Computer Science Program, The University of Texas, Dallas

Software Architectural Design

Lawrence Chung

You've saved your money,
you've purchased a beautiful piece of land,
and you've decided to build the house of your dreams.

Having no experience in such matters, you visit a builder and explain your desires
- number and size of rooms, contemporary styling, spa (of course!),
cathedral ceilings, lots of glass, etc.

The builder listens carefully, asks a few questions, and then tells you
that he'll have a design in a few weeks.

As you wait anxiously for his call, you conjure up many different
(and outrageously expensive) images of your new house.
What will he come up with? Finally, the phone rings and you rush to his office.

Pulling out a large manila folder, the builder spreads a diagram of the plumbing
for the second floor bathroom in front of you and proceeds to explain it
in great detail.

"But what about the overall design?" you say.
"Don't worry," says the builder, "we'll get to that later."

Modular Design

How?

- ❒ encapsulate coherent data items in a module
- ❒ all creation/destruction of these data items by routines in the module
- ❒ all access and reference to them through routines in the module

```
Module   module1
    local1: type1
    local2: type2
    local3: type3
    ...
    Operation   op1 (param1: paramType1, param2: paramType2, ...)
            local3 <- param1 * param2;
    end  op1

    Operation   op2 (param1: paramType1, param2: paramType2, ...)  returns R: type3
            return(local3)
    end  op2
    ...
end  module1
```

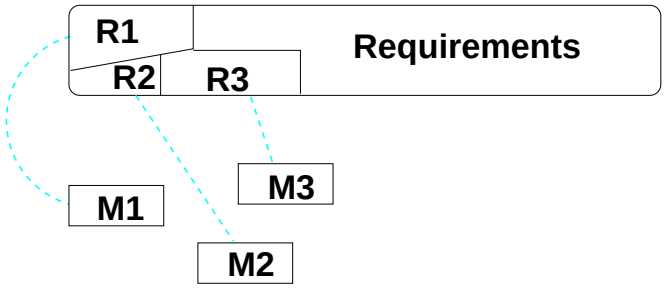
```
Module   module2

    Operation   op21 (..)
        module1.op1(p1, p2, ...)
        local1 <- module1.op2(...);
```

```
Module   module2
    local1: type1
    Operation   op21 (..)
        module1.local3 <- 5;
        local1 <- module1.local3;
```

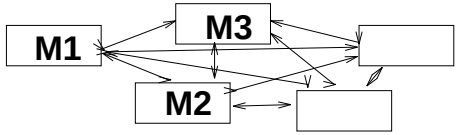
Modular Design

Functional Independence

- ❒ Each module satisfies a small part of overall requirements



- ❒ each module has a small interface:
  - small # of exported items *export op1, op2, ..., op8999, op9000*
  - small # of imported items *module1. op1, module1.op2, ..., module200.op5*
  - small # of parameters for exported functions/procedures
    Operation op1(param1: Type1, param2: Type2, ... param2000: Type2000)
- ❒ reduce interactions between modules



Not desirable

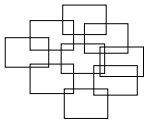
Cohesion and Coupling are measures of functional independence

High Cohesion & Low Coupling

Cohesion

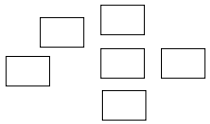
a measure of the "self containedness" of a module
to what degree does it perform a "single" task

- Coincidental cohesion *(Low Cohesion)*



setting: 20,000 lines of code with one 2,500 line subroutine
manager: in your spare time, modularize the program
programmer: what's the proper length of a module?
staff: 75
programmer: with a red pen and a ruler, draws a red line after each 75 lines of code

- logical, temporal, procedural, communicational, sequential
- Functional cohesion *(High Cohesion)*

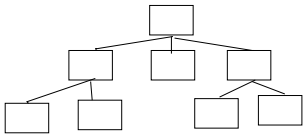


operations implement a single specific function

Coupling

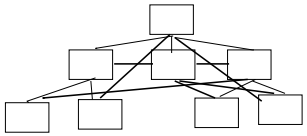
a measure of the degree of interconnections among modules in a sw system

- Data coupling *(Low Coupling)*



modules communicate via simple data parameters

- ... via complex data structures
  ... via control flags, external files, global data



- Content coupling *(High Coupling)*

modules use data maintained by another module or one module branches into another module

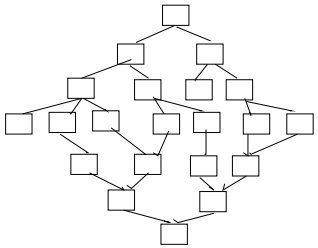
Software Architecture, Design of Data & Procedures

Software Architecture

- **like blueprints**
- **cf. computer architecture**

- Recently recognized as a major element in system design
- A good sw designer will have a broad knowledge of architectural alternatives for a given system

  a good client-server architecture, a bad client-server architecture
- Structure chart can be viewed as a type of sw architecture



castles, towers, domes, condominiums

From DFD to Program Structure

❒ Synonymous with Control Hierarchy, Structure Chart

- represents top-down distribution of "control" among modules (cf. OO)
- does not show procedural aspect (sequencing, repetition, decision making, etc.)

