Software Design

Software Design Questions
From Earlier Courses

- What exactly is a software design?
- How is a design developed?
- How is a design documented?
- How is a design modified?
- Is that important?
- What techniques exist for doing design?

Software Design Questions

From Earlier Courses

- What design "goodness" criteria are there?
  - What is a good design?
  - What are the benefits of a good design?
  - What is a bad design?
  - What are the consequences of a bad design?
- Remember:
  - Specification is what
  - Design is the start of how

Design Goals

- Qualities of a good design:
  - *Correct*
  - *Complete*
  - *Changeable (for maintenance, evolution)*
  - *Simple*
  - *Efficient (or other performance issues)*
- Correctness:
  - It should lead to a correct implementation
- Completeness:
  - It should do everything
- Other issues: security, safety, availability

Design Goals

- Changeable:
  - It should facilitate change—changes are inevitable
- Simplicity
  - It should be as understandable as possible
- Efficiency
  - It should not waste resources, but:

Better a working slow design than a fast design that does not work

Design Process Goals

- Consider: design as *activity* vs. design as *description*
- Goals for the activity of design
  - Make effective use of specifications
  - Be useful in later development
    - Guide development and maintenance
  - Allow for evaluation of alternative designs
    - Based on what? See earlier slides “Design Goals”

Design Views

- Think about:
  - building architecture: are blueprints enough?
  - highway plans: are site-plans enough?

What do or don't these show?

- Software systems:
  - What do we have that are like blueprints or site-plans?
  - What kind of *view* is this?

Design Views

- In general: Static, physical vs. dynamic
- Possible views include:
  - Deployment (physical) view
    - How are components distributed across HW?
  - Process view
    - Threads, processes, communications
  - Logical or module view
    - modules, subsystems, classes, components (e.g. JavaBeans)
  - Implementation view
    - How turned into files, DLLs, webstart, Web 2.0, etc.?

Levels Of Design

- **Very-High-Level Design**—*System Architecture*
- Examples: Distributed, Concurrent, Event-Driven
- **High-Level Design**:
  - Modularity
  - Internal interfaces (module connections)
  - Data sources (Files, databases)
  - External interfaces (hardware connections)
  - High-level algorithms
- **Low-Level or Detailed Design**:
  - Detailed algorithms and data structures

High-Level Design—Modularity

- Myer, 1978:
  - Modularity is the single attribute of software that allows a program to be intellectually manageable.
- What is modularity?
- What is a module?
- *Informally*:

Modularity means decomposition into subprograms and tasks

Module Quality

- A module: a basic unit ("chunk") of software
  - In C etc.: a function
  - In OO languages, usually a class
- Smaller than a "subsystem"
- What makes a good module?

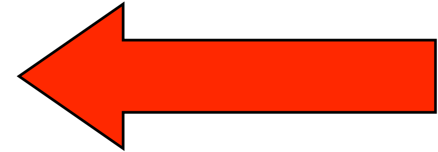
Module Qualities

- Encapsulates just one system task or function
- Reusable
- Testable
  - Some say a module is the smallest chunk of code that can be tested independently
- Hides a design decision from the rest of the system
- Can be modified internally without affecting the rest of the system
  - Interface vs. function in system

High-Level Design—Modularity

- Key questions designers ask:
  - **What should be a separate module or subprogram and why?**
  - **How do modules share information?**
  - **What should modules “know” about other modules or the rest of the system?**
  - **How is control organized in the system?**

- These are *very hard* questions to answer



- And have been for a long time:
  Parnas, 1972: “On the criteria to be used in decomposing systems into modules”

Approaches To Design

Design *Methods*

- Are there techniques for doing design or is it just inspiration?
- There is still a lot of inspiration!

