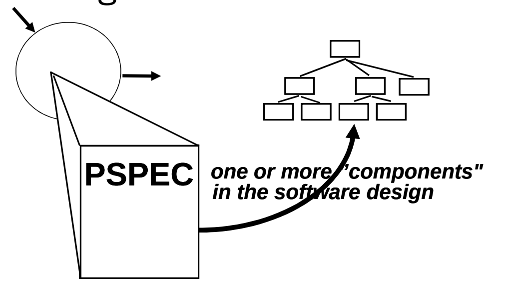
SOFTWARE DESIGN

Oleh :
Ir. I Gede Made Karma, MT

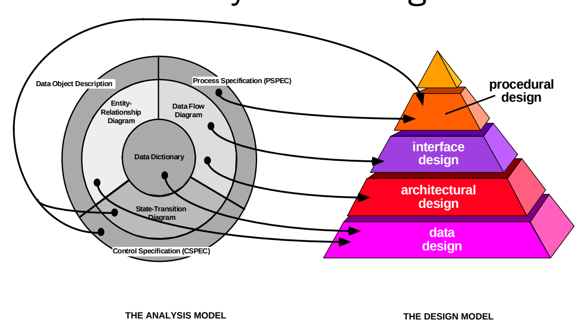
Software Design

- Design Concept and Principles
- Structured Design
- OO Design

A Design Note



Analysis to Design



Design Process

- An iterative process through which requirements are translated into a "blueprint" for constructing the S/W
- Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs
- Guide for evaluation of a good design:
  - The design must implement all of the explicit and implicit requirements
  - The design must be readable
  - The design should provide a complete picture of the software

Evolution of S/W Design

- Development of modular program
- Structural programming
  - Procedural aspect of design definition
- Translation of data flow or data structure into a design definition
- OO design

Design Principles

- The design process should not suffer from "tunnel vision" → should consider alternative approachs
- The design should be traceable to the analysis model
- The design should not reinvent the wheel → use design patterns
- The design should "minimize the intellectual distance" between the S/W and the problem as it exist in the real world
- The design should exhibit uniformity and integration

Design Principles (cont.)

- The design should be structured to accommodate change
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered
- Design is not coding, coding is not design
- The design should be assessed for quality as it is being created, not after the fact
- The design should be reviewed to minimize conceptual (semantic) error

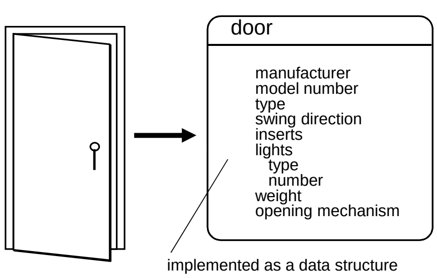
Fundamental Concepts

- **Abstraction** - allows designers to focus on solving a problem without being concerned about irrelevant lower level details (procedural abstraction - named sequence of events, data abstraction - named collection of data objects)
- **Refinement** - process of elaboration where the designer provides successively more detail for each design component
- **Modularity** - the degree to which software can be understood by examining its components independently of one another
- **Software architecture** - overall structure of the software components and the ways in which that structure provides conceptual integrity for a system

Fundamental Concepts (2)

- **Control hierarchy or program structure** - represents the module organization and implies a control hierarchy, but does not represent the procedural aspects of the software (e.g. event sequences)
- **Structural partitioning** - horizontal partitioning defines three partitions (input, data transformations, and output); vertical partitioning (factoring) distributes control in a top-down manner (control decisions in top level modules and processing work in the lower level modules)
- **Data structure** - representation of the logical relationship among individual data elements (requires at least as much attention as algorithm design)
- **Software procedure** - precise specification of processing (event sequences, decision points, repetitive operations, data organization/structure)
- **Information hiding** - information (data and procedure) contained within a module is inaccessible to modules that have no need for such information

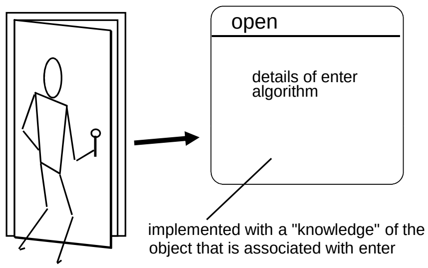
Data Abstraction



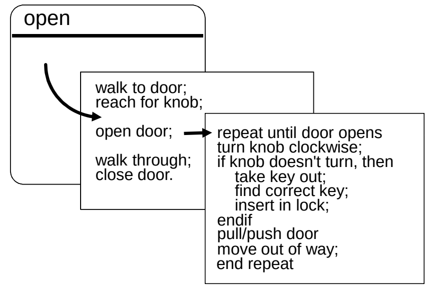
Data Design

- refine data objects and develop a set of data abstractions
- implement data object attributes as one or more data structures
- review data structures to ensure that appropriate relationships have been established
- simplify data structures as required

