage complexity of software.

Examples of Abstraction:

Data abstraction and procedural abstraction

1. **Data Abstraction:**

   Focuses of the operations of data, not on the implementation of the operations

We will concentrate on the study of ***Abstract Data Types (ADTs)***, which is a collection of data objects and a set of operations that can be applied to the data. (Implementation details are hidden from users!)

Motivation: The study of ADTs allows us to create and manipulate instances of these objects *without* knowing anything about how they are implemented internally. An ADT's operations can be used without knowing how the operations are implemented, if the operations' specifications are known

Examples:

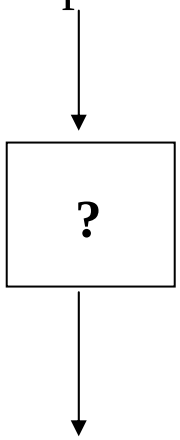
List, Stacks, Queues, Priority Queues, Search Trees, Tables.

Data Structure is a construct that can be defined within a programming language to store a collection of data.

2. Procedural/Functional Abstraction:

Focus on the functionality and the user interface of a function by separating the purpose of a function from its implementation.

Precondition: input



Postcondition: output

Motivation: It allows us to use functions *without* knowing anything about how they are implemented internally. Hence, we can concentrate more on problem solving and algorithmic development

Example: You certainly don't want to write these functions yourself:

Math: sin, cos, sqrt, ...

Using Abstraction in Modular Design:

- Separates the purpose of a module from its implementation
- Specifications for each module are written before implementation
- Separates the purpose of a function from its implementation
- Focuses of the operations of data, not on the implementation of the operations
- Using information hiding to
  - Hide details within a module
  - Ensure that no other module can tamper with these hidden details

  The public view of a module is described by its specifications but the private view of a module consists of details which should not be described by the specifications.

EP 2: Algorithms.

The use of algorithms allows us to concentrate on problem solving instead of specific programming language issues.

Design & Analysis of Algorithms:

- **Design:** Developing algorithms using basic design paradigms such as divide-and-conquer, backtracking, greedy method, etc.
- **Analysis:** Proving the correctness and efficiency of algorithm.

Program Development:

Using

- Top-down design (TDD)
- Object-oriented design (OOD)
- Abstraction
- Information hiding

Top-Down Design with Stepwise Refinement:

- Begin by listing the main tasks your program must accomplish.
- For each task identified, list all subtasks that must be performed.
- Continue this process until a task/subtask is simple enough that it can be accomplished with just a few lines of code.
- The process of continuing to lower levels of design in this fashion is called stepwise refinement.

Example: Computing the median of a set of test scores.

Rough Design:

Input scores	(How? Where? Format?)
Compute median	(How?)
Output median	(How? Where? Format?)

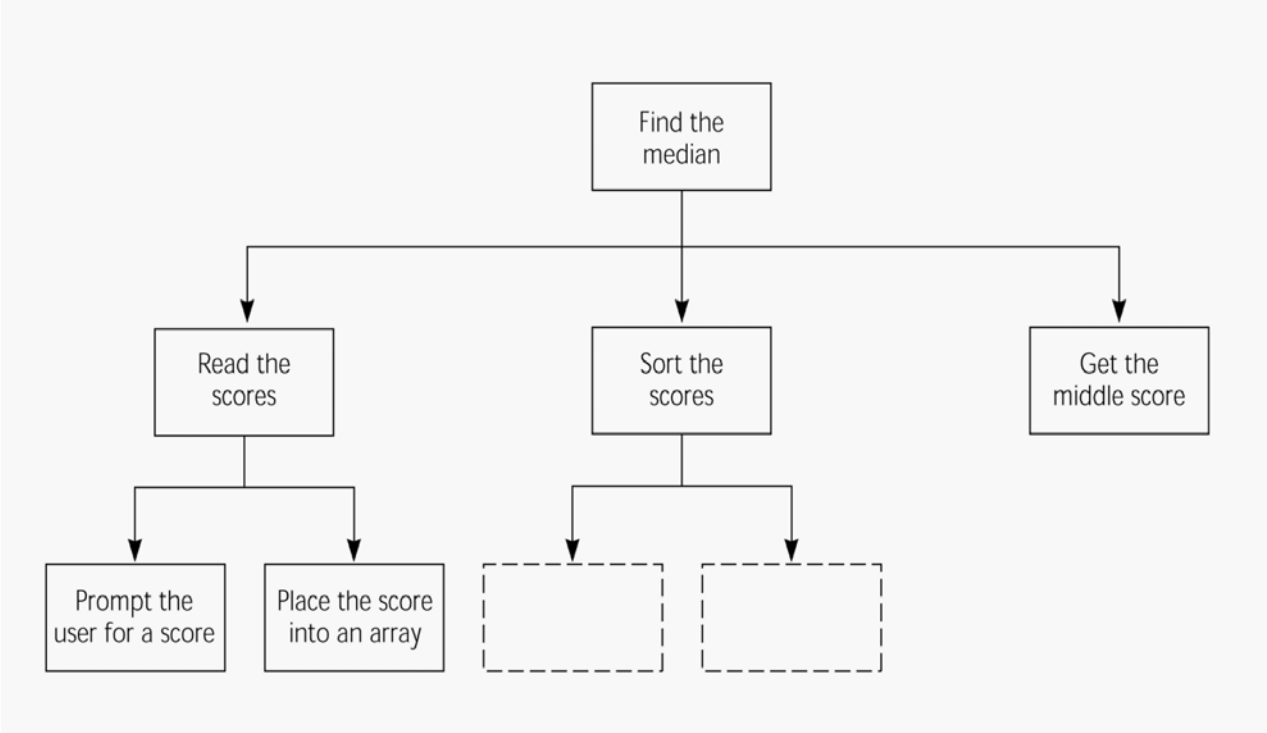
Refinement:

Prompt user for score
Store scores in an array

Sort the array
Pick the median

Return median

Hierarchy of Modules in Median Finding:



Remark:

Each task/subtask forms a module during the design process (needed only specified the pre- and postconditions).

Object-Oriented Design (OOD):

Program design technique bases on the concepts of objects and classes.

Advantages of OOD Approach:

- Existing classes can be reused
- Program maintenance and verification are simpler.

Three Basic Object-Oriented Programming (OOP) Principles:

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

Abstraction:

- Data abstraction
- Function abstraction

Information Hiding:

Using private/protected data member variable/function to:

- Protect the integrity of data
- Allow abstraction for users/programmers

When designing a module, information hiding allows us to

- Hide details within a module
- Ensure that no other module can tamper with these hidden details

Public view of a module is described by its specifications, but the private view of a module consists of details which should not be described by the specifications

Modeling OOD using Unified Modeling Language (UML):

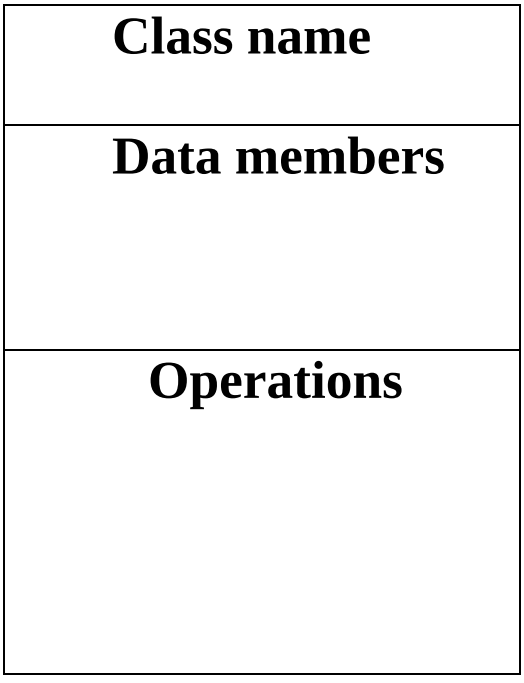
UML is a system modeling language used to express OOD using:

- Diagrams (solution design)
- Text-based description (classes design)

Basic Building Block: Class diagram.

Classes in an OOD are represented by class diagrams by specifying the name of the class, the data members of the class, and the operations.

Class Diagram:



UML Class Syntax:

Data Member Field:

visibility name: type = defaultValue

visibility: + (public)
– (private)
(protected)

name: Identifier

type: Generic data type
integer
float
string
…

defaultValue: Initial value of data member

UML Function Syntax:

Operation Field:

visibility name(parameterList): returnType {propertyString}

visibility: Same

name: Name of operation

parameterList: List of comma separated parameters

Syntax:
direction name: type = defaultValue

direction: in (input)
out (output)
inout (input/output)

name: Name of parameter

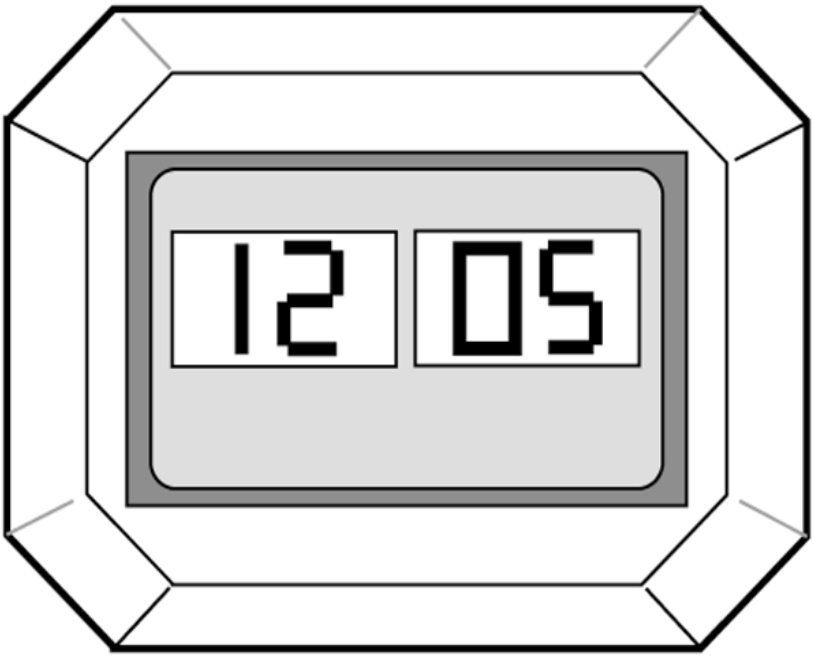
type: Same

defaultValue: Default value of parameter

returnType: Data type of the result of the operation

propertyString: Property values of operation

Example: A digital clock class.