But

- Several techniques exist to help with the selection of:
  - What should be a module
  - Parameters that modules should have
  - Module calling structures
  - Main program structure
  - Some elements of low-level design

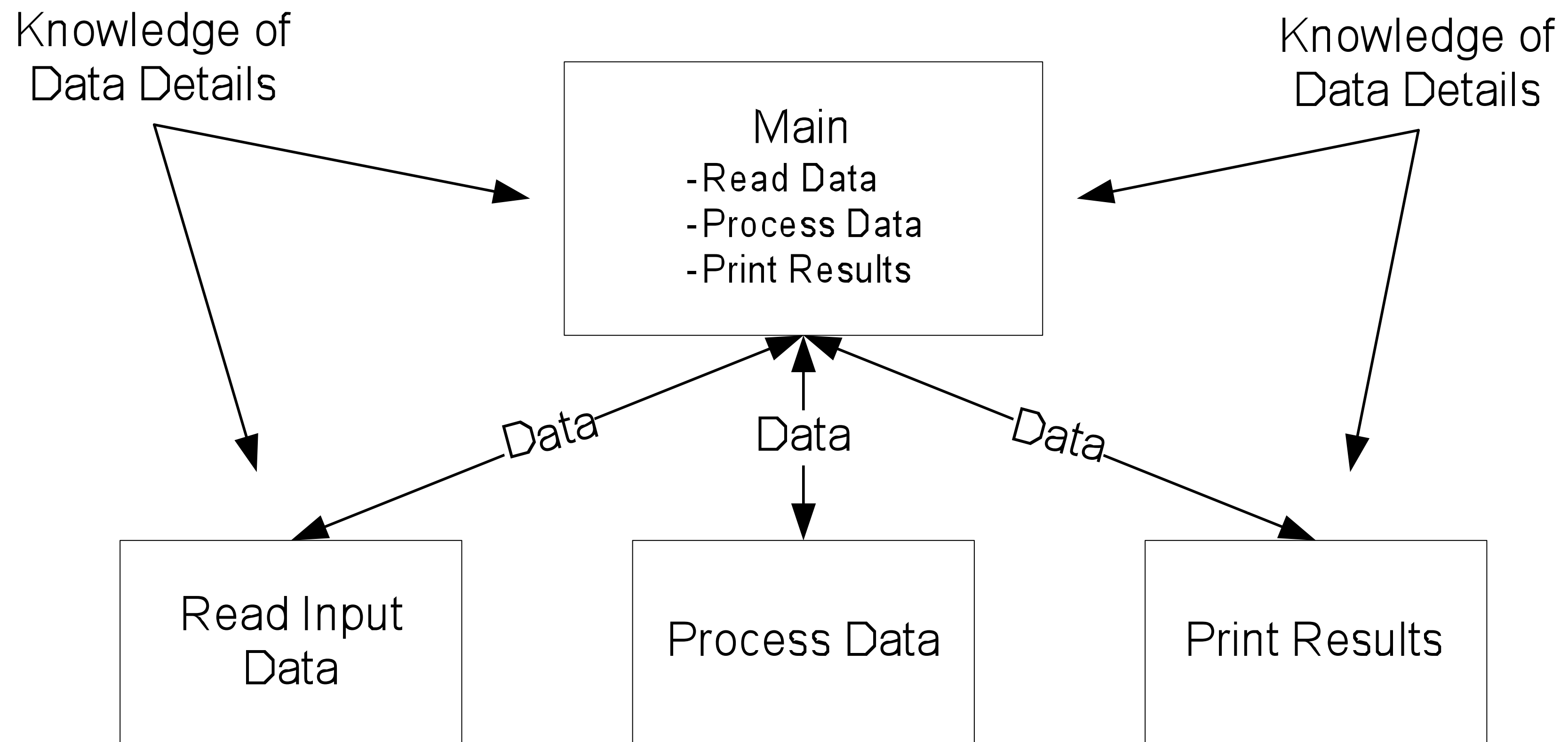
Approaches To Design
Design *Methods*

- Techniques include:
  - Top-down functional decomposition
  - Bottom-up design
  - Structured design
  - Jackson/data structure design
  - Object-oriented design
  - Aspect-oriented design

