# 6. ARCHITECTURAL DESIGN

Last Mod: 2024-05-28

This section of the course will briefly look at:

- Large scale system design.
- Both object-oriented and some other design methods.
- File space/speed design trade-offs.
- Subclass storage space/speed trade-offs.

# Table of Contents

**6. ARCHITECTURAL DESIGN ........................................................ 1**

6.1 INTRO TO ARCHITECTURAL DESIGN ........................................................ 3

*6.1.1 Interaction........................................................................................13*

*6.1.2 Tront's Design Definition........................................................................21*

6.2 AN AIR TRAFFIC CONTROL EXAMPLE ........................................................ 23

*6.2.1 Decomposition ........................................................................................28*

*6.2.2 Break Each Subsystem into Tasks/Programs ..........................................31*

*6.2.3 Location Breakdown................................................................................33*

*6.2.4 Conceive Purpose/Responsibilities of Modules........................................35*

*6.2.5 Software Module Interface Design..........................................................38*

*6.2.6 Low Level Design.....................................................................................39*

## 6.1 Intro to Architectural Design

By this sub-phase of the design phase, you should have a good idea of what the proposed system is exactly capable of doing (from requirements specification), and what it will look and operate like (from the draft user manual).

Architectural Design is also called ‘High Level Design”.

We hierarchically decompose/partition the system into “manageably-small” components.

- Subsystems,
- Modules, and
- eventually to classe.
  - Or classes that front for a subsystem/collection of classes.
    - E.g. Façade pattern.

Each should (hopefully) be simpler in scale, and subsequently simpler to individually design.

Decomposition should get us to components that are small enough in scope and functionality that they are:

- easier to attempt a deeper understanding of,
- easier to design, and
- easier to later program them.

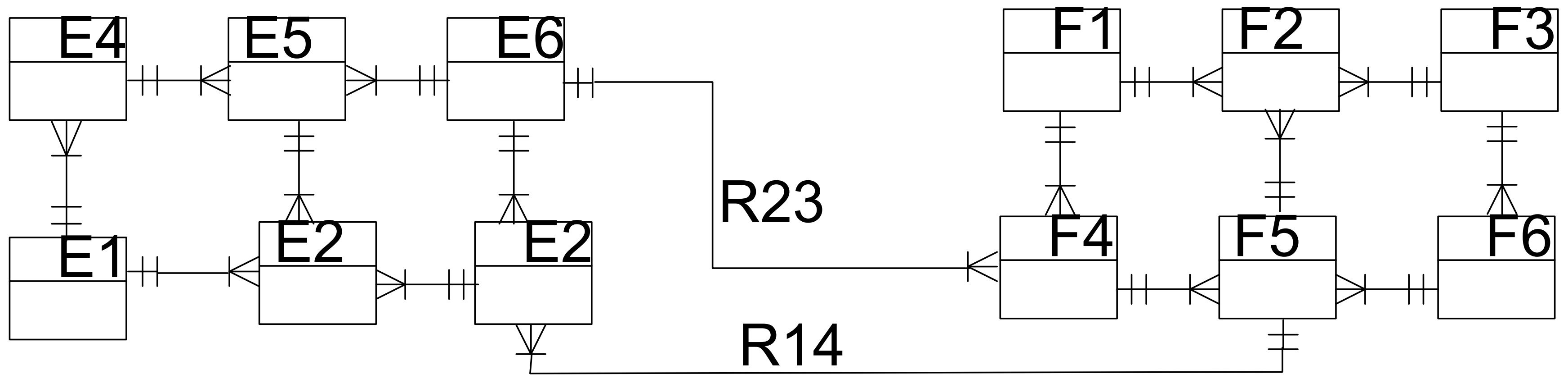
## The decomposition is necessary:

