Chapter 1: Programming Principles

- Object Oriented Analysis and Design
- Abstraction and information hiding
- Object oriented programming principles
- Unified Modeling Language
- Software life-cycle models
- Key programming issues
Abstraction and Information Hiding

• Abstraction
  – provide an easier higher-level interface to mask possibly complex low-level details
  – functional abstraction
    • separates the purpose of a module from its implementation
    • specifications for each module are written before implementation
  – data abstraction
    • focuses on the operations of data, not on the implementation of the operations
Abstraction and Information Hiding

• Abstract data type (ADT)
  – a collection of data and a set of operations on the data
  – you can use an ADT’s operations without knowing their implementations or how data is stored, if you know the operations’ specifications

• Data structure
  – construct that is defined within a programming language to store a collection of data
Abstraction and Information Hiding

• Information hiding
  – hide details within a module
  – ensure that no other module can tamper with these hidden details
  – public view of a module
    • described by its specifications
  – private view of a module
    • implementation details that the specifications should not describe
Object-oriented languages enable us to build classes of objects.

A class combines:
- attributes of objects of a single type
  - typically data
  - called data members
- behaviors (operations)
  - typically operate on the data
  - called methods or member functions
Principles of Object-Oriented Programming (OOP)

• Three principles of OOP
  – Encapsulation
    • objects combine data and operations
    • hides inner details
  – Inheritance
    • classes can inherit properties from other classes
    • existing classes can be reused
  – Polymorphism
    • objects can determine appropriate operations at execution time
Object-Oriented Analysis & Design

• A team of programmers for a large software development project requires
  – an overall plan
  – organization
  – communication

• Problem solving
  • understanding the problem statement
  • design a conceptual solution
  • implement (code) the solution

• OOA/D is a process for problem solving.
Object-Oriented Analysis & Design

• Analysis – Process to develop
  – an understanding of the problem
  – the requirements of a solution
    • what a solution must be and do, and not how to design or implement it

• Object-oriented analysis (OOA)
  – expresses an understanding of the problem and the requirements of a solution in terms of objects
  – objects represent real-world objects, software systems, ideas
  – OOA describes objects and their interactions among one another
Object-Oriented Analysis & Design

• Object-oriented design
  – expresses an understanding of a solution that fulfills the requirements discovered during OOA
  – describes a solution in terms of
    • software objects, and object collaborations
    • objects collaborate when they send messages
  – creates one or more models of a solution
    • some emphasize interactions among objects
    • others emphasize relationships among objects
Applying the UML to OOA/D

- Unified Modeling Language (UML)
  - tool for exploration and communication during the design of a solution
  - models a problem domain in terms of objects independently of a programming language
  - visually represents object-oriented solutions as diagrams
  - enables members of a programming team to communicate visually with one another and gain a common understanding of the system being built
Applying the UML to OOA/D

• UML use case for OOA
  – A set of textual scenarios (stories) of the solution
    • each scenario describes the system’s behavior under certain circumstances from the perspective of the user
    • focus on the responsibilities of the system to meeting a user’s goals
    • main success scenario (happy path): interaction between user and system when all goes well
    • alternate scenarios: interaction between user and system under exceptional circumstances
  – Find noteworthy objects, attributes, and associations within the scenarios
Applying the UML to OOA/D

• An example of a main success scenario
  – customer asks to withdraw money from a bank account
  – bank identifies and authenticates customer
  – bank gets account type, account number, and withdrawal amount from customer
  – bank verifies that account balance is greater than withdrawal amount
  – bank generates receipt for the transaction
  – bank counts out the correct amount of money for customer
  – customer leaves bank
Applying the UML to OOA/D

• An example of an alternate scenario
  – customer asks to withdraw money from a bank account
  – bank identifies, but fails to authenticate customer
  – bank refuses to process the customer’s request
  – customer leaves bank
Applying the UML to OOA/D

Figure 1-2 Sequence diagram for the main success scenario

---

1.14 Figure 1-2  Sequence diagram  for the main success scenario

EECS 268 Programming II
Applying the UML to OOA/D

Figure 1-3  Sequence diagram showing the creation of a new object

bank:Bank → «create» → account:Account

Figure 1-3  Sequence diagram showing the creation of a new object
Applying the UML to OOA/D

• UML class (static) diagram
  – Represents a conceptual model of a class of objects in a language-independent way
  – Shows the name, attributes, and operations of a class
  – Shows how multiple classes are related to one another
Applying the UML to OOA/D

Figure 1-4 Three possible class diagrams for a class of banks
Applying the UML to OOA/D

Figure 1-5  A UML class diagram of a banking system
Applying the UML to OOA/D

- **Class relationships**
  - association
    - classes know about each other (Bank – Customer classes)
  - aggregation (Containment)
    - One class contains instance of another class (Bank – Account classes)
    - lifetime of the containing and contained may be the same (composition)
  - generalization
    - indicates a family of classes related by inheritance
    - “Checking” and “Savings” inherit attributes and operations of “Account”
The Software Life Cycle

• Describes phases of s/w development from conception, deployment, replacement to deletion

• Iterative and Evolutionary Development
  – many short, fixed-length iterations build on the previous iteration
  – iteration cycles through analysis, design, implementation, testing, and integration of a small portion of the problem domain
  – early iterations create the core of the system; further iterations build on that core
Software Life Cycle