Procedural Abstraction

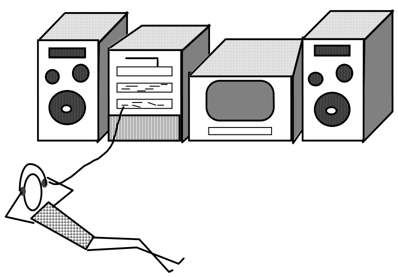


Stepwise Refinement



Modular Design

easier to build, easier to change, easier to fix .



Functional Independence

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

Cohesion and Coupling Spectrum

low	low
Coincidental	No direct coupling
Logical	Data coupling
Temporal	Stamp coupling
Procedural	Control coupling
Communicational	External
Sequential	Common coupling
Functional	Content coupling
high	high

Why Information Hiding?

- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

Why Architecture?

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

(1) analyze the effectiveness of the design in meeting its stated requirements,

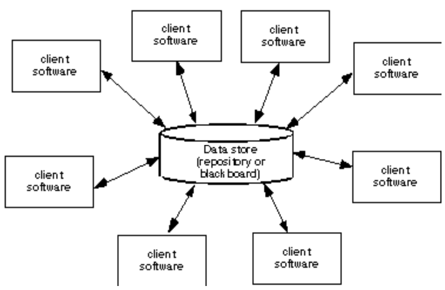
(2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the risks associated with the construction of the software.

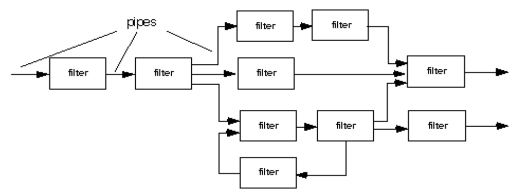
Architectural Styles

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

Data-Centered Architecture



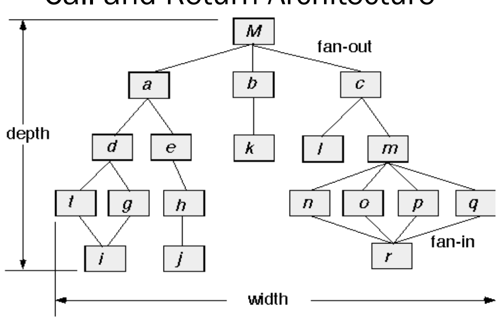
Data Flow Architecture



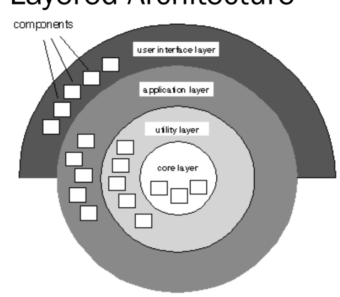
(a) pipes and filters

(b) batch sequential

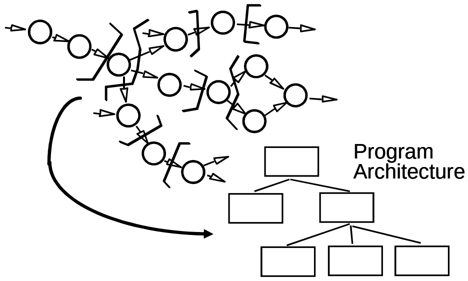
Call and Return Architecture



Layered Architecture

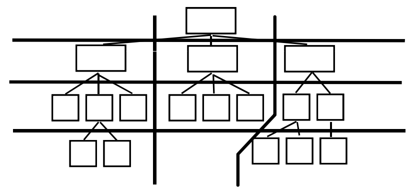


Deriving Program Architecture



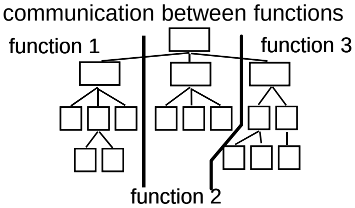
Partitioning the Architecture

- "horizontal" and "vertical" partitioning are required



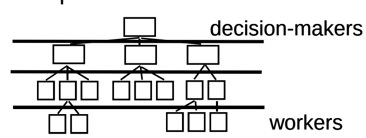
Horizontal Partitioning

- define separate branches of the module hierarchy for each major function
- use control modules to coordinate communication between functions



Vertical Partitioning: Factoring

- design so that decision making and work are stratified
- decision making modules should reside at the top of the architecture



Why Partitioned Architecture?

- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend

Design Heuristics for Effective Modularity

- Evaluate the first iteration of the program structure to reduce coupling and improve cohesion.
- Attempt to minimize structures with high fan-out; strive for fan-in as structure depth increases.
- Keep the scope of effect of a module within the scope of control for that module.
- Evaluate module interfaces to reduce complexity, reduce redundancy, and improve consistency.
- Define modules whose function is predictable and not overly restrictive (e.g. a module that only implements a single subfunction).
- Strive for controlled entry modules, avoid pathological connection (e.g. branches into the middle of another module)

Structured Design

- objective: to derive a program architecture that is partitioned
- approach:
  - the DFD is mapped into a program architecture
  - the PSPEC and STD are used to indicate the content of each module
- notation: structure chart

Structured Design (2)

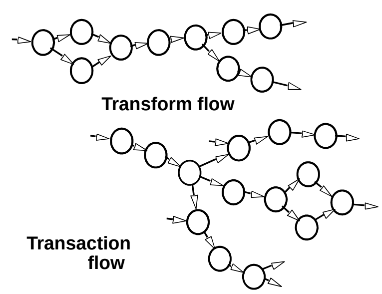
- Architectural design
- Interface design
- Data design
- Procedural design/component-level design

Architectural Design

Mapping Requirements to Software Architecture

- Establish type of information flow (transform flow - overall data flow is sequential and flows along a small number of straight line paths; transaction flow - a single data item triggers information flow along one of many paths)
- Flow boundaries indicated
- DFD is mapped into program structure
- Control hierarchy defined
- Resultant structure refined using design measures and heuristics
- Architectural description refined and elaborated

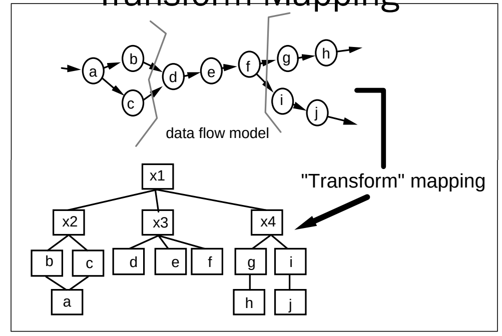
Flow Characteristics



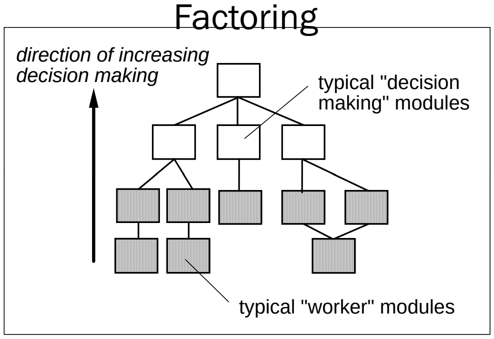
General Mapping Approach

- isolate incoming and outgoing flow boundaries; for transaction flows, isolate the transaction center
- working from the boundary outward, map DFD transforms into corresponding modules
- add control modules as required
- refine the resultant program structure using effective modularity concepts