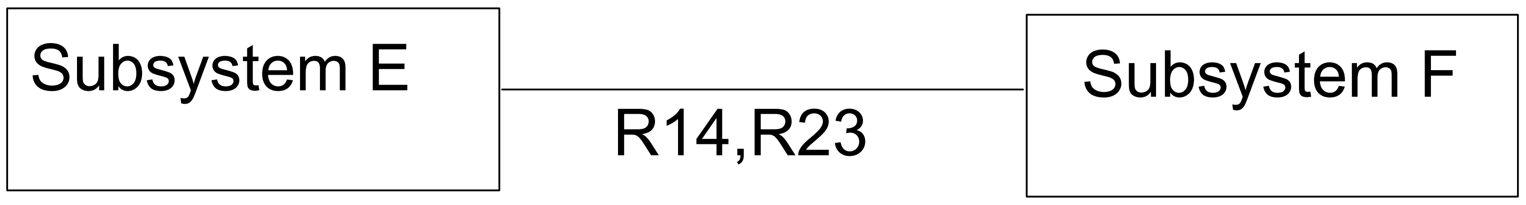
- So resultant pieces small enough and independent enough they can be assigned to different people to design separately.
  - Large projects have more than one staff member!
- Pieces small and 'independent' enough that they're modifiable/correctable/substitutable during the design and coding without unduly affecting the design of the rest of the subsystems and their designs/designers.
  - E.g. Replace list container with tree-based container.

- So that simple, cleanly designed components may be re-used in several subsystems or in later, completely separate projects/products.
- So if you just change a component or two, the whole system need not be recompiled (10 hours for 10 million lines).
  - Instead can recompile only the few components, then relink.

Decomposition may be of the Subsystem Relationship Diagram (SRD), of Data Flow Diagrams, and later State Transition Diagrams.

## Concept of Subsystem Relationship Diagram:





Decomposition is necessary since humans are relatively poor at handling 7+ things at once. E.g.

- Diagram with 7+:
  - Entities and/or connecting lines, or
  - states or
  - processes.
- Module/class with more than 7 member functions.

With 10+ entities or relationships under consideration, humans make errors or oversights in

- design,
- connectivity/relationship lines
- logic,
- comprehension, and in
- their ability to review someone else's design for correctness.

Luckily, this principle applies at each level of hierarchy. Thus a very high-level diagram with 7 sub-systems is graspable.

In some cases, decomposition is part of analysis.

But we may want to revisit that decomposition, and do it again in a more implementation-dependent manner.

We may also modify/re-organize/further the decomposition to:

- improve its organization,
- optimize (or more often trade-off) the design for the desired memory vs. performance.
- decide on deployment of the system over several CPUs at several locations.
  - Might certain processes only run on one CPUs at one location, or run at all locations?
- change it to better allow error handling.
- allow portability to different OSs/GUIs/networks/DBMSs, or migrate the UI to different spoken languages.

## ***6.1.1 <u>Interaction</u>***

For a partitioned system to carry out the functionality desired, the components can’t be totally independent.

- They must interact to get the job done.

The interaction should be simple, clean, and well defined.

The interaction may take one of these forms:

- function call
- shared memory and flags
- message passing (e.g. unix pipes or network packets)
- remote procedure call

The sender and receiver of an interaction must use compatible mechanisms:

- function parameters of correct type, order, and format
- same communication protocol
- same language and characters set (e.g. Java uses unicode).

Writer of a component that provides a service to callers advertises the necessary details for another component to interact and obtain the service in an **interface definition**.

Actually, sometimes a callee is not providing a service, but is expecting to be notified by another component of some event or data.

In any case, the interface definition may take the form of a:

- well-documented C++ header file. This is like an Application Programmer Interface (API), except that it's not necessary for application programmers needing to use say MS-Windows, but for any programmers in your company who may need to get the service.
- a standard or specially-created protocol and a network+port address to send to. Or Unix "pipe".
- global variable names and legal values
  - And if multithreaded, critical section protection mechanism.

Architectural design and partitioning is:

- part art,
- part methodology,
- part experience, and
- part creating a good architectural vision or layout of the system.

In [Booch94], Grady Booch, one of the foremost OO methodologists and consultants, states that in his experience, with *troubled*, large projects:

“One of the traits absent from most projects which fail is **a strong architectural vision**”.

Of course, a strong vision of a bad design is no good.

If you find yourself writing huge and complex interface definitions, you may have your functionality partitioned in a non-optimal way into the various components.

## You may have to rethink things!

- This isn’t easy.
- May a long contemplative walk,
- Have a large brainstorming meeting where you discover a completely different way to partition the systems functionality into a previously unthought of component organization.
  - And this new one might have simpler, more sensible interfaces.

## *6.1.2 <u>Tront’s Design Definition</u>*

**Tront’s Design Definition**:

“Design is evaluating the advantages and disadvantages of alternative designs, and choosing one that has the best trade-off of

– speed,

– memory usage,

– implementability,

– maintainability, and

– reviewability.”