❒ Most applicable when

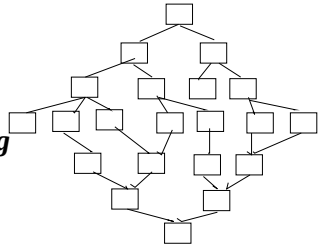
- information is processed (mostly) sequentially
- there is no formal hierarchical structure to the data (cf. OO)

❒ Steps

1. establish type of information flow: *transform or transaction?*
2. determine information flow boundaries: *centers, bundles, individuals?*
3. map the DFDs into a structure chart
4. use "factoring" to determine the program control hierarchy

first-level factoring

second-level factoring



decision making

mix

input/computation/output

5. tune the structure for performance and quality

Types of Flows

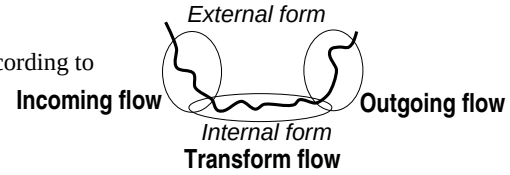
Transform Flow

❒ Transform flow exists when

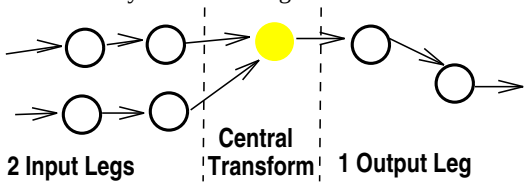
incoming input data is transformed within the system into an "internal form"
information in the system is subject to transformations while in its internal form
outgoing output data is transformed within the sys from an internal form to its external form

❒ In this case,

level 0 DFD partitions the system according to



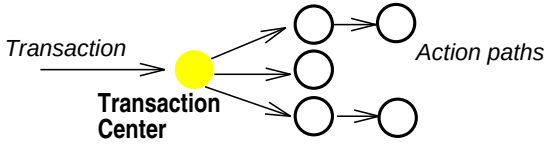
level n DFD partitions the system according to



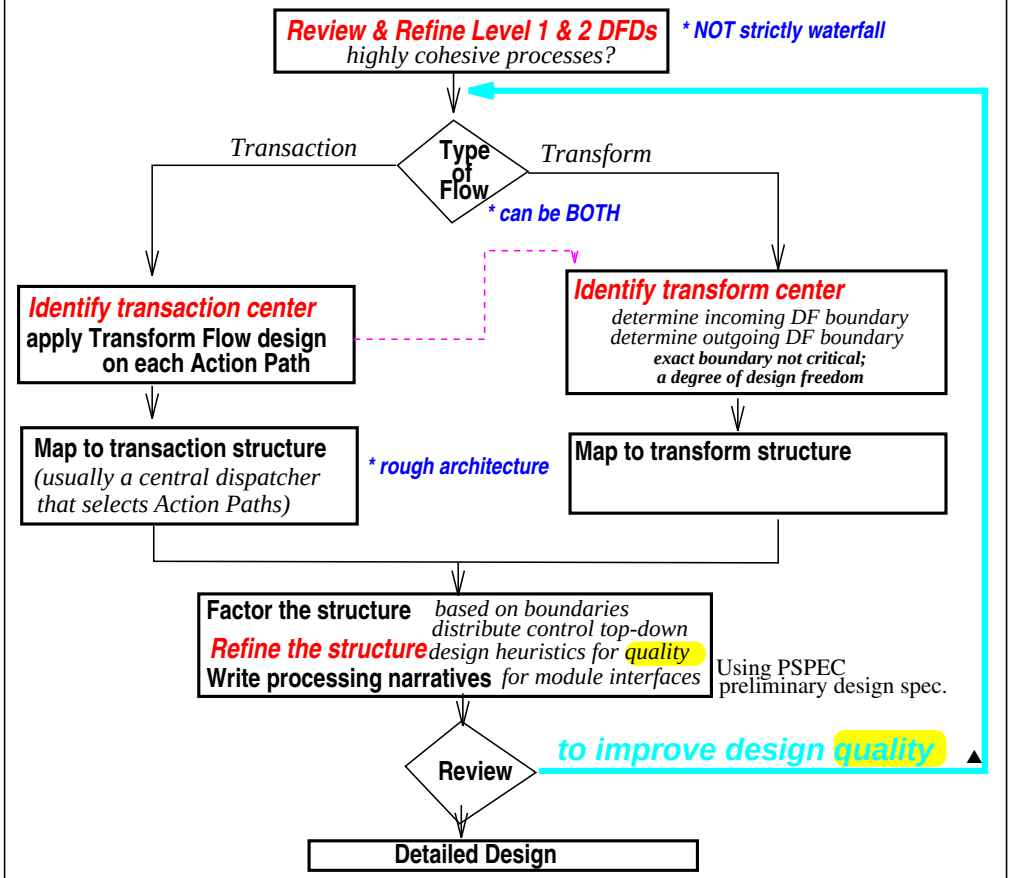
Transaction Flow

A specialized form of Transform Flow where a single input item (transaction) triggers data flow along several (action) paths