UML diagram:

clock
hour **minute** **second**
setTime() **advanceTime()** **displayTime()**

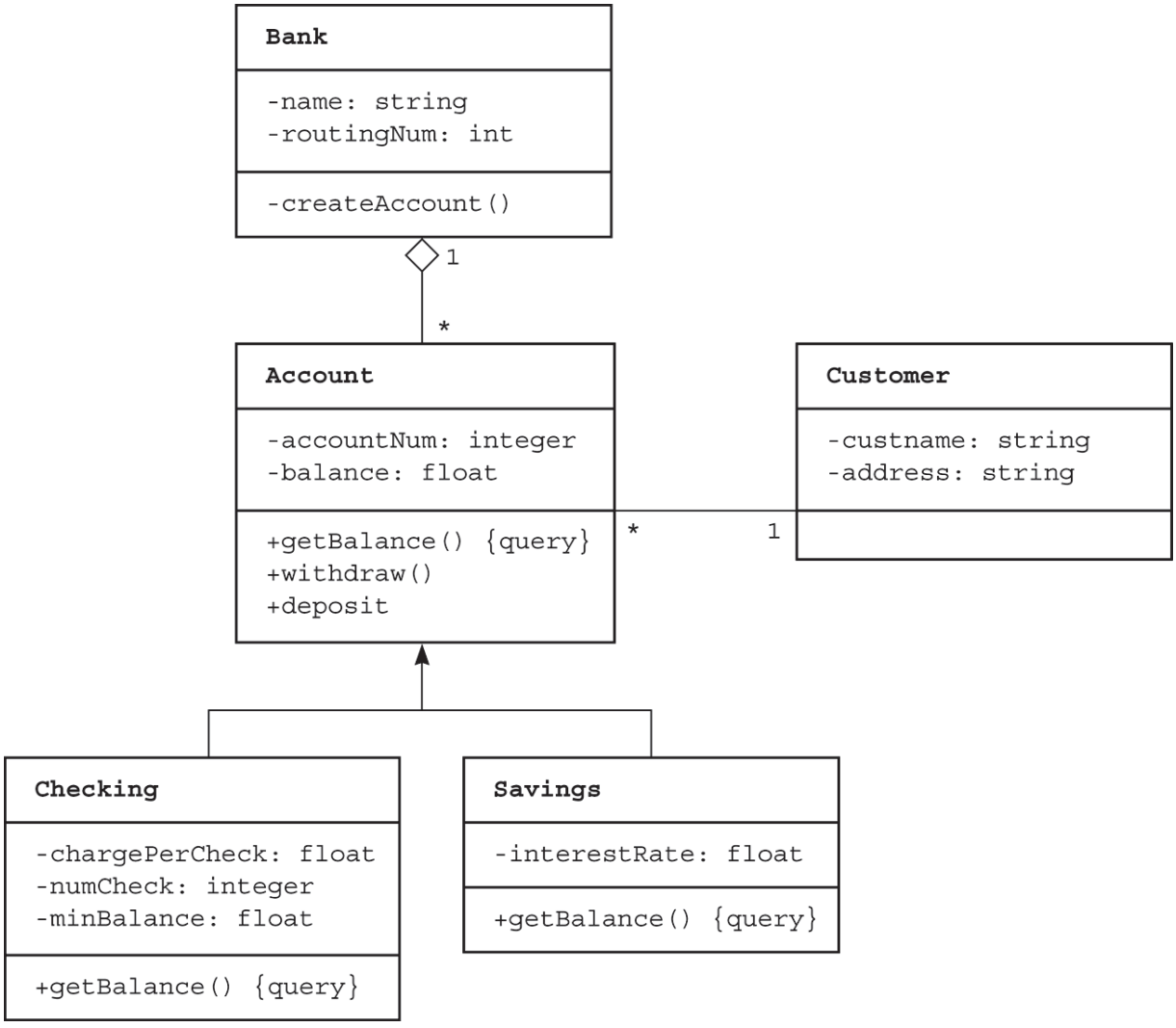
–hour: integer
–minute: integer
–second: integer

+setTime(in hr: integer, …)
–advanceTime()
+displayTime() {query}

Some Basic UML Constructs:

- 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.

Banking System Example (P.17):



Six Key Programming Issues:

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

Eight Issues of Style in Programming:

- Extensive use of functions
- Use of private data fields
- Avoidance of global variables in functions
- Proper use of reference arguments
- Proper use of functions
- Error handling
- Readability
- Documentation

Remark: You are required to follow these guidelines in doing your programming assignments.