Deja Vu?

- In the past: Structured = Good
- Today: Object-Oriented = Good
- e.g.,
  - Object-oriented languages are good
  - Ada is an object-oriented language
    Therefore, Ada is good
- Note, there is even an object-oriented COBOL!

Goals

- Demystify the hype surrounding OOD and OOP
- Focus on OOD/OOP principles, methods, notations, and tools
- Relate OOD/OOP to traditional development methods

Overview

- What are object-oriented (OO) methods?
  - OO methods provide a set of techniques for analyzing, decomposing, and modularizing software system architectures
  - In general, OO methods are characterized by structuring the system architecture on the basis of its objects (and classes of objects) rather than the actions it performs

- What are the benefits of OO?
  - OO enhances key software quality factors of a system and its constituent components

- What is the rationale for using OO?
  - In general, systems evolve and functionality changes, but objects and classes tend to remain stable over time
Software Quality Factors

- Object-oriented techniques enhance key external and internal software quality factors, e.g.,
  1. External (visible to end-users)
     (a) Correctness
     (b) Robustness and reliability
     (c) Performance
  2. Internal (visible to developers)
     (a) Modularity
     (b) Flexibility/Extensibility
     (c) Reusability
     (d) Compatibility (via standard/uniform interfaces)

OOA, OOD, and OOP

- Object-oriented methods may be applied to different phases in the software lifecycle
  - e.g., analysis, design, implementation, etc.

- OO analysis (OOA) is a process of discovery
  - Where a development team models and understands the requirements of the system

- OO design (OOD) is a process of invention and adaptation
  - Where the development team creates the abstractions and mechanisms necessary to meet the system’s behavioral requirements determined during analysis

OOA, OOD, and OOP (cont’d)

- Is it also useful to distinguish between object-oriented design (OOD) and object-oriented programming (OOP)
  - OOD is relatively independent of the programming language used
  - OOP is primarily concerned with programming language and software implementation issues

- Obviously, the more consistent the OOD and OOP techniques, the easier they are to apply successfully in real-life...

OOA, OOD, and OOP (cont’d)

- Basic Definitions
  1. Object-Oriented Design
     - A method for decomposing software architectures based on the objects every system or subsystem manipulates
     - Rather than “the” function it is meant to ensure
  2. Object-Oriented Programming
     - The construction of software systems as structured collections of Abstract Data Type (ADT) implementations, plus inheritance and dynamic binding
Object-Oriented Design Topics

- Object-oriented design concepts include:
  - Decomposition/Composition
  - Abstraction
    * Modularity
    * Information Hiding
    * Virtual Machine Hierarchies
  - Separating Policy and Mechanism
  - Subset Identification and Program Families
  - Reusability

- Main purpose of these design concepts is to manage software system complexity by improving software quality factors

Object-Oriented Programming Topics

- Object-oriented programming features and techniques include
  - Data abstraction and information hiding
  - Active (rather than passive) types
  - Genericity
  - Inheritance and dynamic binding
  - Programming by contract
  - Assertions and exception handling

- Throughout the course we’ll discuss how these OOP features and techniques improve software quality
  - e.g., correctness, reusability, extensibility, reliability, etc.

Review: Goals of the Design Phase

- Decompose System into Modules
  - i.e., identify the software architecture via “clustering”
    * In general, clusters should maximize cohesion and minimize coupling

- Determine Relations Between Modules
  - Identify and specify module dependencies
    * e.g., inheritance, composition, uses, etc.
  - Determine the form of intermodule communication, e.g.,
    * global variables
    * parameterized function calls
    * shared memory
    * RPC or message passing

Review: Goals of the Design Phase (cont’d)

- Specify Module Interfaces
  - Interfaces should be well-defined
    * facilitate independent module testing
    * improve group communication

- Describe Module Functionality
  - Informally
    * e.g., comments or documentation
  - Formally
    * e.g., via module interface specification languages
Decomposition/Composition

- Decomposition and composition are concepts common to all software life-cycle and design techniques.

- The basic concepts are very simple:
  1. Select a portion of the problem (initially, the whole problem)
  2. Decompose the selected portion into one or more constituent components using the design method of choice
     - *e.g.*, functional vs. data structured vs. object-oriented
  3. Determine and depict how the components interact (*i.e.*, composition)
  4. Repeat steps 1 through 3 until some termination criteria is met (*e.g.*, customer is satisfied, run out of money, etc.)

Decomposition/Composition (cont’d)

- A major challenge of the design phase for a system is to determine what the primary units of decomposition and composition ought to be.