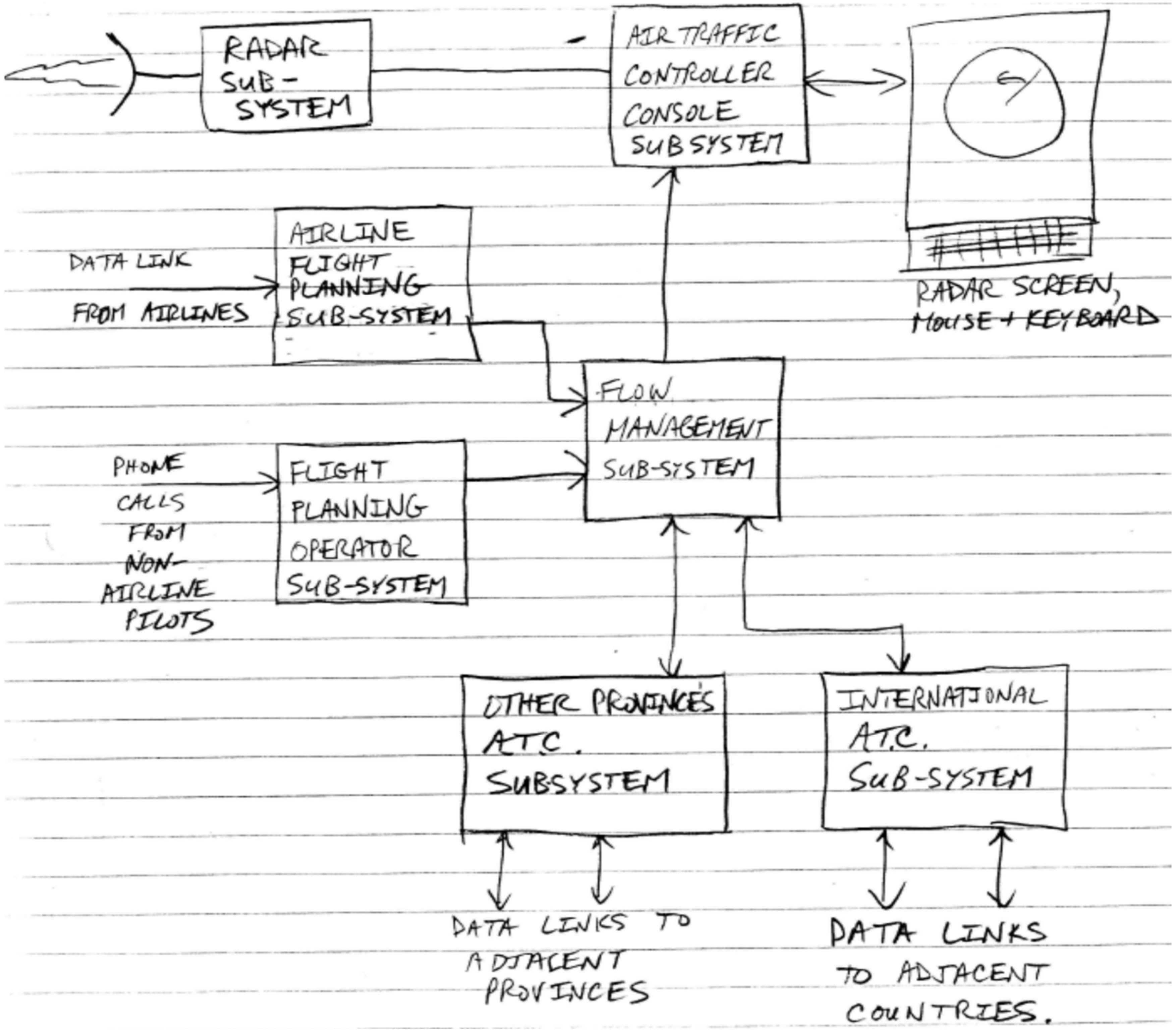
## As a final introductory note:

- Top-down decomposition allows some delay to decide some implementation issues until:
  - they are less likely to require change, or
  - because of abstraction they more easily changed independently.

## 6.2 An Air Traffic Control Example

(Note: Most of the Canadian Automated Air Traffic System (CATS) was designed and written in Vancouver.)

Considering the following “fictional” initial system/subsystem structure for big, distributed air traffic control system.