• Rational Unified Process (RUP) Development
  – RUP uses the OOA/D tools
  – four development phases
    • Inception: feasibility study, project vision, time/cost estimates
    • Elaboration: refinement of project vision, time/cost estimates, and system requirements; development of core system
    • Construction: iterative development of remaining system
    • Transition: testing and deployment of the system
Rational Unified Process (RUP) Development Phases

*Figure 1-8* Relative amounts of work done in each development phase
Software Life Cycle

• Waterfall Method of Development
  – develops a solution through sequential phases
    • requirements analysis, design, implementation, testing, deployment
  – hard to correctly specify a system without early feedback
  – wrong analysis leads to wrong solution
  – outdated (less used)
  – do not impose this method on RUP development
Achieving a Better Solution

• Analysis and design improve solutions
• Cohesion – perform one well-defined task
  – for self-documenting, easy-to-understand code
  – easy to reuse in other software projects
  – easy to revise or correct
  – Robust – less likely to be affected by change;
    performs well under unusual conditions
  – promotes low coupling
Achieving a Better Solution

• Coupling – not dependent on other modules
  – system of modules with low coupling is
    • easier to change and understand
  – module with low coupling is
    • easier to reuse and has increased cohesion
  – coupling is necessary for objects to collaborate
    • should be minimized; well-defined
  – class diagrams show dependencies among classes, and hence coupling
Achieving a Better Solution

• Minimal and complete interfaces
  – class interface declares publicly accessible methods (and data)
  – classes should be easy to understand, and so have few methods
  – complete interface
    • provide all methods consistent with the responsibilities of the class
  – minimal interface
    • provide only essential methods
Operation Contracts

• A module’s operation contract specifies its
  – purpose, assumptions, input, output

• Begin during analysis, finish during design
  – used to document code

• Contract shows the responsibilities of one
  module to another

• Does not describe how the module will
  perform its task
Operation Contracts

• Specify data flow among modules
  – what data is available to a module?
  – what does the module assume?
  – what actions take place?
  – what effect does the module have on the data?

• Precondition
  – statement of conditions that must exist before a module executes

• Postcondition
  – statement of conditions that exist after a module executes
Operation Contracts

• First draft specifications -- sort(anArray, num)
  // Sorts an array.
  // Precondition: anArray is an array of num integers; num > 0.
  // Postcondition: The integers in anArray are sorted.

• Revised Specifications -- sort(anArray, num)
  // Sorts an array into ascending order.
  // Precondition: anArray is an array of num integers; 1 <= num <= MAX_ARRAY, where
  // MAX_ARRAY is a global constant that specifies the maximum size of anArray.
  // Postcondition: anArray[0] <= anArray[1] <= ... <= anArray[num-1], num is unchanged
Verification

- **Assertion** – a statement about a particular condition at a certain point in an algorithm
  - like, preconditions and postconditions
- **Invariant** – a condition that is always true at a certain point in an algorithm
- **Loop invariant** – a condition that is true before and after each loop iteration
  - can be used to detect errors before coding is started
A solution is good if:
- the total cost it incurs over all phases of its life cycle is minimal

The cost of a solution includes:
- computer resources that the program consumes
- difficulties encountered by users
- consequences of a program that does not behave correctly

Programs must be well structured and documented

Efficiency is one aspect of a solution’s cost
Key Issues in Programming

- Modularity
- Style
- Modifiability
- Ease of Use
- Fail-safe programming
- Debugging
- Testing
Key Issues in Programming: Modularity

• Modularity has a favorable impact on
  – Constructing programs
  – Debugging programs
  – Reading programs
  – Modifying programs
  – Eliminating redundant code
Key Issues in Programming: Style

- Use of private data members
- Proper use of reference arguments
- Avoidance of global variables in modules
- Error handling
- Readability
- Documentation
Key Issues in Programming: Modifiability

• Modifiability is easier through the use of
  – Named constants
  – The typedef statement
Key Issues in Programming: Ease of Use

• In an interactive environment, the program should prompt the user for input in a clear manner
• A program should always echo its input
• The output should be well labeled and easy to read
Key Issues in Programming: Fail-Safe Programming

• Fail-safe programs will perform reasonably no matter how anyone uses it
• Test for invalid input data and program logic errors
• Check invariants
• Enforce preconditions
• Check argument values
Key Issues in Programming: Debugging

• Programmer must systematically check a program’s logic to find where an error occurs

• Tools to use while debugging:
  – single-stepping
  – watches
  – breakpoints
  – print statements
  – dump functions
Key Issues in Programming: Testing

• Levels of testing
  – Unit testing: Test methods, then classes
  – Integration testing: Test interactions among modules
  – System testing: Test entire program
  – Acceptance testing: Show system complies with requirements

• Types
  – Open-box (white-box or glass-box) testing
    • test knowing the implementation
    • test all lines of code (decision branches, etc.)
  – Closed-box (black-box or functional) testing
    • test knowing only the specifications
Key Issues in Programming: Testing

- Developing test data
  - include boundary values
  - need to know expected results

- Techniques
  - assert statements to check invariants
  - disable, but do not remove, code used for testing
    - /* and */
    - boolean checks
    - pre-processor macros