Top Down *Functional* Decomposition

- Break problem into functional parts
- Parts are usually based on chronological decomposition
  - Do parta, do partb, do partc, etc.
- Make each part a module, call modules in sequence from main program
- Break parts down into smaller parts
- Make each smaller part a module called by the larger module

Top Down *Functional* Decomposition



Top Down *Functional* Decomposition

- Data needed by smaller parts becomes parameters
- Module calling structure referred to as *structure chart*
  - Central to a method called *structured design*
- Problems with this approach:
  - Leads to designs that are difficult to reconcile with principles of information hiding
  - Each module has to “know” details of data structures
  - Data “issues”: scattered across system
- Solution to design does ***not*** lie in “top down functional decomposition”

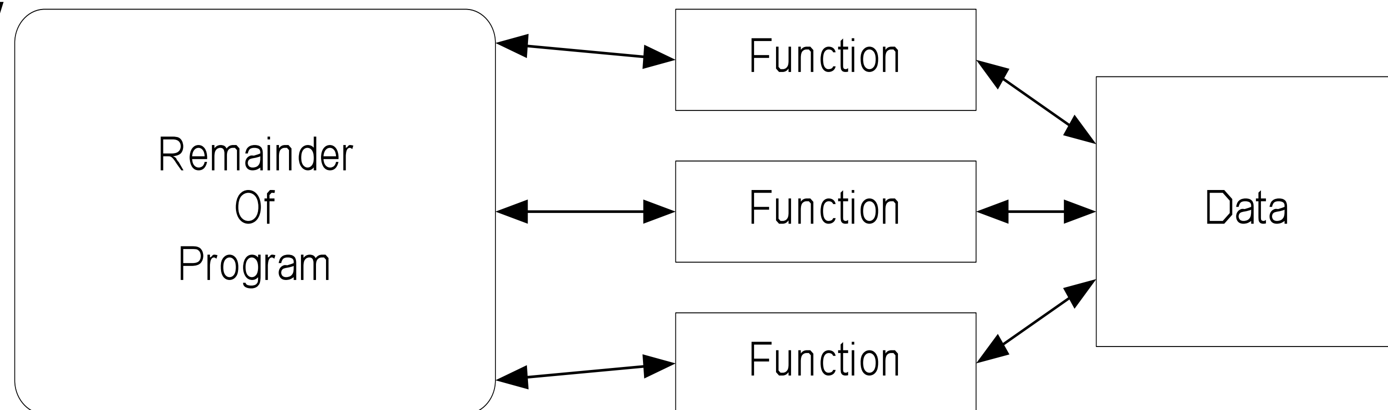
Some Principles of Good Modularity

- Information Hiding
- Abstraction
- Coupling, Cohesion

Information Hiding

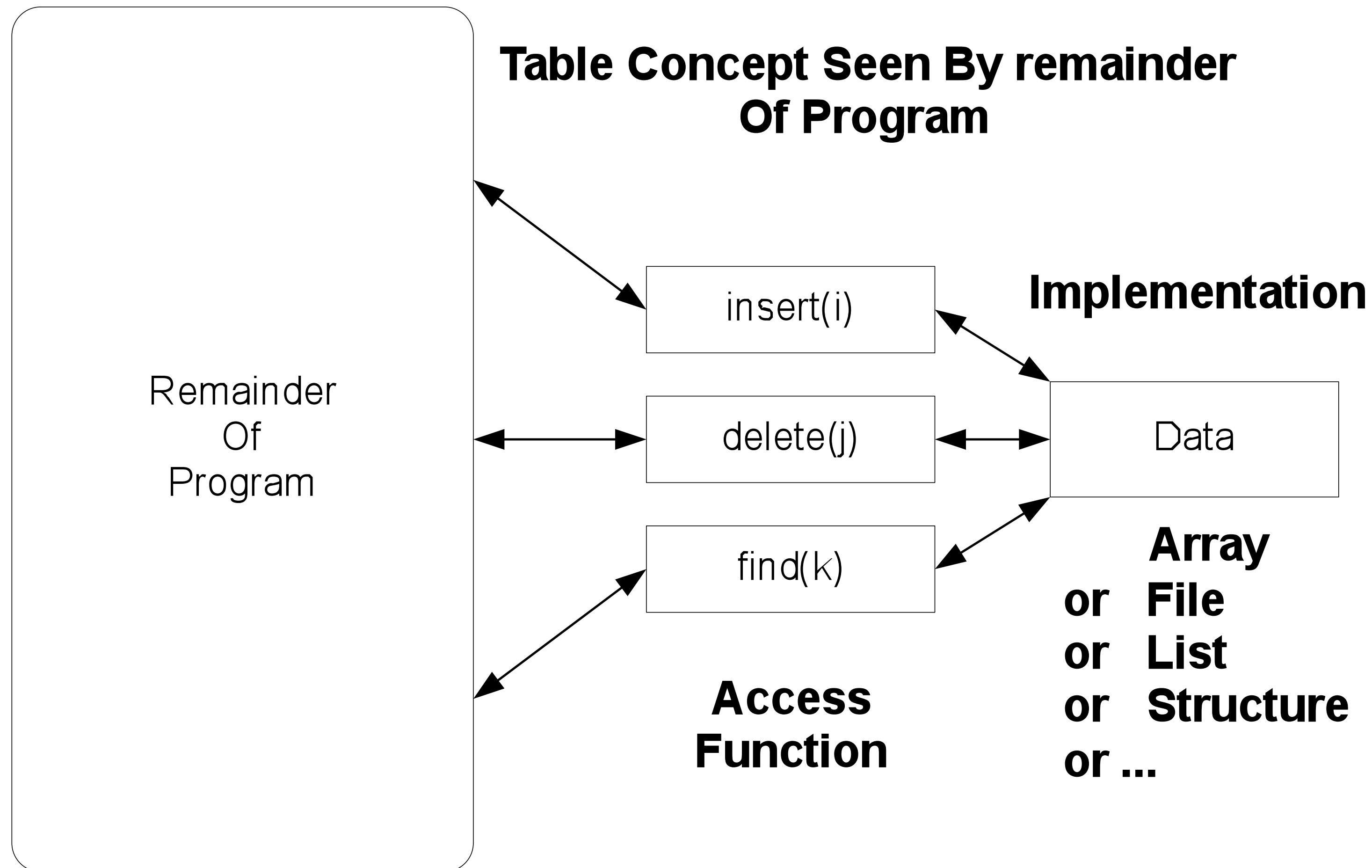
- The concept is actually about hiding ***design decisions***

- Extremely important idea, *impossible to overstate this*
- *Very influential, led to modern notion of object*
- *MUCH harder concept than it appears*
- One view of the Concept*:*



- Justification:
  - Details of data structure hidden
  - Change restricted to one location

Benefit Of Information Hiding



Classes In Programming Languages

- Classes in programming languages give the ability to define new types just as powerful as the built-in types
  - C++, C# and Java are extensible languages
