ICS 52: Introduction to Software Engineering
Fall Quarter 2004
Professor Richard N. Taylor
Lecture Notes
Week 4  Design

http://www.ics.uci.edu/~taylor/ICS_52_FQ04/syllabus.html
Today’s Lecture

◆ Architectural design revisited
◆ Modules
◆ Interfaces
Design

- Architectural design
  - High-level partitioning of a software system into separate modules (*components*)
  - Focus on the interactions among parts (*connections*)
  - Focus on structural properties (*architecture*)
    » “How does it all fit together?”

- Module design
  - Detailed design of a component
  - Focus on the internals of a component
  - Focus on computational properties
    » “How does it work?”
Architectural Design

- A simple diagram is not enough
  - It is only a start
- Additional decisions need to be made
  - Define the primary purpose of each component
  - Define the interface of each component
    » Primary methods of access/use
    » As complete as possible
- **Always** requires multiple iterations
  - Cannot do it right in one shot
  - Use the fundamental principles
A Good Design...

…is half the implementation effort (at least)!

- **Rigor** ensures all requirements are addressed
- **Separation of concerns**
  - Modularity allows work in isolation because components are independent of each other
  - Abstraction allows work in isolation because interfaces guarantee that components will work together
- **Anticipation of change** allows changes to be absorbed seamlessly
- **Generality** allows components to be reused throughout the system
- **Incrementality** allows the software to be developed with intermediate working results
A Bad Design...

• …will never be implemented!
  – Lack of rigor leads to missing functionality
  – Separation of concerns
    » Lack of modularity leads to conflicts among developers
    » Lack of abstraction leads to massive integration problems (and headaches)
  – Lack of anticipation of change leads to redesigns and reimplementations
  – Lack of generality leads to “code bloat”
  – Lack of incrementality leads to a big-bang approach that is likely to “bomb”
Design Interaction

Architectural design (previous lecture)

Module design (this lecture)
From Architecture to Modules

- Repeat the design process
  - Design the internal architecture of a component
    » (Break it apart into several modules, and articulate their interconnections and dependencies)
  - Define the purpose of each module
  - Define the provided interface of each module
  - Define the required interface of each module
- Do this over and over again
  - Until each module has…
    » …a simple, well-defined internal architecture
    » …a simple, well-defined purpose
    » …a simple, well-defined provided interface
    » …a simple, well-defined required interface
- Until all modules “hook up”
But What About Those Interfaces?
Interfaces

- Abstraction of the functionality of a component
  - Defines the set of services that a component provides or requires
  - Other components use or supply these services
  - Components themselves implement the services
    » Perhaps with the help of other components
- Serves as a contract
  - Other components rely on the contract
  - Any change can have far-reaching consequences

*Interfaces are the key to proper design*
Example: Network Protocols (1)

- boolean sendSmallPacket(Packet p)
- boolean sendReliableSmallPacket(Packet p)
- boolean sendReliableBigPacket(Packet p)
- Result callRemoteFunction(Function f)

Diagram:

- Provided Interface
  - RPC
  - Required Interface
- Provided Interface
  - Big & Reliable
  - Required Interface
- Provided Interface
  - Reliable
  - Required Interface
- Provided Interface
  - Small
  - Required Interface
Example: Network Protocols (2)

- boolean receiveSmallPacket(Packet)
- boolean receiveReliableSmallPacket(Packet)
- boolean receiveReliableBigPacket(Packet)
- Result RemoteFunction(Function f)
Example: Stock Market

- Provided Interface
- Required Interface

Event receiveEvent()

Broker

Broker

Broker

NASDAQ

Provided Interface

Required Interface

Provided Interface

Required Interface

Provided Interface

Required Interface

Provided Interface

Required Interface

Provided Interface

Required Interface

boolean BroadcastEvent(Event e)

void registerInterest(EventType et)
Interfaces and Fundamental Principles

- Interfaces are rigorously and formally defined
- Interfaces separate concerns
  - Interfaces modularize a system
  - Interfaces abstract implementation details
    » With respect to what is provided
    » With respect to what is required
- (Good) Interfaces anticipate change
Tools of the Trade

- Apply information hiding
  - “Secrets should be kept from other modules”
  - Abstract data types
- Use requirements specification
  - Objects, entities, relationships, algorithms
- Determine usage patterns
- Anticipate change
- Design for generality and incrementality
  - Reuse
- Design for program families
Apply Information Hiding

✦ One module “hides secret information” from other modules
  – Data representations
  – Algorithms
  – Sequencing of processing
✦ Why?
  – To create a clean separation of concerns
Abstract Data Types

- Goal: Encapsulate the concrete representation of a data structure with all functions that access the representation
- Users see only the abstract characteristics of the structure
- Access to the structure is only through the provided access functions
- No extraneous functions included

