Motivation: Goals of the Design Phase

- Decompose System into Modules
  - *i.e.*, identify the software architecture
  - *Modules are abstractions that should:*
    * be independent,
    * have well-specified interfaces, and
    * have high cohesion and low coupling.

- Determine Relations Between Modules
  - Identify module dependencies
  - Determine the form of intermodule communication, *e.g.:
    * global variables
    * parameterized function calls
    * shared memory
    * RPC or message passing
Primary Design Phases

- **Preliminary Design**
  - External design describes the real-world model
  - Architectural design decomposes the requirement specification into software subsystems
- **Detailed Design**
  - Formally specify each subsystem
  - Further decomposed subsystems, if necessary
- Note: in design phases the orientation moves
  - from customer to developer
  - from what to how

Key Design Concepts and Principles

- Important design concepts and design principles include:
  - Decomposition
  - Abstraction
  - Subset Identification
  - Information Hiding
  - Virtual Machine Structuring
  - Modularity
  - Separating Policy and Mechanism
  - Hierarchy
- Main purpose of these concepts and principles is to manage software system complexity and improve software quality factors.

Decomposition

- Decomposition is a concept common to all life-cycle and design techniques.
- Basic concept is very simple:
  1. Select a piece of the problem (initially, the whole problem)
  2. Determine its components using the mechanism of choice, e.g., functional vs data structured vs object-oriented
  3. Show how the components interact
  4. Repeat steps 1 through 3 until some termination criteria is met (e.g., customer is satisfied, run out of money, etc.;-))

Decomposition (cont’d)

- Some guiding decomposition principles
  - Because design decisions transcend execution time, modules might not correspond to execution steps . . .
  - Decompose so as to limit the effect of any one design decision on the rest of the system
  - Remember, anything that permeates the system will be expensive to change
  - Modules should be specified by all information needed to use the module and nothing more
Abstraction

- Abstraction provides a way to manage complexity by emphasizing essential characteristics and suppressing implementation details.
- Allows postponement of certain design decisions that occur at various levels of analysis, e.g.,
  - Representational/Algorithmic considerations
  - Architectural/Structural considerations
  - External/Functional considerations

Abstraction (cont’d)

- Three basic abstraction mechanisms
  - Procedural abstraction
    * e.g., closed subroutines
  - Data abstraction
    * e.g., ADTs
  - Control abstraction
    * iterators, loops, multitasking, etc.

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.
- Modules should communicate only through well-defined interfaces.
- Each module is specified by as little information as possible.
- If internal details change, client modules should be minimally affected (may require recompilation and relinking, however . . .)

Information Hiding (cont’d)

- Information to be hidden includes:
  - Data representations
    * i.e., using abstract data types
  - Algorithms e.g., sorting or searching techniques
  - Input and Output Formats
    * Machine dependencies, e.g., byte-ordering, character codes
  - Policy/mechanism distinctions
    * i.e., when vs how
    * e.g., OS scheduling, garbage collection, process migration
  - Lower-level module interfaces
    * e.g., Ordering of low-level operations, i.e., process sequence
Modularity

- A Modular System is a system structured into highly independent abstractions called modules.
- Modularity is important for both design and implementation phases.
- Module prescriptions:
  - Modules should possess well-specified abstract interfaces.
  - Modules should have high cohesion and low coupling.

Modularity (cont'd)

- Modularity facilitates certain software quality factors, e.g.:
  - Extensibility - well-defined, abstract interfaces
  - Reusability - low-coupling, high-cohesion
  - Compatibility - design “bridging” interfaces
  - Portability - hide machine dependencies
- Modularity is an important characteristic of good designs because it:
  - allows for separation of concerns
  - enables developers to reduce overall system complexity via decentralized software architectures
  - enhances scalability by supporting independent and concurrent development by multiple personnel

Modularity (cont’d)

- A module is
  - A software entity encapsulating the representation of an abstraction, e.g., an ADT
  - A vehicle for hiding at least one design decision
  - A “work” assignment for a programmer or group of programmers
  - a unit of code that
    - has one or more names
    - has identifiable boundaries
    - can be (re-)used by other modules
    - encapsulates data
    - hides unnecessary details
    - can be separately compiled (if supported)

- A module is not necessarily a subroutine or arbitrary pieces of code
- All ADTs are modules
  - Note all modules are ADTs, however.
    - e.g., groups of functionally related procedures (such as sorting) can form a module, but not be an ADT.
Modularity (cont’d)

- A module interface consists of several sections:
  - Imports
    * Services requested from other modules
  - Exports
    * Services provided to other modules
  - Access Control
    * not all clients are equal! (e.g., C++’s distinction between protected/private/public)
  - Heuristics for determining interface specification
    * define one specification that allows multiple implementations
    * anticipate change
    * e.g., use structures and classes for parameters

Modularity Dimensions

- Modularity has several dimensions and encompasses specification, design, and implementation levels:
  - Criteria for evaluating design methods with respect to modularity
    * Modular Decomposability
    * Modular Composability
    * Modular Understandability
    * Modular Continuity
    * Modular Protection
  - Principles for ensuring modular designs:
    * Language Support for Modular Units
    * Few Interfaces
    * Small Interfaces (Weak Coupling)
    * Explicit Interfaces
    * Information Hiding

Criteria for Evaluating Modular Designs

- 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 Modular Designs (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?
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.