- Do you think that this is the only important way we use classes?
- Classes very often used as a mechanism to support *information hiding*
  - Some say you are *encapsulating* a design decision (or hiding it from the rest of the system)
  - Even better: it's an *abstraction* to the rest of the system

Reminder: Abstraction

- Wikipedia (CS): In CS **abstraction** is a mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time.
  - Allows us to associate and use things that "are the same"
- Data Abstraction
  - An abstraction that hides details of how data or information is stored, represented, etc.
- Procedural Abstraction
  - An abstraction that hides details associated with executing an operation or task

(Reminder) Hiding Details

- Two important viewpoints to keep in mind
  - Abstractions hide low-level or implementation detail hidden from “user”
    - You, as a coder
    - Or, the rest of the system (client-code)
  - Note that giving something a name is an important part of abstraction
    - I can refer to it. We understand it. We agree on it.
    - Often in a programming language, we can write code using the abstraction-name as a type

Abstract Data Types (ADTs)

- Important CS concept:
  Abstract Data Type (ADT)
  - a model of data items stored, and
  - a set of operations that manipulate them
- ADT: no implementation implied
  - A particular data structure implements an ADT
    - Remember: “data structure” defines how it’s implemented. A data structure is **not** abstract.
- Examples: List, Stack, Queue (array, linked-list,…); Tree; Graph; Table/Map

Why Is Abstraction and Hiding Details Important in Design?

