Lecture
Chapter 2
Software Development

Large Software Projects

- Software Design
  - Team of programmers
  - Cost effective development
- Organization
- Communication

Problem Solving

- Analysis of the problem
- Multiple solutions
- Selection of the best solution
  - Software solution
    - Algorithms
      - Step by step specification of the method
    - Data collection & storage
      - Organize the data in a manner that facilitates use
  - Personnel solution
- Implementation of the solution
- Acceptance of the solution

Software Life Cycle

Specification
Design
Risk Analysis
Verification
Coding
Testing
Refining

Incremental &/or Iterative development

Production
Maintenance

Life Cycle

Specification

- ✓ Specify all aspects of the problem
- ✓ Communicate with non-programmers
- ✓ Clarify unclear aspects of the specifications
- ✓ Determine
  - ➢ valid input data
  - ➢ appropriate error messages
  - ➢ user base
  - ➢ appropriate interface
  - ➢ output required
  - ➢ output format
  - ➢ appropriate documentation
- ✓ Specify potential future enhancements
- ✓ Create a prototype program for user approval

Design

- ✓ Identify and design modules – objects
- ✓ Classes should be designed so that the objects are
  - ➢ loosely coupled
  - ➢ highly cohesive

> **Tightly coupled objects**
> - High flow of information between objects
>
> **Loosely coupled objects**
> - Changes in one object will have minimal effect on the other objects
>
> **Highly cohesive objects perform only one well-designed task**

- ✓ Interactions between objects
  - ➢ Messages – method calls – data flows -- information flow
  - ➢ Specify, in detail, the assumptions, input, and output for each method
    - ▪ What data within the object is utilized by the method
    - ▪ What does the method assume
    - ▪ What actions are performed by the method
    - ▪ How is the data changed by the method
  - ➢ Specifications serve as a contract between method & the outside
  - ➢ Contract serves to
    - ▪ systematically decompose the program into smaller tasks
    - ▪ delineate responsibilities among programmers &/or modules
  - ➢ Precondition
    - ○ the conditions that must exist at the beginning of a method
  - ➢ Postcondition
    - ○ the conditions that will exist at the end of the method

Sufficient time spent in design ➔ less time required in implementation
Reuse of Software Components **Java Application Interface**

Risk Analysis

- **all projects**
- **specific projects**
- **known risks**
- **unknown risks**
- **affects**
    - **timetable**
    - **costs**
    - **life**
    - **health**
- **techniques to identify, assess, and manage some risks**

Verification

formal methods for proving algorithms correct are incomplete

<u>Useful Aspects Of The Verification Process</u>

- **assertion – statement regarding a condition at a specified point in an algorithm**
    - **Java assertion statements – check the Jgrasp assertion feature**

- **invariant – condition that is always true a a specified point in a program**
    - **loop invariant – always true before & after each loop execution**

- **proving an algorithm is correct**
    - **proving each step of the algorithm is correct, i.e.,**
      for all appropriate assertions,
      an assertion before the step remains the same after the step is executed
    - **errors encountered during the process should be corrected and the specifications modified if appropriate**

The result is less errors encountered during the programming

Coding

- **Translating the design into a particular programming language**
- **Not a major portion of the life cycle for most projects**

Testing

- **Bottom Up Testing**
- **Range Limits**
- **Idiot Proof**

Refining the Original Solution

- **Retesting**

The phases of the Life Cycle are

- **not completely isolated from one another and**
- **not linear**

Installation

- **Acceptance Testing**

Production

- **Black Box Cutover**
- **Parallel Testing**

Maintenance

- **Users detect errors**
- **Request**
    - **enhancements, i.e., require more features**
    - **modifications to the existing software to better serve the users**

Fredrick Brooks
Mythical Man-Month
Ten Pounds in a Five Pound Bag
First Solution -- Second Solution

First solution is based on ~~some~~ simplifying assumptions
Refined solution provides a sophisticated program that meets the original program specifications