RADAR SUB-SYSTEM
AIR TRAFFIC CONTROLLER CONSOLE SUBSYSTEM
RADAR SCREEN, MOUSE + KEYBOARD
DATA LINK FROM AIRLINES
AIRLINE FLIGHT PLANNING SUB-SYSTEM
FLOW MANAGEMENT SUB-SYSTEM
PHONE CALLS FROM NON-AIRLINE PILOTS
FLIGHT PLANNING OPERATOR SUB-SYSTEM
OTHER PROVINCES ATC. SUBSYSTEM
INTERNATIONAL ATC. SUB-SYSTEM
DATA LINKS TO ADJACENT PROVINCES
DATA LINKS TO ADJACENT COUNTRIES.

There are 8 boxes.  If there were any more in this decomposition of the whole, it would be more probable that boxes or links would be mistakenly left out or mis-understood.

You certainly can’t begin writing code for a system until you know even what sub-system you are writing, and how to ‘talk’ to other sub-systems/CPUs.

- Call
- Network link
- Radar data format to exchange.

Note:

- Architectural Design is normally programming language independent.
- The resultant architecture design spec should be implementable in either of several programming languages.

## ***6.2.1 <u>Decomposition</u>***

In a large system composed of hundreds of program modules, making up dozens of tasks (running on several distributed CPUs), the first job is to just break the system into subsystems.

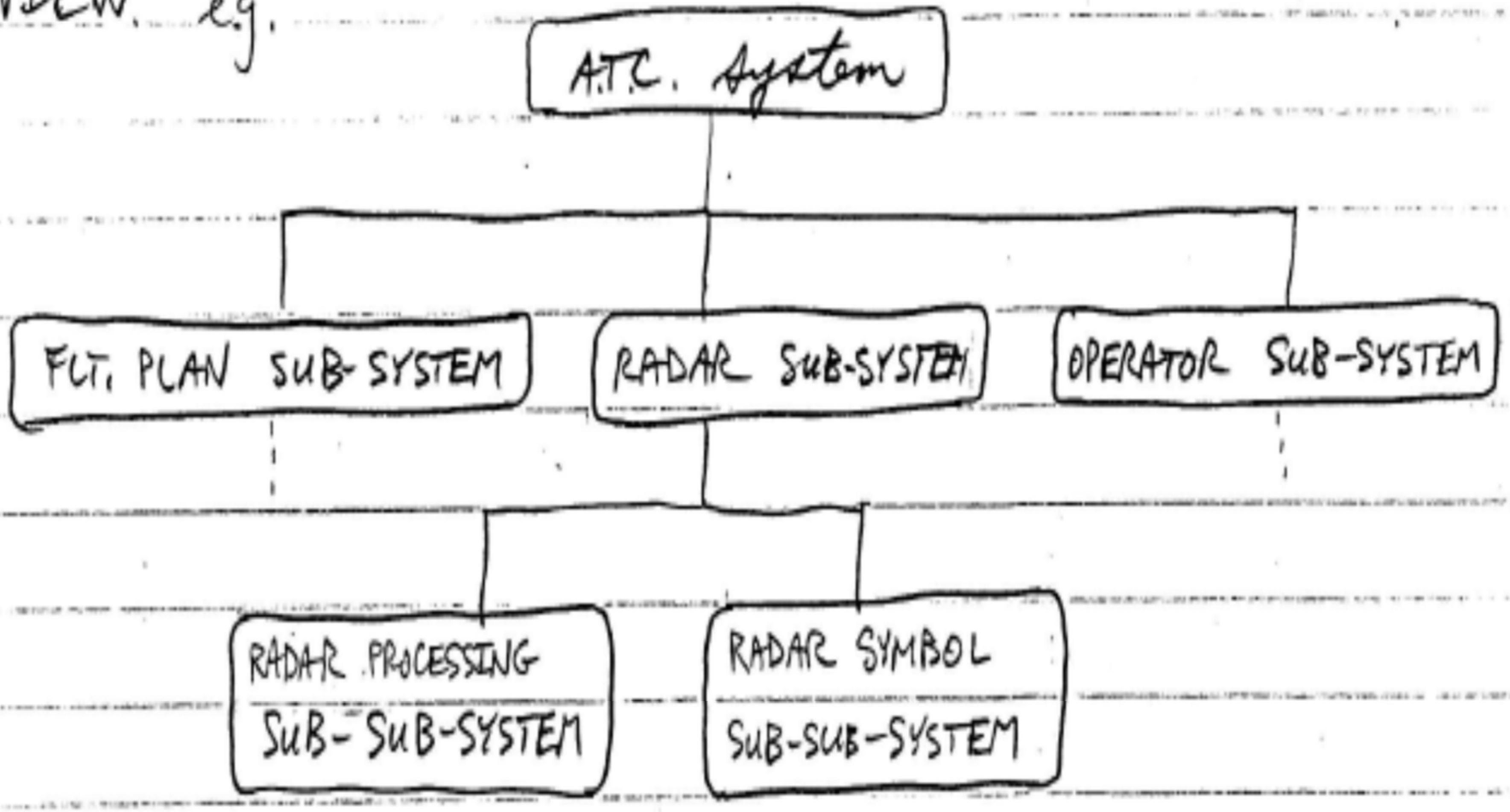
Basically, this means determining the **purpose/responsibility** and **name** of each subsystem.  E.g.

