1. Development process: Software system life cycle

- Requirements/Specification
  - understand what your program is supposed to do
  - get a detailed and precise description of the input and output requirements
  - write prototype program if it is needed
  - write a requirement specification document

- Design and Analysis
  - break the problem down into management modules, each module should be a well defined object or component.
  - use technique such as object oriented design and top-down design.
  - specify the data flow among modules
  - example modules: manage student records, manage courses etc.
  - write a precise module specification document
• specify functions of each module (note: a module may contain several functions).

define precondition and postcondition of each function

precondition: statement of conditions that must exist at the beginning of a function

postcondition: statement of conditions at the end of function

example: to insert a new student record

    Precondition: new record to be inserted
    Postcondition: indicate the operation was successful, if yes, the record was inserted.

write a precise functional specification document

• write a design specification document. It should include risk analysis, software and hardware resource, engineer time needed to complete the project.
• Refinement, Verification and Coding
  • choose the representations of data objects and outline algorithms for each operation on them.
  • Verify correctness of algorithms: using theoretical methods such as mathematical induction
  • correctness proof: too difficult to develop for large projects
  • at this point, you may discover that your original design might not be the best one, go back to Design & Analysis phase again.
  • after settled down on specific data objects and algorithms, translate the data objects & algorithms into high-level programs.

• Testing and Debugging.
  • this is done by tester (QA)
  • carefully develop a set of test data that include all possible scenarios (need to have test plan)
  • goal: error removal, i.e. fix bugs

• Production and maintenance
  • package the software and distribute it to users
  • may need to fix more bugs, add new features, port to new platforms etc
2. How to produce good solutions to problems?

- Consider cost
  - computer resources that are used by your program
  - ease of use
  - development cost and time needed

- Main goals of a good program
  - should always produce correct solution (either expected results or error messages)
  - should not crash (especially, should not cause other programs or systems to crash)
  - program must be well structured and documented
    - should be able to maintain by other engineers, including bug fixes and future enhancement
    - should include error handling (check for error conditions and provide good error messages)
  - efficiency is very important, clean and simplicity programs are also important
  - may include debug features (the programmer may turn this feature on or off)
3. How to achieve modular design?

- Goals: Abstraction and information hiding
  - each module can be viewed as a box
  - it contains specifications (what it does) – abstraction
  - implementation details should be hidden from users
  - this course focus on various general ways to organize and access data
  - Abstract data type (ADT) is an abstract model that specifies the type of data stored and the operations that support data.
  - To implement an ADT:
    - using a data structure to store data. Example: arrays or pointers
    - writing programs for operations to manipulate the data structure.

- Object oriented design approach
  - hiding the representation of objects and implementation of operations from its users (encapsulation)
  - the designer may alter the representation objects and implementation of operations as long as the user interface remains the same.
  - Use OOP to support ADT implementations.
  - will talk about C++ inheritance and polymorphism!
• Top-down design (for designing good algorithms)

  • break down a function into smaller sub-functions.

  • a sub-function can be further broken down into smaller sub-sub-functions etc.

  • the process continue until the bottom functions are simple enough to translate directly to C++ functions (usually, less than 50 statements)

• Example (from the text) :

  Find the median

  Read the scores

    Prompt the user for a score
    Place score into array

  Sort the scores

  ...

  Get the middle score
• Basic Unified Modeling Language (UML)

• Use to express OOD

• Syntax for data members

\[
\text{visibility name: type } = \text{default Value}
\]

• Syntax for operations

\[
\text{visibility name parameterList: return Type } \{ \text{property String} \}
\]

visibility : + (public), - (private) or # (protected)

type or return Type : the data type

parameterList : each parameter is direction name:type=default Value

direction is in (input), out (output) or inout (both)

property String : property values that apply to the operation.
For the text book, we only use “query” – do not modify any data

See  http://www.omg.org

• Example : the class Clock

// data member
-hour: integer
-minute: integer
-second: integer

// operations
+setTime(in hr: integer; in min: integer; in sec: integer)
-advanceTime()
+displayTime() {query}