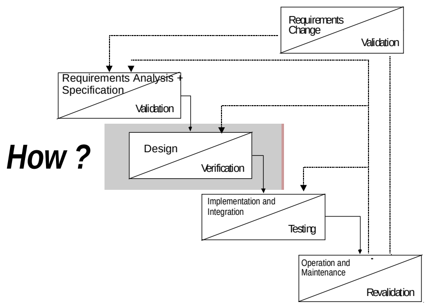
ICS 121 Lecture Notes

Basic Design Concepts



Relation to Other Phases

- Requirements Specification
  - Specifies what should be accomplished, but not how
  - But how do you avoid design bias?
    - requirements hierarchy
    - user model
  - User interfaces partially specified, but further decisions may be made during design
- Implementation
  - Design stops and coding begins when design specifications are sufficient for coding assignments.
    - can be given to programmers unaware of the overall architecture

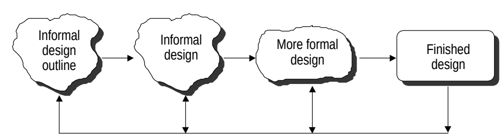
Define the components and the interfaces between them

Goals and Objectives

- Develop a coherent representation of software that will satisfy the requirements
- (Identify inadequacies in the requirements)
- Develop a review plan that, when carried out, will yield confidence in the design
- Develop a validation test plan for determining if implementation meets design

The Design Process

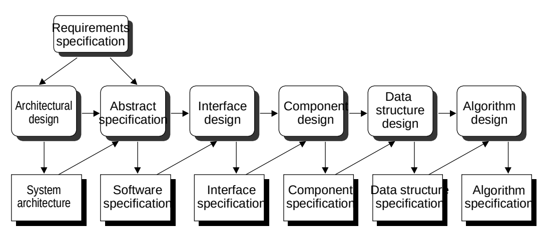
- Design process developes several models of the system at different levels of abstraction
  - Starting point is an informal design
  - Adding information to make it consistent and complete
  - Feedback to earlier design models to be improved



Design Activities

- Architectural design
  - Identification of the sub-systems and abstract specification of their services and their constraints
- Abstract specification
  - For each sub-system, an abstract specification is produced
- Interface design
  - For each sub-system, its interface with other sub-systems is designed and documented (e.g. using formal methods)
- Component design
  - Services are allocated to different components and the interfaces of these components are designed
- Data structure design
  - The data structures used in the system implementation are designed in detail and specified
- Algorithm design
  - The algorithms used to provide services are designed in detail and specified

Design Activities - 2



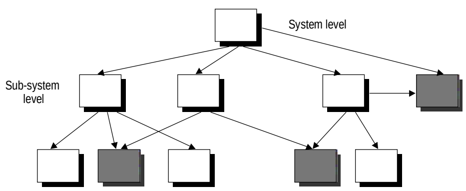
Topic 7 Design

Topic 7 Design 7

Top-Down Design

ICS 121

- **One way of tackling a design problem:**
  - Recursively partitioning the problem into sub-problems until tractable sub-problems are identified
  - Valid approach especially where design components are tightly coupled



Topic 7 Design 8

Products

ICS 121

- **Refined requirements specification**
- **Documentation of decisions and rationale**
- **Data dictionary of all defined objects**
- **Description of program to be constructed**
  - software architecture described as a decomposition diagram
  - abstract module interface specifications
  - internal module designs: data and algorithm descriptions
- **Integration test plan**

Topic 7 Design 9

Desirable Characteristics

ICS 121

- **Uniform and complete**
- **Rigorous and confirmable**
- **Supportable by tools**
- **Desensitized to change**
- **Accommodates independent development**

Topic 7 Design 10

Common Problems

ICS 121

- **Depth-first design: only partial satisfaction of requirements**
- **Failure to consider potential changes**
- **Too detailed: overconstrains implementation**
- **Ambiguous: misinterpreted during implementation**
- **Undocumented: designers become essential**
- **Inconsistent: system cannot be integrated**