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.

The Open/Closed Principle

- A satisfactory module decomposition technique should yield modules that are both open and closed:
  - **Open Module**: is one still available for extension. This is necessary because the requirements and specifications are rarely completely understood from the system's inception.
  - **Closed Module**: is available for use use by other modules, usually given a well-defined, stable description and packaged in a library. This is necessary because otherwise code sharing becomes unmanageable because reopening a module may trigger changes in many clients.

The Open/Closed Principle (cont’d)

- Traditional design techniques and programming languages do not offer an elegant solution to the problem of producing modules that are both open and closed.
- Object-oriented methods utilize inheritance and dynamic binding to solve this problem.
Case Study: Arithmetic Expression Evaluation

- Initial specification: write a four-function calculator utility that evaluates arithmetic expressions of the form:
  \[(10 + 20) \times 30.0 / 49 - 37\]

- Typical solution in Ada, Pascal, or C makes use of enumerated types and a `case` or `switch` statement, e.g.,
  ```
  enum op_type { ADD, SUB, MUL, DIV }
  
  void apply (op_type op_code, double v1, double v2) {
    switch (op_code) {
      case ADD: // . . .
      case SUB: // . . .
      case MUL: // . . .
      case DIV: // . . .
    }
  }
  ```

However, problems arise when the calculator’s functionality is enhanced by adding new operations, e.g., extending it to handle complete C or C++ expression syntax.

- Programs that utilize case analysis and enumerated types are often difficult to update, enhance, and maintain.

OOD/OOP provide a more flexible and extensible solution using inheritance and dynamic binding.

Virtual Machine Structuring

- A virtual machine architecture is used to decompose system into smaller, more manageable units.

- A virtual machine provides an extended “software instruction set”
  - Extensions provide additional data types and associated “software instructions”
  - Modeled after hardware instruction set primitives that work on a limited set of data types

- Virtual machines allow incremental extensions
  - But beware of overly narrow interfaces in lower layer virtual machines . . .
**Virtual Machine Structuring (cont’d)**

- Good examples of virtual machine concept:
  - Computer Architectures
    - *e.g.*, compiler → assembler → obj code → microcode → gates, transistors, signals, *etc.*
  - Communication protocol stacks, *e.g.*, ISO, TCP/IP
  - Operating systems, *e.g.*, Mach, BSD UNIX, *e.g.*

<table>
<thead>
<tr>
<th>HARDWARE MACHINE</th>
<th>SOFTWARE VIRTUAL MACHINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction set</td>
<td>set of system calls</td>
</tr>
<tr>
<td>restartable instructions</td>
<td>restartable system calls</td>
</tr>
<tr>
<td>interrupts/traps</td>
<td>signals</td>
</tr>
<tr>
<td>interrupt/trap handlers</td>
<td>signal handlers</td>
</tr>
<tr>
<td>blocking interrupts</td>
<td>masking signals</td>
</tr>
<tr>
<td>interrupt stack</td>
<td>signal stack</td>
</tr>
</tbody>
</table>

**Hierarchy**

- Motivation: reduces module interactions by restricting the topology of relationships
- A relation defines a hierarchy if it partitions units into levels (note connection to virtual machines)
  - 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 $\geq i$.
- Hierarchical structure forms basis of design
  - Facilitates independent development
  - Isolates ramifications of change
  - Allows rapid prototyping

**Hierarchy (cont’d)**

- Relations that define hierarchies:
  - Uses
  - *Is-Composed-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.

**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
    - *e.g.*, error logging
  - Some uses don’t involve invocations
    - *e.g.*, message passing, interrupts, shared memory access
- A *uses* relation does not necessarily yield a hierarchy (avoid cycles . . .)
The Uses Relation (cont’d)

- Allow X to use Y when:
  - X is simpler because it uses Y
    * e.g., Standard C library routines
  - Y is not substantially more complex because it is not allowed to use X
    * i.e., hierarchies should be semantically meaningful
  - 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.