- Another way of looking at this is to ask “at what level of abstraction should the modules be specified?”

- Typical units of decomposition and composition include:
  - Subsystems
  - Virtual machine levels
  - Classes
  - Functions

Abstraction

- **Motivation**
  - Abstraction provides a way to manage complexity by emphasizing essential characteristics and suppressing implementation details

- Traditional abstraction mechanisms
  - Name abstraction
  - Expression abstraction
  - Procedural abstraction
    - *e.g.*, closed subroutines
  - Data abstraction
    - *e.g.*, ADTs
  - Control abstraction
    - *e.g.*, iterators, loops, multitasking, etc.
Modularity

- **Motivation**
  - Modularity is an essential characteristic of good designs since it:
    - Enables developers to reduce overall system complexity via *decentralized* software architectures
      - *i.e.,* divide and conquer
    - Enhances *scalability* by supporting independent and concurrent development by multiple personnel
      - *i.e.,* Separation of concerns

- To be both useful and reusable, modules should possess
  1. Well-specified abstract interfaces
  2. High cohesion and low coupling

Criteria for Evaluating Design Methods

- **Modular Decomposability**
  - Does the method aid decomposing a new problem into several separate subproblems?
    - *e.g.,* top-down functional design

- **Modular Composability**
  - Does the method aid constructing new systems from existing software components?
    - *e.g.,* bottom-up design

- **Modular Understandability**
  - Are modules separately understandable by a human reader?
    - *e.g.,* how tightly coupled are they?

Criteria for Evaluating Design Methods (cont’d)

- **Modular Continuity**
  - Do small changes to the specification affect a localized and limited number of modules?

- **Modular Protection**
  - Are the effects of run-time abnormalities confined to a small number of related modules?

- **Modular Compatibility**
  - Do the modules have well-defined, standard and/or uniform interfaces?
    - *e.g.,* “principle of least surprise”

Principles for Ensuring Modular Designs

- **Language Support for Modular Units**
  - Modules must correspond to syntactic units in the language used

- **Few Interfaces**
  - Every module should communicate with as few others as possible

- **Small Interfaces (Weak Coupling)**
  - If any two modules communicate at all, they should exchange as little information as possible
Principles for Ensuring Modular Designs (cont'd)

• **Explicit Interfaces**
  - Whenever two modules A and B communicate, this must be obvious from the text of A or B or both.

• **Information Hiding**
  - All information about a module should be private to the module unless it is specifically declared public.

Information Hiding

• **Motivation**
  - Details of design decisions that are subject to change should be hidden behind abstract interfaces
    * i.e., modules
  - Information hiding is one means to enhance abstraction

• Typical information to hide includes:
  - *Data representations*
  - *Algorithms*
  - *Input and Output Formats*
  - *Policies and/or mechanisms*
  - *Lower-level module interfaces*

Virtual Machines

• **Motivation**
  - To reduce overall complexity, software system architectures may be decomposed into, more manageable “virtual machine” units

• A virtual machine provides an extended “software instruction set”
  - Provides additional data types and associated “software instructions” that extend the underlying hardware instruction set
  - Virtual machines allow incremental extensions to existing “application programmatic interfaces” (APIs)

Virtual Machine (cont’d)

• **Common examples of virtual machines include**
  - *Computer Architectures*
    * e.g., compiler → assembler → object code → microcode → gates, transistors, signals, etc.*
  - *Communication protocol stacks*
    * e.g., ISO OSI reference model, Internet reference model*
Virtual Machine (cont’d)

- Several challenges must be overcome to effectively use virtual machines as an architectural structuring technique:
  - Ensuring Adequate Performance:
    - It is difficult to obtain good performance at level \(N\) if below \(N\) are not implemented efficiently
    - This often requires implementing the virtual machine differently than the design may dictate...
  - Alleviating Inter-level Dependencies
    - To maximize reuse, it is essential to eliminate/reduce dependencies “between” virtual machine levels...
    - Therefore, virtual machines are often organized into hierarchical layers or levels of abstraction

Virtual Machine (cont’d)

- A “hierarchy” may be defined to reduce module interactions by restricting the topology of relationships between virtual machines
  - A relation defines a hierarchy if it partitions units into levels
    - Level 0 is the set of all units that use no other units
    - Level \(i\) is the set of all units that use at least one unit at level \(< i\) and no unit at level \(> i\)

- Advantages of hierarchical structuring
  - Facilitates independent development of levels or layers
  - Isolates ramifications of change
  - Enables rapid prototyping

Virtual Machine (cont’d)