- The “flight planning subsystem” accepts flight plan info from several kinds of sources at each air traffic control station in Canada (each running on a different CPU).  It stores and distributes this data as necessary.

A hierarchical composition diagram can be useful. It shows what systems and subsystems are made up of what other subsystems.

- (Unlike the previous diagram that showed communication links).

A HIERARCHICAL DIAGRAM IS USEFUL FOR CRITICAL REVIEW. e.g.

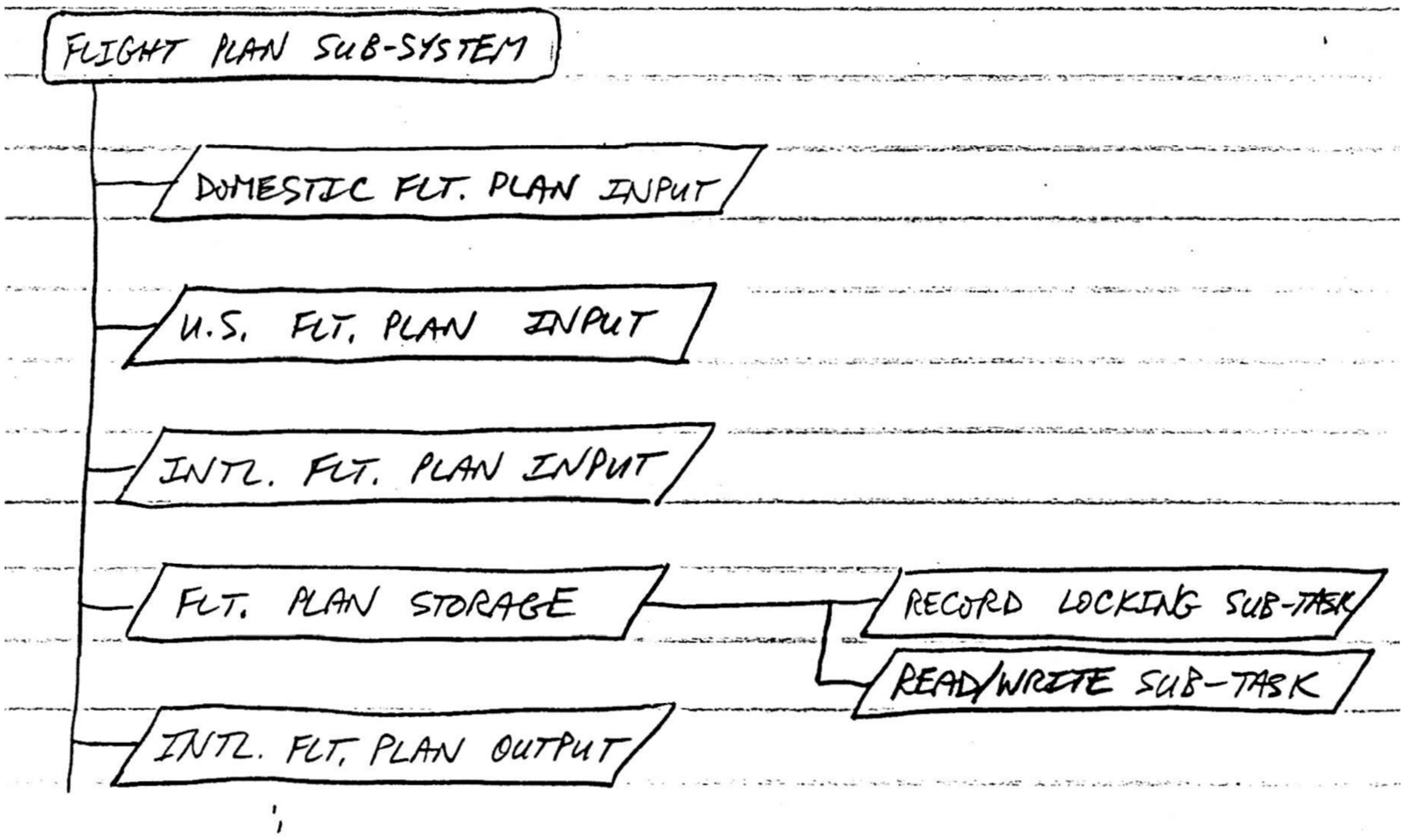


## ***6.2.2  <u>Break Each Subsystem into Tasks/Programs</u>***

Second job is breaking each subsystem into tasks and programs.  E.g.

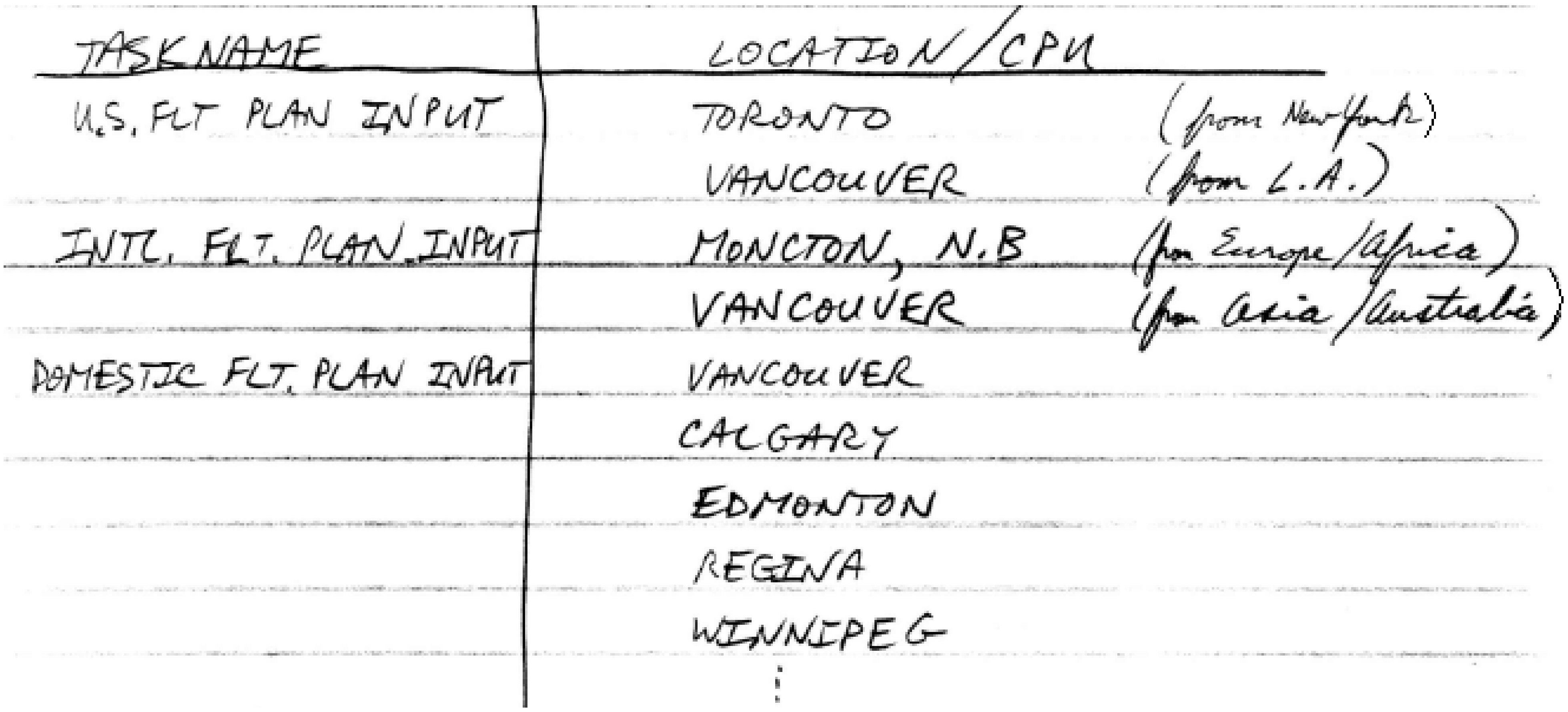
“The U.S. flight plan input task/program accepts data from U.S. sources in U.S. format.  It translates it into Canadian format, and enters it in the flight plan database.

