7. ARCHITECTURAL DESIGN

Last Mod: 2024-06-02

This section of the course will briefly look at:

- Large scale system design.
- 1NF/3NF file space/speed design trade-offs.
- Subclass storage space/speed trade-offs.
- Many general design principles (in depth).
- Appendix A: Test Cases (for Assignment #3)

Table of Contents

7. ARCHITECTURAL DESIGN 1

7.1 INTRO TO ARCHITECTURAL DESIGN 4

7.1.1 Interaction 14

7.1.2 Tront's Design Definition 22

7.2 AN AIR TRAFFIC CONTROL EXAMPLE 24

7.2.1 Decomposition 29

7.2.2 Break Each Subsystem into Tasks/Programs 32

7.2.3 Location Breakdown 34

7.2.4 Conceive Purpose/Responsibilities of Modules 36

7.2.5 Software Module Interface Design 39

7.2.6 Low Level Design 40

7.3 FILE SPACE VS. ACCESS SPEED 42

7.3.1 2NF vs. 1NF Calculation 47

7.3.2 3NF vs. 2NF Calculation 54

7.4 SUB-CLASS STORAGE IMPLEMENTATIONS 64

7.4.1 Roll Up, Roll Down, or Split 67

7.5 GENERAL DESIGN PRINCIPLES 76

7.5.1 Design Decomposition 81

7.5.2 Design Hiding 84

7.5.3 Encapsulation *91*
7.5.4 Cohesion *93*
7.5.5 Coupling *97*
7.5.6 Abstraction *106*
7.5.7 Hierarchical Abstraction *107*
7.5.8 Data Abstraction *113*
7.5.9 Control Abstraction *115*
7.5.10 Scope/Effect of Control *119*
7.5.11 Fan-in/Fan-out *132*
7.6 DESIGN SPECIFICATIONS 135
7.7 DESIGN REVIEWS 136
7.8 REFERENCES 138
7.9 APPENDIX A – BLACK BOX TESTING 140
7.9.1 Test Cases *143*

7.1 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.

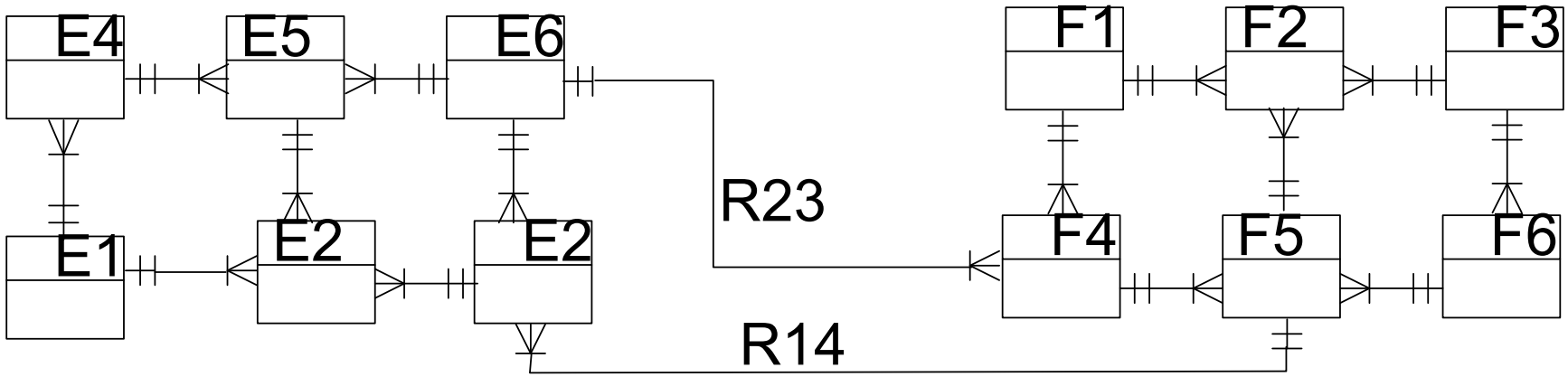
The decomposition is necessary:

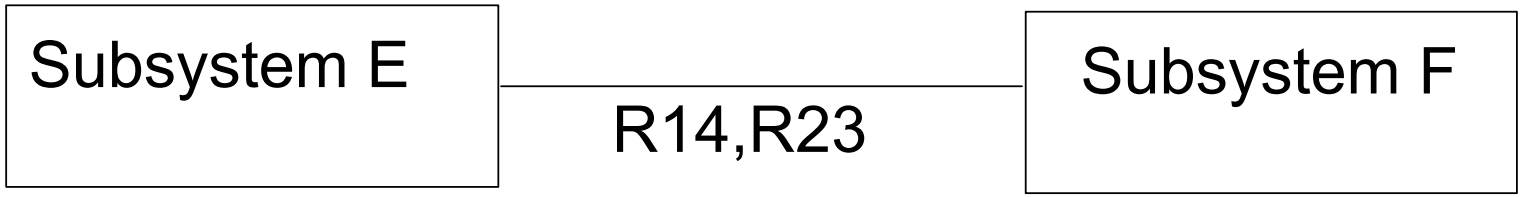
- So resultant pieces small enough and independent enough they can be assigned to different people to design separately.
  - Large projects have more than one staff member!
- Pieces small and 'independent' enough that they're modifiable/correctable/substitutable during the design and coding without unduly 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.

7.1.1 Interaction

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.

7.1.2 Tront's Design Definition

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."

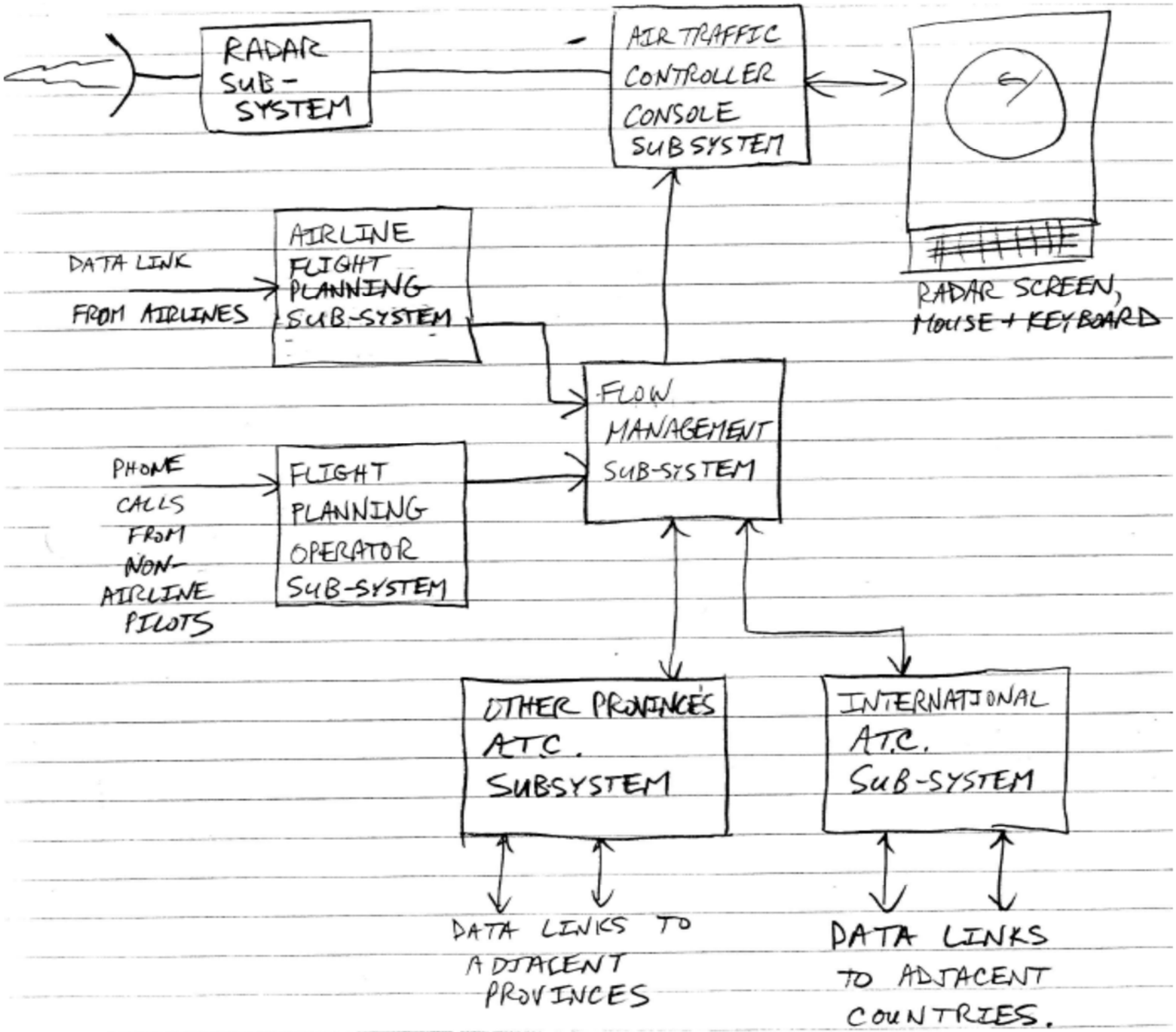
As a final introductory note:

- 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.

7.2 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 A.T.C. SUBSYSTEM
INTERNATIONAL A.T.C. 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.

7.2.1 Decomposition

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.

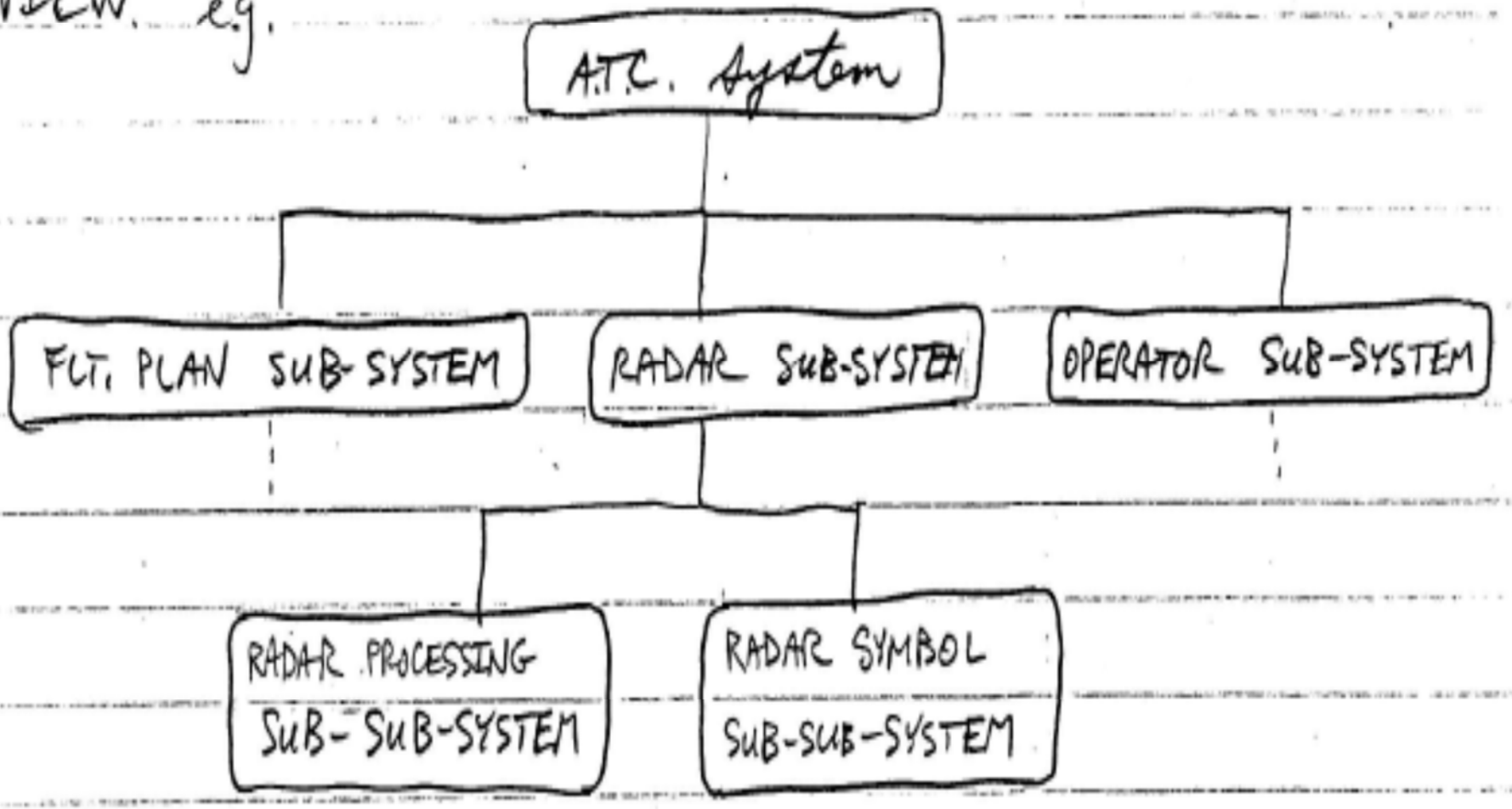
Basically, this means determining the **purpose/responsibility** and **name** of each subsystem. E.g.