The Is-Composed-Of Relation

- The is-composed-of relationship shows how the system is broken down in components.
- X is-composed-of \{x_i\} if X is a group of units x_i that share some common purpose
- The system structure graph description can be specified by the is-composed-of relation such that:
  - non-terminals are “virtual” code
  - terminals are the only units represented by “actual” (concrete) code

The Uses Relation, (cont’d)

- How should recursion be handled?
  - Group X and Y as a single entity in the uses relation.
- A hierarchy in the uses relation is essential for designing non-trivial reusable software systems.
- Note that certain software systems require some form of controlled violation of a uses hierarchy
  - e.g., asynchronous communication protocols, call-back schemes, signal handling, etc.
  - Upcalls are one way to control these non-hierarchical dependencies
- “Rule of thumb”:
  - Start with an invocation hierarchy and eliminate those invocations (“calls”) that are not uses relationships.

The Is-Composed-Of Relation, (cont’d)

- Many programming languages support the is-composed-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)
- Modules and levels need not be identical, as a module may have several components on several levels of a uses hierarchy.
The Is-A and Has-A Relations

- These two relationships are associated with object-oriented design and programming languages that possess inheritance and classes.
- *Is-A* or *Descendant* 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* or *Containment* relationship
  - class X possesses a *Has-A* relationship with class Y if instances of class X contain one or more instance(s) of class Y.
  - *e.g.*, a car has an engine and four tires . . .

Separate Policy and Mechanism

- Very important design principle, used to separate concerns at both the design and implementation phases.
- Multiple policies can be implemented by shared mechanisms.
  - *e.g.*, OS scheduling and virtual memory paging
- Same policy can be implemented by multiple mechanisms.
  - *e.g.*, FIFO containment can be implemented using a stack based on an array, or a linked list, or . . .
  - *e.g.*, reliable, non-duplicated, bytestream service can be provided by multiple communication protocols.

Program Families and Subsets

- Program families are a collection of related modules or subsystems that form a framework
  - *e.g.*, BSD UNIX network protocol subsystem.
  - Note, a framework is a set of *abstract* and *concrete* classes.
- Program families are natural way to detect and implement subsets.
  - Reasons for providing subsets include cost, time, personnel resources, *etc*.
  - Identifying subsets:
    * Analyze requirements to identify minimally useful subsets.
    * Also identify minimal increments to subsets.

Program Families and Subsets (cont’d)

- Advantages of subsetting:
  - Facilitates software system extension and contraction
  - Promotes reusability
  - Anticipates potential changes
- Note: it is most important to design for flexibility, not necessarily for generality . . .
Program Families and Subsets (cont'd)

- Families provide integrated support for
  - 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!

A General Design Process

- Given a specification, design involves an iterative decision making process with the following general steps:
  - List the difficult decisions and decisions likely to change
  - Design a module specification to hide each such decision
    * Make decisions that apply to whole program family first
    * Modularize most likely changes first
  - Then modularize remaining difficult decisions and decisions likely to change
  * Design the uses hierarchy as you do this (include reuse decisions)

A General Design Process (cont'd)

- General steps (cont'd)
  - Treat each higher-level module as a specification and apply above process to each
  - Continue refining until all design decisions are:
    * hidden in a module
    * contain easily comprehensible components
    * provide individual, independent, low-level implementation assignments

Traditional Development Methodologies

- Waterfall Model
  - Specify, analyze, implement, test (in sequence)
  - Assumes that requirements can be specified up front
- Spiral Model
  - Supports iterative development
  - Attempts to assess risks of changes
- Rapid Application Development
  - Build a prototype
  - Ship it :-)

Copyright © 1997-2000 Dept. of Computer Science, Washington University
eXtreme Programming

- Stresses customer satisfaction, and therefore, involvement
  - Provide what the customer wants, as quickly as possible
  - Provide only what the customer wants
- Encourages changes in requirements
- Relies on testing
- XP Practices
  - Planning, designing, coding, testing

---

eXtreme Programming: Planning

- Start with user stories
  - Written by customers, to specify system requirements
  - Minimal detail, typically just a few sentences on a card
  - Expected development time: 1 to 3 weeks each, roughly

Planning game creates commitment schedule for entire project
- Each iteration should take 2-3 weeks

---

eXtreme Programming: Designing

- Defer design decisions as long as possible

Advantages:
  - Simplifies current task (just build what is needed)
  - You don’t need to maintain what you haven’t built
  - Time is on your side: you’re likely to learn something useful by the time you need to decide
  - Tomorrow may never come: if a feature isn’t needed now, it might never be needed

Disadvantages:
  - Future design decisions may require rework of existing implementation
  - Ramp-up time will probably be longer later
    * Therefore, always try to keep designs as simple as possible

---

eXtreme Programming: Coding

- Pair programming
  - Always code with a partner
  - Always test as you code
- Pair programming pays off by supporting good implementation, reducing mistakes, and exposing more than one programmer to the design/implementation
- If any deficiencies in existing implementation are noticed, either fix them or note that they need to be fixed.
**eXtreme Programming: Testing**

- Unit tests are written *before* code.
- Code **must** pass both its unit test and all regression tests before committing.
- In effect, the test suite defines the system requirements.
  - Significant difference from other development approaches.
  - If a bug is found, a test for it **must** be added.
  - If a feature isn’t tested, it can be removed.