## Here's a further hierarchical breakdown (even though it looks more like a list).



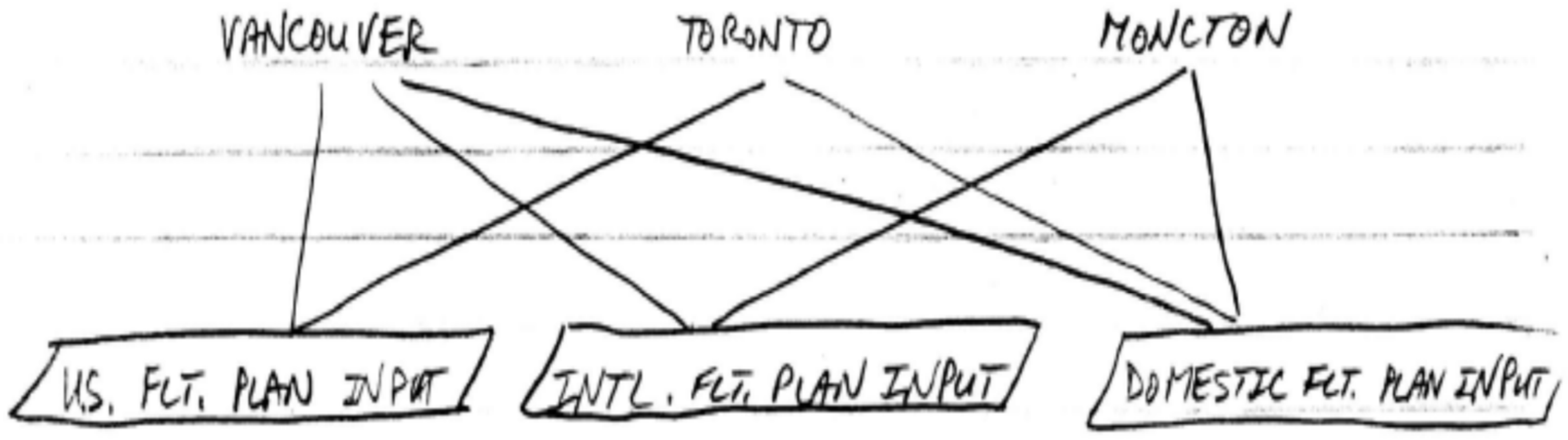
## 6.2.3 Location Breakdown

The third job in a large system architecture is to determine the number and location of each task/program. This could be a table.



| TASKNAME | LOCATION/CPU | |
|---|---|---|
| U.S. FLT PLAN INPUT | TORONTO | (from New York) |
| | VANCOUVER | (from L.A.) |
| INTL. FLT. PLAN INPUT | MONCTON, N.B | (from Europe/Africa) |
| | VANCOUVER | (from Asia/Australia) |
| DOMESTIC FLT. PLAN INPUT | VANCOUVER | |
| | CALGARY | |
| | EDMONTON | |
| | REGINA | |
| | WINNIPEG | |
| | ⋮ | |

Corresponding to the above table, you could alternately draw a diagram.   Or invert the indexing of this list to show how many and which tasks are running in each city's CPU.



## ***6.2.4 <u>Conceive Purpose/Responsibilities of Modules</u>***

Now conceive the purpose and responsibilities of each task's modules.