Good Solution – Computer Program

- **performs specified task**
- **real tangible cost**
    - **total cost over all phases of the life cycle + the burial costs**
    - **efficiency – choice of a solutions components**
        - **algorithms**
        - **choice of data structure & storage**
    - **code reusability**
        - **code libraries & open source repositories**
        - **components developed within a project**
            - ✓ **reused multiple times in the same projects**
            - ✓ **reused unchanged in other projects**
            - ✓ **embedded in other components**

Abstraction

- **Procedural Abstraction**
  - embed procedure in a "Black Box", i.e., a module, a class, an object
  - users of the procedure know it's pre and post conditions, but not how it performs its tasks
  - separates the purpose of a method from its implementation
  - helps to segment the design into loosely coupled and highly cohesive modules
  - Java API, e.g., Math.abs(), Math.tan(), etc.

- **Data Abstraction**
  - collection of data
  - set of operations on that data, with a focus on
    - what the operations will do to the data
    - without any consideration of how to implement them

- **Abstract Data Type (ADT)**
  - collection of data
  - <u>specification</u> of the set of operations that can manipulate that data
  - implemented by defining data structures and creating methods in a selected computer language

- **Problem Solving**
  - Develop ADT' s and algorithms at the same time
  - Global Algorithm ➔ support algorithms and ADT's (top-down design)
  - Collection of feasible ADT's and related algorithms ➔
    Set of Potential Global Algorithms (bottom-up design)
  - Algorithms ➔ clever data structure solution
  - Data Structure ➔ clever algorithm solution

- **Information Hiding**
  - Abstraction ➔
    - write a specification that describes the outside, i.e., public view of a module
    - identify details that should remain private, i.e., should be hidden from the public view

User of a Module ➔ do not worry about the implementation details.
Implementer of a Module ➔ do not worry about the use of the module.

Object Oriented Design

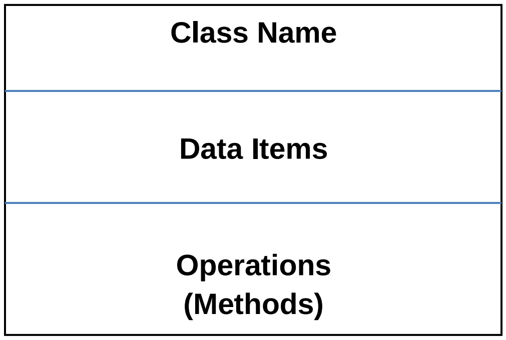
Object

- **encapsulates**
  - **data**
  - **actions**
- **identification**
  - **nouns ➔ objects**
  - **verbs ➔ actions**
  - **objects of the same type ➔ class**
- **inheritance – classes can inherit properties from other classes**
- **polymorphism**
  - **objects can determine appropriate operations at execution time**
  - **the outcome of a particular operation depends upon the objects being acted upon**
- **functional decomposition (top-down design)**
  - **break a task into smaller subtasks**
  - **structure chart (Prichard 3rd ed pg 101)**

General Design Guidelines

1. **Use Object-Oriented Design (OOD), i.e., <u>objects & ADTs</u>, and Functional Decomposition (FD), i.e., <u>algorithmic structure charts</u>, to produce modular solutions**
2. **Use OOD for problems that primarily involve data**
3. **Use FD to design algorithms for an objects operations**
4. **As a first approach, use FD to design solutions to <u>problems that emphasize algorithms over data</u>. During the design process, keep watch for signs that the OOD approach would yield better results.**
5. **Keep the focus on <u>WHAT</u> when designing ADTs & algorithms**
6. **Incorporate reusable software components into the design where possible**