**eXtreme Programming: Information Sources**

- [http://www.extremeprogramming.org/](http://www.extremeprogramming.org/)

**Design Guidelines: Motivation**

- Design is the process of organizing structured solutions to tasks from a problem domain.
- This process is carried out in many disciplines, in many ways.
  - There are many similarities and commonalities among design processes.
  - There are also many common design mistakes . . .
- The following pages provide a number of “design rules.”
  - Remember, these rules are simply suggestions on how to better organize your design process, *not* a recipe for success!

**Common Design Mistakes**

- Depth-first design
  - only partially satisfy the requirements
  - experience is best cure for this problem . . .
- Directly refining requirements specification
  - leads to overly constrained, inefficient designs
- Failure to consider potential changes
  - always design for extension and contraction
- Making the design too detailed
  - this overconstrains the implementation
Common Design Mistakes (cont’d)

- Ambiguously stated design
  - misinterpreted at implementation
- Undocumented design decisions
  - designers become essential to implementation
- Inconsistent design
  - results in a non-integratable system, because separately developed modules don’t fit together.

Rules of Design

- Make sure that the problem is well-defined
  - All design criteria, requirements, and constraints, should be enumerated before a design is started.
  - This may require a “spiral model” approach.
- What comes before how
  - *i.e.*, define the service to be performed at every level of abstraction before deciding which structures should be used to realize the services.
- Separate orthogonal concerns
  - Do not connect what is independent.
  - Important at many levels and phases . . .

Rules of Design (cont’d)

- Design external functionality before internal functionality.
  - First consider the solution as a black-box and decide how it should interact with its environment.
  - Then decide how the black-box can be internally organized. Likely it consists of smaller black-boxes that can be refined in a similar fashion.
- Keep it simple.
  - Fancy designs are buggier than simple ones; they are harder to implement, harder to verify, and often less efficient.
  - Problems that appear complex are often just simple problems huddled together.
  - Our job as designers is to identify the simpler problems, separate them, and then solve them individually.

Rules of Design (cont’d)

- Work at multiple levels of abstraction
  - Good designers must be able to move between various levels of abstraction quickly and easily.
- Design for extensibility
  - A good design is “open-ended,” *i.e.*, easily extendible.
  - A good design solves a class of problems rather than a single instance.
  - Do not introduce what is immaterial.
  - Do not restrict what is irrelevant.
- Use rapid prototyping when applicable
  - Before implementing a design, build a high-level prototype and verify that the design criteria are met.
Rules of Design (cont'd)

- Details should depend upon abstractions
  - Abstractions should not depend upon details
  - Principle of Dependency Inversion
- The granule of reuse is the same as the granule of release
  - Only components that are released through a tracking system can be effectively reused
- Classes within a released component should share common closure
  - That is, if one needs to be changed, they all are likely to need to be changed
  - i.e., what affects one, affects all

Rules of Design (cont'd)

- Classes within a released component should be reused together
  - That is, it is impossible to separate the components from each other in order to reuse less than the total
- The dependency structure for released components must be a DAG
  - There can be no cycles
- Dependencies between released components must run in the direction of stability
  - The dependee must be more stable than the depender
- The more stable a released component is, the more it must consist of abstract classes
  - A completely stable component should consist of nothing but abstract classes

Where possible, use proven patterns to solve design problems

- When crossing between two different paradigms, build an interface layer that separates the two
  - Don't pollute one side with the paradigm of the other

Rules of Design (cont'd)

- Software entities (classes, modules, etc) should be open for extension, but closed for modification
  - The Open/Closed principle – Bertrand Meyer
- Derived classes must usable through the base class interface without the need for the user to know the difference
  - The Liskov Substitution Principle
Rules of Design (cont’d)

- *Make it work correctly, then make it work fast*
  - Implement the design, measure its performance, and if necessary, optimize it.

- *Maintain consistency between representations*
  - *e.g.*, check that the final optimized implementation is equivalent to the high-level design that was verified.
  - Also important for documentation . . .

- Don’t skip the preceding rules!
  - Clearly, this is the most frequently violated rule!!! ;-)