Notes
- Abstract does not mean "vague"
- Abstract does not mean highly mathematical
- Abstract means conceived apart from special cases or instances
- Abstract implies a many-to-one mapping that models some aspects of an entity, but not all
Specification and Implementation of ADTs

- Specification of an Abstract Data Type
  - Domain: the types(s) of the functions
    » one domain/type is being defined; the others are assumed to be known
    » objects may have structure, but aspects of the structure are only observable as functions are applied
  - Access Functions (semantics)
    » Primitive constructors
    » Combinational constructors
    » Query functions
  - Exceptions

- Implementation of ADTs
  - Internal objects
  - Internal functions
  - Internal errors and error handling

- Examples: Stacks and queues; date packages
package rational_numbers is
  type rational is limited private;
  function "=" (x,y: rational) return boolean;
  function "+" (x,y: rational) return rational;
  function "-" (x,y: rational) return rational;
  function "*" (x,y: rational) return rational;
  function "/" (x,y: rational) return rational;
  function "/" (x,y: integer) return rational;
  procedure assign (x: out rational; y: rational);
private
  zero_denominator: exception;
end;

Rational Numbers: Use

with rational_numbers; declare
   use rational_numbers;
   x, y, z: rational;
begin
   assign (x,3/4);
   assign (y, 6/8);
   if x=y then put ("equal");
       else put ("not equal");
   end if;
   assign (z, x*y);
end;
Rational Numbers: Implementation

private
type rational is
  record
    numerator: integer;
    denominator: integer range 1..integer'last;
  end record;
package body rational_numbers is
  procedure same_denominator (x,y: in out rational) is
    begin
      -- changes x and y to have the same denominator
    end;
  function "=" (x,y: rational) return boolean is
    u,v: rational:
    begin
      u := x;
      v := y;
      same_denominator (u,v);
      return (u.numerator = v.numerator);
    end "=";
  function "/" (x,y: integer) return rational is
    begin
      return (x,y);
    end "/";
    -- you can guess what +, -, * look like
    -- and of course the other "/" must be defined
end rational_numbers;
Use Requirements Specification

- A requirements specification contains lots of useful information to be leveraged during design
  - Nouns: modules / classes (SOMETIMES!)
  - Verbs: methods (SOMETIMES!)
  - Adjectives: properties/attributes/member variables (SOMETIMES!)
- Why?
  - To identify likely design elements
Determine Usage Patterns

- Usage patterns are incredible sources of information
  - Common tasks often can be placed into a single interface method
    » Specific combinations of method invocations
    » Specific iterations over a single method
  - Some usage patterns require non-existing functions
- Why?
  - To refine the interface of a module
Anticipate Change

- Wrap items likely to change within modules
  - Design decisions
  - Data representations
  - Algorithms
- Design module interfaces to be insensitive to change
  - The changeable items go into the module itself
- Why?
  - To limit the effects of (un)anticipated system modifications
Design for Generality/Incrementality

- Design a module to be usable in more than one context
  - Generalize the applicability of methods
    - Do not just draw red squares
    - Do not just stack integers
  - Allow for the addition of extra methods
- Why?
  - To increase reuse
A system is typically used in more than one setting

- Different countries
  - Different languages
  - Different customs
  - Different currencies
- Different hardware/software platforms

Why?
- To enhance applicability
- To keep your company in the black!

Special case of generality and incrementality at the system level
From Architecture to Modules

- Repeat the design process
  - Design the internal architecture of a component
  - Define the purpose of each module
  - Define the provided interface of each module
  - Define the required interface of each module

- Do this over and over again
  - Until each module has...
    » ...a simple, well-defined internal architecture
    » ...a simple, well-defined purpose
    » ...a simple, well-defined provided interface
    » ...a simple, well-defined required interface

- Until all interfaces “hook up”
Good Examples of Modules

- Java 1.3 collection classes
- Standard template library for C++
Next Topics

- USES relation
- IS-COMPOSED-OF relation
- COMPRIZES diagram
- USES diagram
- [Stepwise refinement]
In Design, We Can Do Anything...

- Big Component
  - Provided Interface
  - Required Interface
- Tiny Component
  - Provided Interface
  - Required Interface
- A Component
  - Provided Interface
  - Required Interface
- B Component
  - Provided Interface
  - Required Interface
- Mr. Component
  - Provided Interface
  - Required Interface
- Mrs. Component
  - Provided Interface
  - Required Interface
- One Component
  - Provided Interface
  - Required Interface
- Some Component
  - Provided Interface
  - Required Interface
- Yet Component
  - Provided Interface
  - Required Interface
...Even when Restricted by Style

What happened here?
Fan-in and Fan-out

High fan-in

Low fan-out

Low fan-in

High fan-out

USUALLY GOOD!

USUALLY BAD!
The Uses Relation

◆ A useful concept for examining a set of modules w.r.t. flexibility, reuse, and incremental testability

◆ Definition: $M_i$ uses $M_j$ if and only if correct execution of $M_j$ is necessary for $M_i$ to complete the task described in its specification.