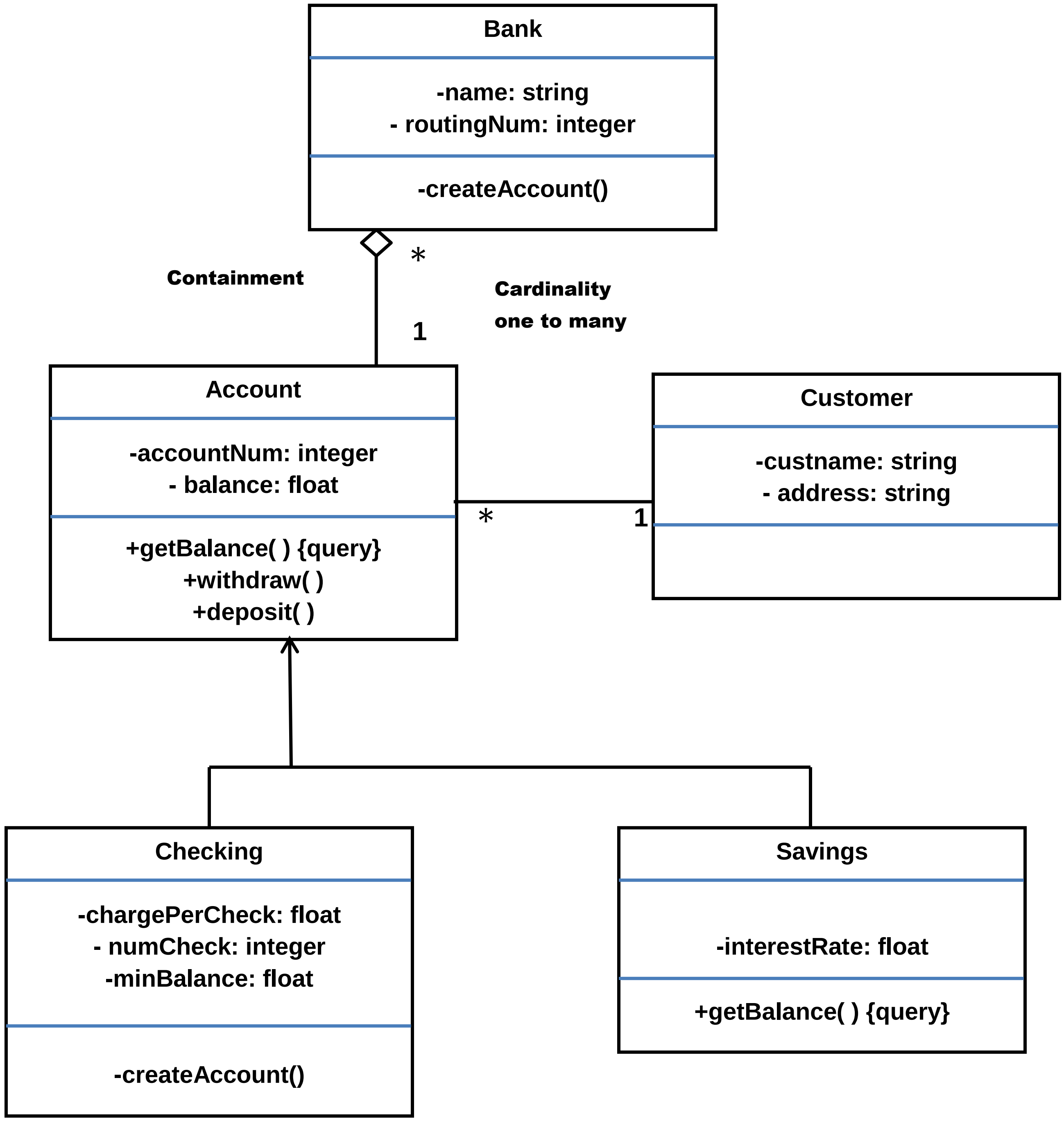
Unified Modeling Language (UML)



data members
visibility name: type = defaultValue
--, +   hour: integer = 1

operations
visibility name(parameter-list); return-type {property-string}
-, +
parameter-list
direction name: type = defaultValue
in, out getHour: integer = 1