Topic 7 Design 11

Abstraction

ICS 121

- **Abstraction is a primary guiding design principle**
- **Intellectual tool that allows us to focus on important, inherent properties and suppress unnecessary detail**
- **Permits separation of conceptual aspects of system from the implementation details**
- **Provides a *model* of behavior**
- **Allows postponement of design decisions**
  - external/functional
  - structural/architectural
  - representational/algorithmic

Topic 7 Design 12

Abstraction - 2

ICS 121

- **Three basic abstraction mechanisms:**
  - procedural abstraction
    - specification describes input/output
    - implementation describes algorithm
    - types: structural
  - data abstraction
    - specification describes attributes, values, properties, operations
    - implementation describes representation and implementation
    - types: compound data structure, abstract data type
  - control abstraction
    - specification describes desired effect
    - implementation describes mechanism
    - types: selection, repetition

Information Hiding

- Each design unit *hides* internal details of processing activities
- Design units communicate only through *well-defined* interfaces (as opposed, e.g. to global variables)
- Each design unit is specified by as little information *as possible*
- If internal details change, client units *should* need no change
- Sample things to modularize and encapsulate
  - Abstract data types
  - Algorithms (e.g., sorting)
  - Input and output formats
  - Processing sequence
  - Machine dependencies (e.g., character codes)
  - Policies (e.g. when and how to do garbage collection)

Cohesion and Coupling

- Cohesion: a design unit has high cohesion if all its elements are strongly related
  - coincidental: multiple, completely unrelated actions
  - logical: series of related actions selected by parameter
  - temporal: series of actions related in time
  - procedural: series of actions sharing sequence of steps
  - communicational: procedural cohesion but on the same data
  - informational: series of independent actions on the same data
  - functional: exactly one action
- Coupling: a decomposition has low coupling if the design units are not strongly dependent on each other
  - content: one directly references content of another
  - common: both have access to same global data
  - control: one passes an element of control to another
  - stamp: one passes a data structure to another, which only uses part
  - data: one passes only homogeneous data items

Modules vs. Sub-systems

- Sub-system
  - Is a system in its own right whose operation does not depend on the services provided by other sub-systems
  - Are composed of modules
  - Have defined interfaces which are used for communication with other sub-systems
- Module
  - Is a system component that provides one or more services to other modules
  - It is not normally considered to be an independent system
- Both
  - Encapsulate the representation of an abstraction
  - Hide a design decision, unnecessary details, a secret, implemention

Module Interfaces

- Imports (Uses)
  - services the module requests from other modules
- Exports (Public)
  - services the module provides to other modules
  - what should be known externally
- Discriminatory use
  - not all uses will require (or be granted) the same services
- Negotiating interface specifications
  - abstraction implies one specification with many possible implementations
  - determine potential services for all possible uses
  - determine likely usage patterns and purposes
  - determine feasibility
  - anticipate potential changes

What is a Module ?

- Various attempts to define this term, e.g.
  - [Stevens/Myers/Constantine, 1974]

    "A set of one or more contiguous program statements having a name by which other parts of the system can invoke it, and preferably having its own distinct set of variable names."

    Problems:
    - assemler macros, header files (e.g., in C, C++), Ada packages, etc. not included in this definition
  - [Yourdon/Constanine, 1979]

    "A module is a lexically contiguous sequence of program statements, bounded by boundary elements (e.g. `begin...end`, `{...}`), having an aggregate identifier"

- Discussion in textbook p. 139 ff.