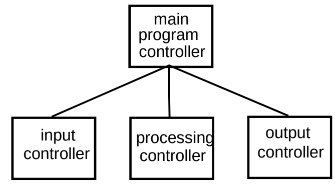
Transform Mapping



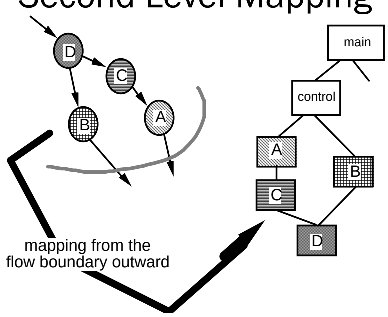
Factoring



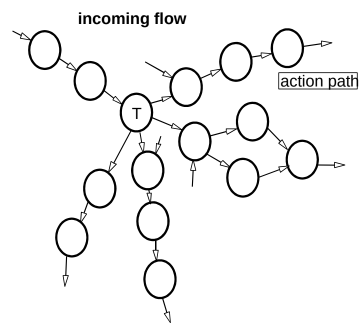
First Level Factoring



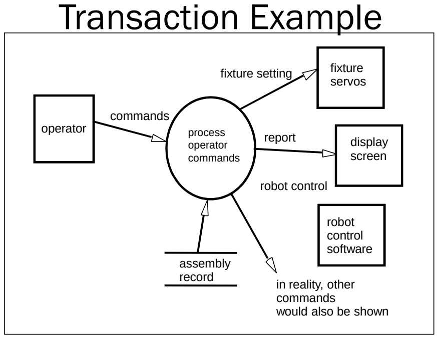
Second Level Mapping



Transaction Flow



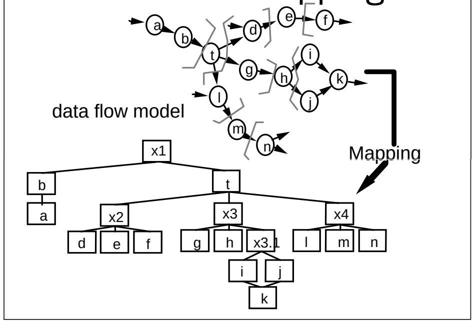
Transaction Example



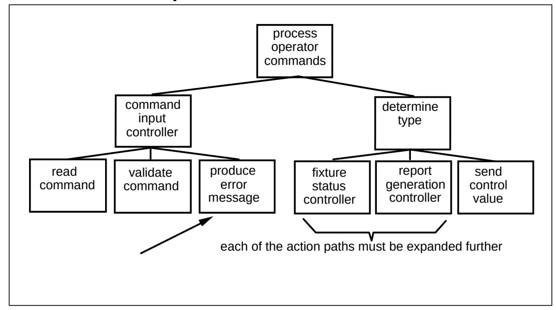
Transaction Mapping Principles

- isolate the incoming flow path
- define each of the action paths by looking for the "spokes of the wheel"
- assess the flow on each action path
- define the dispatch and control structure
- map each action path flow individually

Transaction Mapping



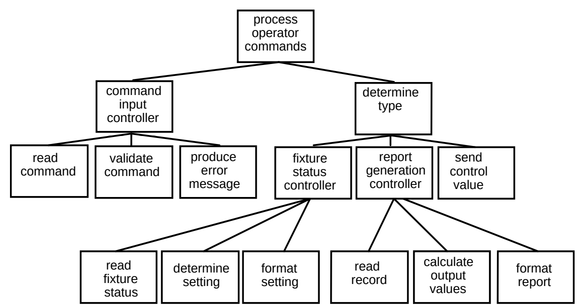
Map the Flow Model



Refining Architectural Design

- Processing narrative developed for each module
- Interface description provided for each module
- Local and global data structures are defined
- Design restrictions/limitations noted
- Design reviews conducted
- Refinement considered if required and justified

Refined Structure Chart



Architecture Design Assessment Questions

- How is control managed within the architecture?
- Does a distinct control hierarchy exist?
- How do components transfer control within the system?
- How is control shared among components?
- What is the control topology?
- Is control synchronized or asynchronous?
- How are data communicated between components?
- Is the flow of data continuous or sporadic?
- What is the mode of data transfer?
- Do data components exist? If so what is their role?
- How do functional components interact with data components?
- Are data components active or passive?
- How do data and control interact within the system?

Interfaces Design

- Inter-modular interface design
  - driven by data flow between modules
- external interface design
  - driven by interface between applications
  - driven by interface between software and non-human producers and/or consumers of information
- human-computer interface design
  - driven by the communication between human and machine

Place User in Control

- Define interaction in such a way that the user is not forced into performing unnecessary or undesired actions
- Provide for flexible interaction (users have varying preferences)
- Allow user interaction to be interruptible and reversible
- Streamline interaction as skill level increases and allow customization of interaction
- Hide technical internals from the casual user
- Design for direct interaction with objects that appear on the screen

Reduce User Memory Load

- Reduce demands on user's short-term memory
- Establish meaningful defaults
- Define intuitive short-cuts
- Visual layout of user interface should be based on a familiar real world metaphor
- Disclose information in a progressive fashion