+ setTime(in hr: integer, in min: integer, in sec: integer): void


Bank
-name: string
- routingNum: integer
-createAccount()
Containment
*
Cardinality
one to many
1
Account
-accountNum: integer
- balance: float
+getBalance( ) {query}
+withdraw( )
+deposit( )
*
1
Customer
-custname: string
- address: string
Checking
-chargePerCheck: float
- numCheck: integer
-minBalance: float
-createAccount()
Savings
-interestRate: float
+getBalance( ) {query}

Object-Oriented Programming

- **time expended on design will increase**
- **solution will be more general than is absolutely necessary**
- **implementation time will be reduced**
- **improved program maintenance and verification stages**
    - **change in ancestor class ➔ change in all descendant classes**
    - **add descendant classes that do not affect the ancestor class**
    - **add descendant classes modifies the ancestor's original behavior**

Design Procedure

- **specification of each class including data and operations**
- **implementation of each class**
- **identify families of related classes**
    - **identify the ancestor class**
    - **implement the descendant classes**
- **test each class**
    - **write test program to exercise each operation wrt specifications**
    - **write test programs to exercise sets of classes working together to solve a larger programs (expands up to the set which encompasses the entire project)**

Key Programming Issues

Strive for modularity in all phases of the program-solving problem

- **Modularity**
    - **Programming tasks become more difficult as the size and complexity of a program grows; modularity reduces the rate at which the difficulty grows**
    - **Permits team programming, i.e., permits the continuation of current programming efforts**
    - **Isolates errors; debugging a modular program is reduced to debugging many small programs**
    - **Facilitates reading the program**
    - **Isolates modifications; modifying modular program is reduced to a small set of relatively simple modifications to isolated parts of the program**
    - **Eliminates redundant code**

- **Modifiability**
- **Ease of Use**

- **Fail-Safe Programming**
    - **Input Errors Prohibited by Code**
    - **Logic Errors Eliminated by EXTENSIVE Testing**
    - **Hardware Errors Eliminated**
        - **Hardened Hardware – Isolated Power Sources**
        - **Multiple Systems Voting on Outcomes**
    - **Life-Support Systems**
    - **Financial Systems**

- **Idiot-Proof Programming**
    - **Prichard 3rd ed pgs 112-116**

- **Style**
    - **Extensive Use of Methods**
    - **Private Data Fields**
        - **accessor methods**
        - **mutator methods**
    - **Error Handling**
    - **Readability**
        - **good structure & design**
        - **well-chosen identifiers that describe their purpose**
        - **readable indentation**
            - ✓ **next-line blocks**
            - ✓ **DO NOT USE end-of-line blocks**
            - ✓ **2-4 spaces**
            - ✓ **beware of <u>rightward drift</u>**
        - **blank lines to separate modules, methods, blocks, etc.**
        - **well-chosen documentation**

Program Documentation

1. **Program Comment**
    a. **Statement of Purpose**
    b. **Author & Date**
    c. **Description of Input/Output**
    d. **Description of how to Use the Program**
    e. **Assumptions about the type of data expected**
    f. **Statement of Exceptions**
    g. **Brief Description of the Major Classes**
2. **Comment in Each Class**
    a. **Statement of Purpose**
    b. **Description of the Data contained in the Class**
3. **Comment at the Beginning of Each Method**
    a. **Statement of Purpose**
    b. **Preconditions**
    c. **Postconditions**
    d. **Method Called**
4. **Comments in the Bodies of Selected Methods that explain important features or subtle logic**

Write the Documentation While Writing the Code

Make sure that you write
Comments for Users of the Methods
and
Comments for Programmers who will revise the Implementations

- **Debugging**
    - **Debugger**
        - **step by step**
        - **breakpoints**
    - **System.out.println(…) statements**
    - **Use the invariants established for various parts of the program**
    - **Dump selected data structures, e.g., arrays**