A sample Module description

Sample module description using a textual design notation:

module X
uses Y imports (B,C)
selective import of B and C from module Y
exports var A:integer;
type B: array (1..10) of real;
procedure C(.....);
optional natural language description of what A, B, and C actually are, possible constraints, etc. that clients need to know.
implementation
if needed, here are general comments about the rationale of the modularization, hints on the implementation, etc.
end X

ICS 121 Lecture Notes

Levels of Cohesion

- Functional cohesion
- Informational cohesion
- Communicational cohesion
- Procedural cohesion
- Temporal cohesion
- Logical cohesion
- Coincidental cohesion

(Good) ↕ *(Bad)*

Levels of Coupling

- Data coupling
- Stamp coupling
- Control coupling
- Common coupling
- Content coupling

(Good) ↕ *(Bad)*

Modules in the Lifecycle

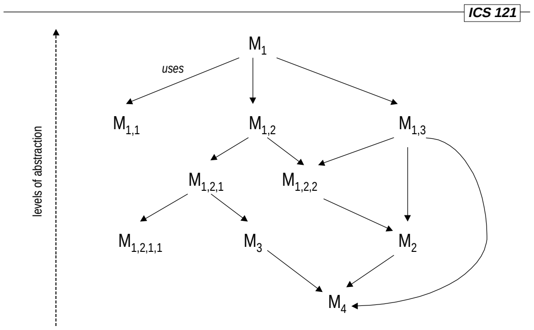
- **Problem Definition (Scenarios & Mockup's)**
  - Task Description
- **Requirements**
  - UML Class Diagram
- **Implementation**
  - Function, Subroutine, (Class, Method, Macro)
- **Maintenance**
  - Test Case
- **Operation**
  - End User Macro, Script

Hierarchy: *Uses*

- **Definition: *a uses b* if there exist situations in which the correct functioning of *a* depends on the ability of a correct implementation of *b***
  - Allow *a* to use *b* when:
    - *a* is simpler because it uses *b*
    - *b* is not substantially more complex because it is not allowed to use *a*
    - there is a useful subset containing *b* and not *a*
    - there is no conceivably useful subset containing *a* but not *b*
    - What do you do with recursion? Group *a* and *b* as a single entity in the uses relation
- **System structure can be specified by the uses relation**
  - Level 0 is the set of all programs that use no other program
  - Level *i* (*i* > 0) is the set of all programs that use at least one program on level *i* -1 and no program at level • *i.*
- **The uses relation should be acyclic**

Hierarchy: *Uses*

28a

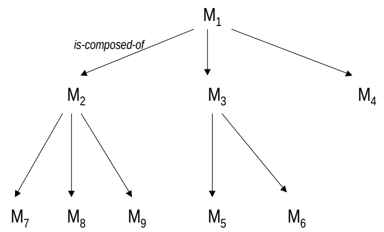


Hierarchy: *Is-Composed-Of*

- **Definition: *a is-composed-of b* if *b* is a component of *a* and encapsulated within it**
- **System structure can be specified by the *is-composed-of* relation where**
  - non-terminals are virtual code
  - terminals are the only units represented by code
- **In such a case the *uses* relation is specified over the set of terminals only**
- **The *is-composed-of* relation should be acyclic**

Hierarchy: *Is-Composed-Of*

29a



Integration Test Plan

- **Developed as part of (architectural) design**
- **Test plan to exercise module interactions**
  - actual test data and expected results for each potential module interaction
  - order of test executions
  - completion criteria
    - simple coverage (one test per interaction)
    - input/output coverage (range of values)
    - data flows (flows from user to used and back)
- **Basic goal is to test how modules interact with each other and with data under the assumption that they have passed module testing**

Integration Testing

- **Testing based on integration test plan after module testing**
- **Integration is the process by which modules are aggregated to create larger components**
- **Integration may be determined by *uses* hierarchy**
- **Integration testing examines each combination to determine whether it is also correct or to find defects in the interaction between "correct" components**

Ensures modules make compatible assumptions

Integration Test Plan Process

- For a given module interaction:
  - Design test cases to test that interaction
    - design typical test case
    - design test cases specific to this interaction
    - design special and boundary value test cases
  - For each test case, provide the "values" of parameters and any environment (e.g., persistent data) required
  - Plan the order of the test cases for this interaction
    - initialize, set-up, process, wrap-up
  - Describe any stubs or drivers required for this interaction
- Plan the order of integration testing
  - top-down
  - bottom-up
  - combination