Make Interface Consistent

- Allow user to put the current task into a meaningful context
- Maintain consistency across a family of applications
- If past interaction models have created user expectations, do not make changes unless there is a good reason to do so

User Interface Design Models

- Design model (incorporates data, architectural, interface, and procedural representations of the software)
- User model (end user profiles - novice, knowledgeable intermittent user, knowledgeable frequent users)
- User's model or system perception (user's mental image of system)
- System image (look and feel of the interface and supporting media)

User Interface Design Process (Spiral Model)

- User, task, and environment analysis and modeling
- Interface design
- Interface construction
- Interface validation

Task Analysis and Modeling

- Software engineer studies tasks human users must complete to accomplish their goal in the real world without the computer and map these into a similar set of tasks that are to be implemented in the context of the user interface
- Software engineer studies existing specification for computer-based solution and derives a set of tasks that will accommodate the user model, design model, and system perception
- Software engineer may devise an object-oriented approach by observing the objects and actions the user makes use of in the real world and model the interface objects after their real world counterparts

Interface Design Activities

- Establish the goals and intentions of each task
- Map each goal/intention to a sequence of specific actions (objects and methods for manipulating objects)
- Specify the action sequence of tasks and subtasks (user scenario)
- Indicate the state of the system at the time the user scenario is performed
- Define control mechanisms → object dan action
- Show how control mechanisms affect the state of the system
- Indicate how the user interprets the state of the system from information provided through the interface

Interface Design Issues

- System response time (time between the point at which user initiates some control action and the time when the system responds)
- User help facilities (integrated, context sensitive help versus add-on help)
- Error information handling (messages should be non-judgmental, describe problem precisely, and suggest valid solutions)
- Command labeling (based on user vocabulary, simple grammar, and have consistent rules for abbreviation)

User Interface Evaluation Cycle

1. Preliminary design
2. Build first interface prototype
3. User evaluates interface
4. Evaluation studied by designer
5. Design modifications made
6. Build next prototype
7. If interface is not complete then go to step 3

User Interface Design Evaluation Criteria

- Length and complexity of written interface specification provide an indication of amount of learning required by system users
- Number of user tasks and the average number of actions per task provide an indication of interaction time and overall system efficiency
- Number of tasks, actions, and system states in the design model provide an indication of the memory load required of system users
- Interface style, help facilities, and error handling protocols provide a general indication of system complexity and the degree of acceptance by the users

Data Design

- Data Design Principles
  - Systematic analysis principles applied to function and behavior should also be applied to data.
  - All data structures and the operations to be performed on each should be identified.
  - Data dictionary should be established and used to define both data and program design.
  - Low level design processes should be deferred until late in the design process.
  - Representations of data structure should be known only to those modules that must make direct use of the data contained within in the data structure.
  - A library of useful data structures and operations should be developed.
  - A software design and its implementation language should support the specification and realization of abstract data types.

Component Level Design

- The purpose of component level design is to translate the design model into operational software.
- Component level design occurs after the data, architectural, and interface designs are established.
- Component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built.
- The work product produced is the procedural design for each software component, represented using graphical, tabular, or text-based notation

Design Notation

- Flowcharts (arrows for flow of control, diamonds for decisions, rectangles for processes)
- Box diagrams (also known as Nassi-Scheidnerman charts - process boxes subdivided to show conditional and repetitive steps)
- Decision table (subsets of system conditions and actions are associated with each other to define the rules for processing inputs and events)
- Program Design Language (PDL - structured English or pseudocode used to describe processing details)

Program Design Language Characteristics

- Fixed syntax with keywords providing for representation of all structured constructs, data declarations, and module definitions
- Free syntax of natural language for describing processing features
- Data declaration facilities for simple and complex data structures
- Subprogram definition and invocation facilities

Design Notation Assessment Criteria

- Modularity (notation supports development of modular software)
- Overall simplicity (easy to learn, easy to use, easy to write)
- Ease of editing (easy to modify design representation when changes are necessary)
- Machine readability (notation can be input directly into a computer-based development system)
- Maintainability (maintenance of the configuration usually involves maintenance of the procedural design representation)

Design Notation Assessment Criteria (2)

- Structure enforcement (enforces the use of structured programming constructs)
- Automatic processing (allows the designer to verify the correctness and quality of the design)
- Data representation (ability to represent local and global data directly)
- Logic verification (automatic logic verification improves testing adequacy)
- Easily converted to program source code (makes code generation quicker)