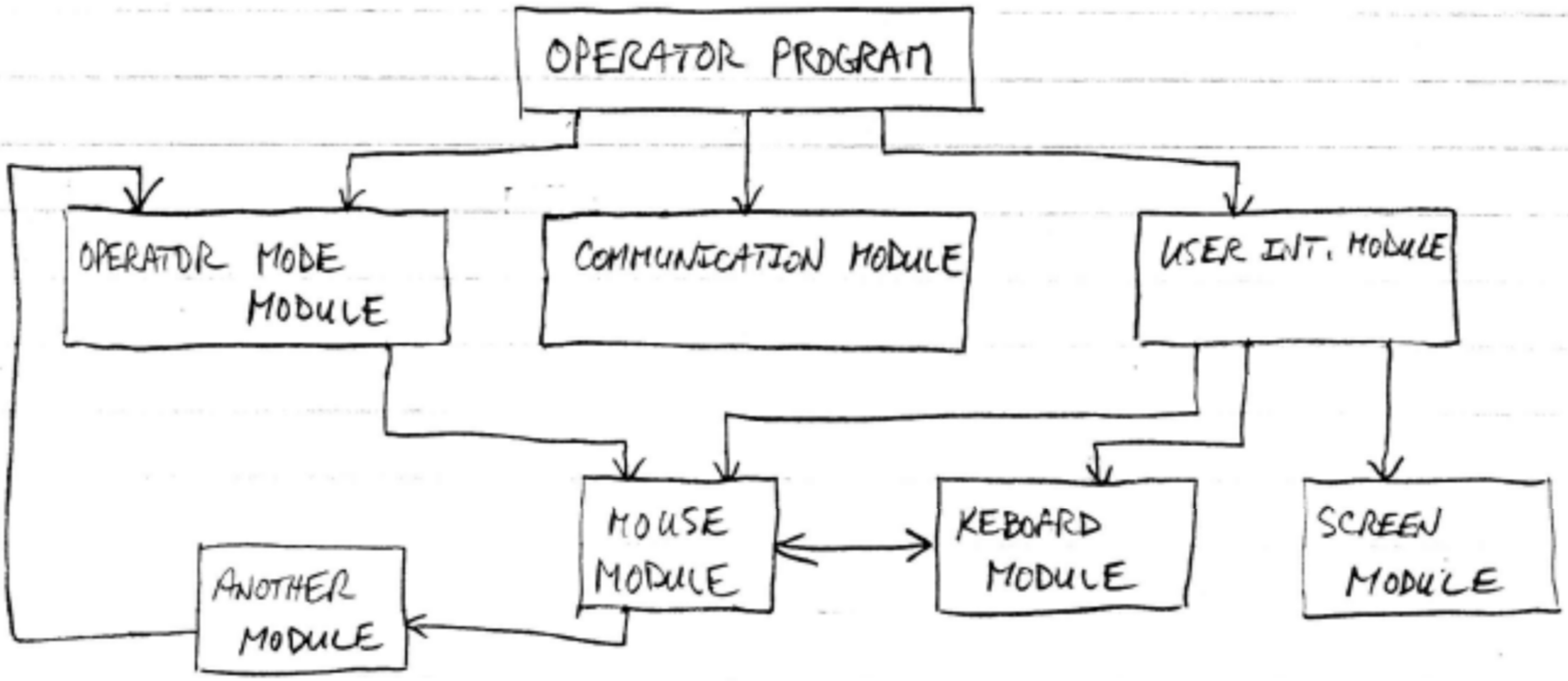
By 'modules' I mean software programming files.

- Typically a .cpp file and it's .h file.
- A Java class.

This is not trivial if there are 20+ modules.

E.g. The user interface module concerns itself with the presentation and interaction mechanism with the user. It uses the services of lower level keyboard, mouse, and screen display modules.

In addition to a text description like the above for each module, some kind of hierarchical module chart is useful for critical review of the modules and their import/#include dependencies.



## This is not a tree.

- Arrows show dependency of either .h or .cpp parts of a module on another module.
- And mutual or circular imports should be noted for later resolution.

### 6.2.5 ***Software Module Interface Design***

Next is software interface design.

- Names and types of exported programming language functions (and their parameter types).
- Exported variables.
- Exported types and constants.

All the above documented with comments!

Interface design will be described later in the course.

## ***6.2.6 Low Level Design***

Low level design is choice and reasoning for:

- Data structures
  - file record data fields.
  - Object class data members.
- Algorithms
  - Will stack be implemented as an array or linked list?
  - In RAM or on disk?
- Network packet internal data layout.
  - Network packets are fixed width data record/struct.
  - Though some might be variable length.

It’s especially important to document why a given design was chosen, over some alternate.

- Can be documented in code comments.

Low level design will be discussed later in the course.