◆ Note: uses is not the same as invokes:
  – Some invocations are not uses
    » they are just transfers of control
  – Some uses don't involve invocations
    » interrupt handlers
    » shared memory (gag!)
USES Relation

- **Definition**
  - Level 0: those modules that do not use any other modules
  - Level i: those modules that use at least one module at level i – 1 and use no modules at level i or greater

- **Use**
  - Determine flexibility
  - Determine reuse
  - Determine incremental testability
Example
Observations

◆ The USES relation does not necessarily form a hierarchy
  – An acyclic directed graph is good
  – Cycles generally are bad
    » Indication of high coupling
    » Indication of broken separation of concerns
◆ Rules of thumb: allow $a$ to use $b$…
  – …if it makes $a$ simpler
  – …if $b$ is not only used by $a$ but also by other components
Observations

- Some invocations are *not* USES
  - Consider a transfer of control
  - Consider a scheduler inside a program
- Some USES do *not* involve invocations
  - Consider interrupt handlers
  - Consider global variables
  - Consider a blackboard
IS-COMPONENT-OF Relation

- **Definition**
  - Module $M_i$ IS-COMPONENT-OF module $M$ if $M$ is realized by aggregating several modules, one of which is $M_i$
  - The combined set of all modules that exhibit the IS-COMPONENT-OF relation with respect to module $M$ are said to implement module $M$

- **Use**
  - Determine hierarchical decomposition of a component in its subcomponents
  - Abstract details
Example

Compressor IS-COMPONENT-OF Audio Encoder
Encoder IS-COMPONENT-OF Audio Encoder
Reader IS-COMPONENT-OF Audio Encoder
Compressor, Encoder, and Reader IMPLEMENT Audio Encoder
Audio Encoder IS-COMPOSED-OF Compressor, Encoder, and Reader
Comprises Diagram

- Bla Component
  - Duh Component
    - Big Component
      - A Component
        - Provided Interface
        - Required Interface
    - Required Interface
  - Provided Interface
  - Required Interface

- Doh Component
  - Provided Interface
  - Tiny Component
    - Required Interface
  - Provided Interface
  - B Component
    - Required Interface
  - Provided Interface
  - Mr. Component
    - Required Interface
  - Provided Interface
  - Required Interface
USES Diagram – Step 1

- Provided Interface
- Required Interface

Duh Component
- Provided Interface
- Big Component
- Required Interface
- Provided Interface
- A Component
- Required Interface
- Required Interface

Yet Component
- Provided Interface

Doh Component
- Provided Interface
- Tiny Component
- Required Interface
- Provided Interface
- B Component
- Required Interface
- Provided Interface
- Mr. Component
- Required Interface
- Required Interface
USES Diagram – Step 2

- **Big Component**
  - Provided Interface
  - Required Interface

- **A Component**
  - Provided Interface
  - Required Interface

- **Yet Component**
  - Provided Interface
  - Required Interface

- **Tiny Component**
  - Provided Interface
  - Required Interface

- **B Component**
  - Provided Interface
  - Required Interface

- **Mr. Component**
  - Provided Interface
  - Required Interface
USES Diagram – Step 3

This is tricky!
USES Diagram – Step 4

- Big Component
  - Provided Interface
  - Required Interface
- A Component
  - Provided Interface
  - Required Interface
- Yet Component
  - Provided Interface
  - Required Interface
- Tiny Component
  - Provided Interface
  - Required Interface
- B Component
  - Provided Interface
  - Required Interface
- Mr. Component
  - Provided Interface
  - Required Interface
Observations

- Why do we identify higher-level modules in the first place?
  - Understanding
  - Abstraction through composition

- IS-COMPONENT-OF is not
  - is-attribute-of
  - is-inside-of-on-the-screen
  - is-subclass-of
  - is-accessed-through-the-menu-of
The Design Process

- Repeat the design process
  - Design the internal architecture of a component
  - Define the purpose of each module
  - Define the provided interface of each module
  - Define the required interface of each module

- Do this over and over again
  - Until each module has...
    » ...a simple, well-defined internal architecture
    » ...a simple, well-defined purpose
    » ...a simple, well-defined provided interface
    » ...a simple, well-defined required interface

- Until all modules “hook up”
Techniques to Use

◆ Tools of the trade
  – Apply information hiding
  – Use requirements specification
  – Determine usage patterns
  – Anticipate change
  – Design for generality/incrementality
  – Design for program families

◆ Strive for
  – Low coupling/high cohesion
  – A clean IS-COMPOSED-OF structure
  – A clean USES structure
Low-Coupling/High-Cohesion

- Cohesion measures the rate of interconnectedness within a module.
- Coupling measures the rate of interconnectedness among modules.

- Shows critical issues:
  - a rate, rather than an absolute number (we like percentages)
  - what it measures: interconnectedness (how well it all hangs together)
  - within or among a module