Many interactive systems and query/response systems fit this model

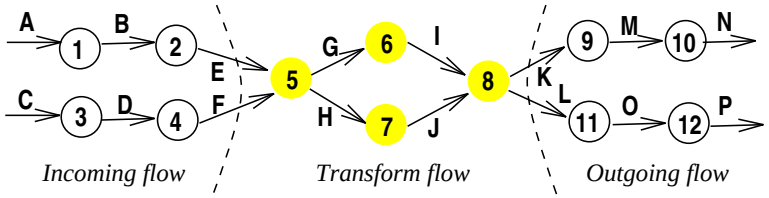


DFD-Oriented Architectural Design Process

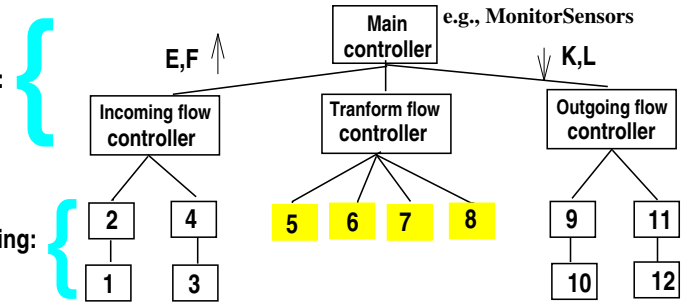


Transform Analysis

1. **Review Level 0 & 1 DFDs (or more) and supporting documentation**
2. **Review and Refine DFDs:** *for highly cohesive modules*
3. **Identify Transform center:** *Where is I/O data translated from external/internal to internal/external?*
   several incoming paths and/or several outgoing paths --- at high level?
   one incoming and one outgoing path --- (*the lowest-level DFD)
4. **Identify Boundaries around Transform center**



5. **Perform First-Level Factoring:**



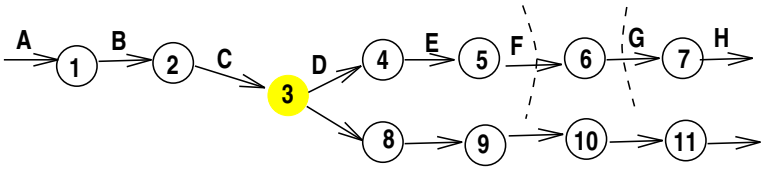
6. **Perform Second-Level Factoring:**

7. **Refine the "First-Cut" Structure Using Design Heuristics for better Quality:**

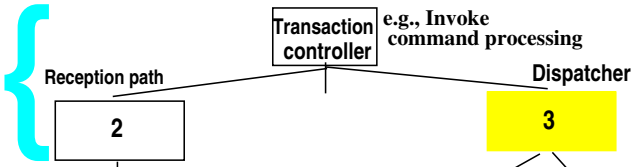
Low coupling & high cohesion; Low fan-out & high fan-in as depth increases;
Low complexity for module interfaces; Single-Entry-Single-Exit Modules; etc.

Transaction Analysis

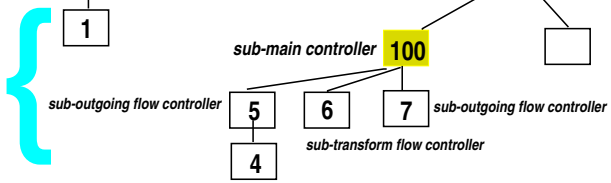
1. **Review Level 0 & 1 DFDs (or more) and supporting documentation**
2. **Review and Refine DFDs:** *for highly cohesive modules*
3. **Identify Transform center:**
   A single data item, a transaction, triggers other data flow along many paths
4. **Identify Boundaries around Transform center**



5. **Perform First-Level Factoring:**



6. **Perform Second-Level Factoring:**



7. **Refine the "First-Cut" Structure Using Design Heuristics for better Quality:**
   Low coupling & high cohesion; Low fan-out & high fan-in as depth increases;
   Low complexity for module interfaces; Single-Entry-Single-Exit Modules; etc.

Data Flow Heuristics

- Reduce coupling & increase cohesion
  - **"explode" complicated modules into multiple modules**
  - **"implode" related modules**
- Minimize structures with high fan out
  - **Fan in should increase with design level**
- Decision in a module should affect only subordinate modules
- Tune module interfaces to reduce complexity
  - **E.g., order & type of parameters, naming conventions**
- Try to make modules "single-entry, single-exit"

DOCUMENT each module, interface, locall/global data structure design restrictions/limitations

References

R. Pressman, Software Engineering: A Practitioner's Approach, Prentice-Hall
T. Demarco, Structured Analysis and System Specification, Prentice-Hall
M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, PH