Why Is Abstraction and Hiding Details Important in Design?

- Make system flexible, less rigid, easier to change
  - Maintenance, changing requirements
- Easier to understand by development team, maintenance team
- Reuse
- Testability

Design Quality Principles / Metrics

- How can we evaluate a design description of modules and just quality?
- Important principles:
  - Information hiding
  - Cohesion: is everything in this module dedicated to one common goal or purpose
  - Coupling: how complex is a given interaction between two modules?
    - too complex is probably bad
- In structured design, functions are modules
  - interactions are control or data-sharing

Levels of Coupling

Amount of data across interface maters, but also “nature” of sharing of information.

- Uncoupled
- Data coupling
  - Data is passed between components
- Stamp coupling
  - Data structure is passed between components; only part of it used
- Control coupling
  - Data passed affects control flow. Passing a boolean or flag.
- Common coupling
  - Shared data
- Content coupling
  - One module changes internal data of another

Levels of Cohesion

What is "cohesion"? Levels or types of cohesion are:

- Coincidental cohesion
  - Grouped by chance
- Logical cohesion
  - Grouped by function but not related
- Temporal cohesion
  - Grouped by time of use
- Procedural cohesion
  - Grouped to ensure order of use
- Communicational cohesion
  - Grouped because of common I/O source/sink
- Sequential cohesion
  - Output of one module is input to next
- Functional cohesion
  - Information-hiding group

Object-Oriented Design

Perhaps our best hope for successful design?

- Includes the notions of:
  - Classes (data and procedural encapsulation)
  - Composition of objects
  - Abstractions (inheritance, interfaces, polymorphism)
- Critical questions remain:
  - *What should be an object? (in the problem domain)*
  - *What are the right member functions?*
  - *What objects are added when design takes place?*
  - *How is control managed?*

Object-Oriented Design

- Classes define object types
- ***Remember: If you have a class name that implies an activity, there is almost certainly something wrong.***
  - `class read_input` WRONG
  - `class input_sequence` RIGHT
- Member methods / functions:
  - Be complete
  - Have good cohesion and coupling
- Use classes to hide design decisions
- A single large class in a program is probably wrong

OO Design Quality

- Your ideas?
- Some proposed “rules” (presented on the board)
  - More on this in CS441, Prin. of SW Design

Object-Oriented Design Quality

- Basic things, like: Comprehensive interface functionality:
  - All necessary member functions, constructors
  - Appropriate parameters
- Coupling?
- Cohesion?
- The set of classes hierarchic structure?
  - No cycles; breaches of packages/levels, etc.
- Interactions through abstractions vs. concrete things
  - E.g. use of abstract classes or interfaces in Java
    List theList = new ArrayList(); // why? Better example?

Object-Oriented Design Quality

- Attention to reuse:
  - Generality of function where reasonable
    - Between and within classes
  - Generality of detailed design
    - Design patterns
- Comprehensive information hiding:
  - Instance variables and objects
  - Member functions and ***polymorphism***
    - Abstraction in action
- Appropriate algorithms within methods

What's Next?

- High-level design
  - Notations, common patterns
- Object-oriented Design
- Concurrency and Design

Object-Oriented Analysis and Design

- Some history…
- Many methods and notations for OOA and OOD developed since the 1980s:
  - Object-modeling Technique (OMT)
  - Booch method
  - CRC cards
  - Fusion
  - ROOM (Real-Time Object Oriented Modeling)

Object-Oriented Analysis and Design

- Each had different:
  - Notations (diagrams)
  - Activities (process)
- Notations had much in common, which made learning and using new methods more confusing
- Why can't we at least have just one set of notations?
- UML (the *Unified Modeling Language*) tries to be just that