- Relations that define hierarchies:
  - Uses
  - Is-Composition-Of
  - Is-A
  - Has-A

- The first two are general to all design methods, the latter two are more particular to object-oriented design and programming

Virtual Machine (cont’d)

- The Uses Relation
  - \(X\) Uses \(Y\) if the correct functioning of \(X\) depends on the availability of a correct implementation of \(Y\)
  - Note, \(uses\) is not necessarily the same as \(invokes\).
    - Some invocations are not \(uses\) relations
      - e.g., error logging
    - Some \(uses\) relations don’t involve direct invocations
      - e.g., message passing, interrupts, shared memory access
  - A simple, but effective design heuristic is to design \(uses\) relations that yield a hierarchy
    - \(i.e.,\) avoid cycles in the “uses graph”
The Uses Relation (cont'd)

- Allow X to use Y when:
  - X is simpler because it uses Y
    - e.g., standard C library routines, OSI layers
  - Y is not substantially more complex because it is not allowed to use X
    - i.e., hierarchies should be designed to be useful, and not just to blindly satisfy soft-
      ware engineering principles
  - There is a useful subset containing Y and not X
    - i.e., allows sharing and reuse of Y
  - There is no conceivably useful subset containing X but not Y
    - i.e., Y is necessary for X to function correctly

How should recursion be handled?

- A hierarchy in the uses relation is essential for designing non-trivial reusable software systems
  - e.g., asynchronous communication protocols, call-back schemes, signal handling, etc.
  - Upcalls are one way to control these non-hierarchical dependencies

The Is-Comp osed-Of Relation (cont'd)

- The Is-Comp osed-Of Relation
  - The is-comp osed-of relationship illustrates how the system is statically decomposed into its constituent components
    - X is-comp osed-of {x_i} if X is a group of units x_i that share some common purpose
    - A graphical description of a system's architecture may be specified by the is-comp osed-of relation such that:
      - Non-terminal are "virtual" code
      - Terminals are the only units represented by "actual" code

Many programming languages support the is-comp osed-of relation via some higher-level module or record structuring technique

- Note: the following are not equivalent:

1. Level (virtual machine)
2. Module (an entity that hides a secret)
3. A subprogram (a code unit)
4. A record (a passive data structure)

- Modules and levels need not be identical, as a module may have several components on sev-
  eral levels of a uses hierarchy
  - Likewise, a level may be implemented via several modules...
Virtual Machine (cont’d)

- The Is-A and Has-A Relations
  - These two relationships are associated with object-oriented design and programming languages that possess inheritance and class features
  - Is-A (descendant or inheritance) relationship
    * class X possesses Is-A relationship with class Y if instances of class X are specialization of class Y
    * e.g., a square is a specialization of a rectangle, which is a specialization of a shape...
  - Has-A (client or composition) relationship
    * class X possesses a Has-A relationship with class Y if instances of class X contain an instance(s) of class Y
    * e.g., a car has an engine and four tires...

Separate Policies and Mechanisms

- Motivation
  - Separate concerns between the what/when and the how at both the design and implementation phases

- Multiple policies may be implemented via a set of shared mechanisms
  - e.g., OS scheduling and virtual memory paging

- Same policy can be implemented by multiple mechanisms
  - e.g., reliable, non-duplicated, bytestream service can be provided by multiple communication protocols

- What is a policy and what is a mechanism is a matter of perspective...

Program Families and Subsets

- Program families are a collection of related modules or subsystems that form a reusable application framework, e.g.,
  - UNIX System V STREAMS I/O subsystem
  - Graphical user interface frameworks such as InterViews, MFC, and Fresco

- The components in a program family are similar enough that it makes sense to emphasize their similarities before discussing their differences

- Motivation
  - Program families are useful for implementing subsets
  - Reasons for providing subsets include cost, time, personnel resources, etc.

Program Families and Subsets (cont’d)

- Identifying subsets:
  - Analyze requirements to identify minimally useful subsets
  - Also identify minimal increments to subsets

- Advantages of subsetting:
  - Facilitates software system extension and contraction
  - Promotes reusability
  - Anticipates potential changes
Program Families and Subsets (cont’d)

- Program families support:
  - Different services for different markets
    * e.g., different alphabets, different vertical applications, different I/O formats
  - Different hardware or software platforms
    * e.g., compilers or OSs
  - Different resource trade-offs
    * e.g., speed vs. space
  - Different internal resources
    * e.g., shared data structures and library routines
  - Different external events
    * e.g., UNIX I/O device interface
  - Backward compatibility
    * e.g., sometimes it is important to retain bugs!