- 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.

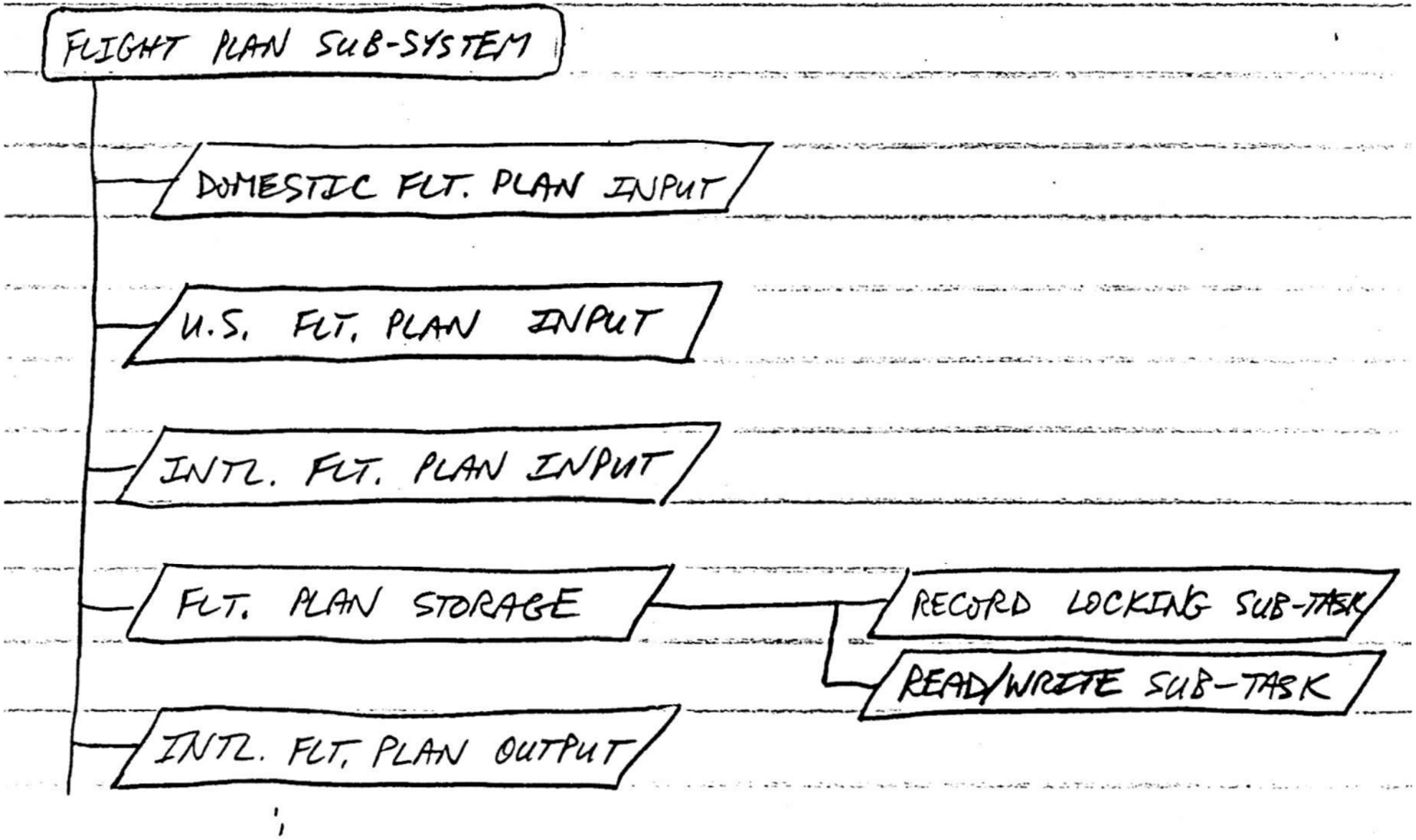


7.2.2 Break Each Subsystem into Tasks/Programs

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).

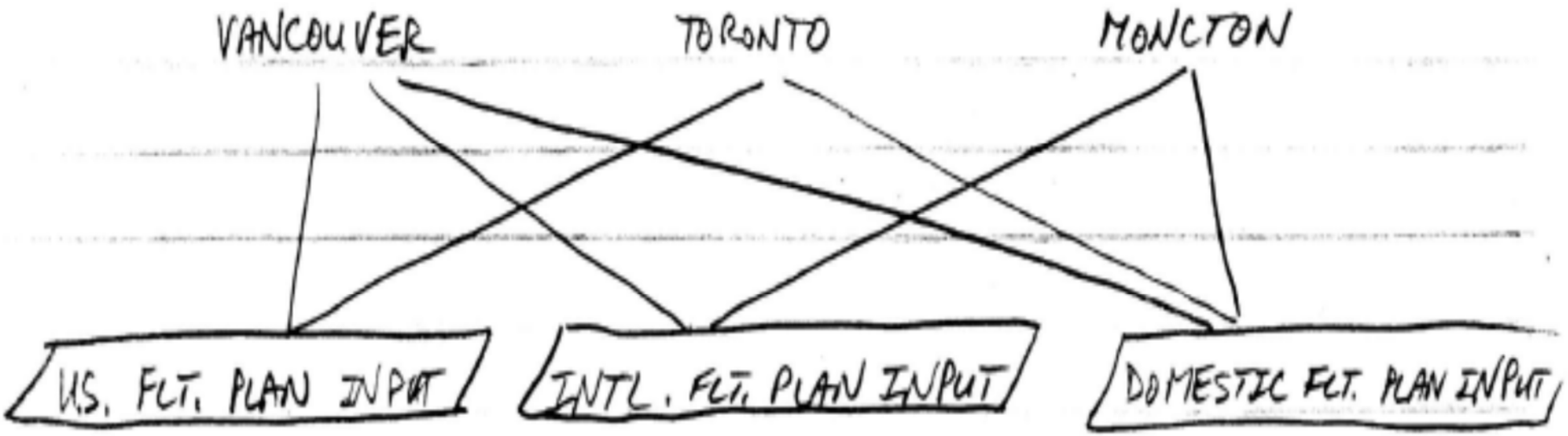


7.2.3 Location Breakdown

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

TASK NAME	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.



7.2.4 Conceive Purpose/Responsibilities of Modules

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

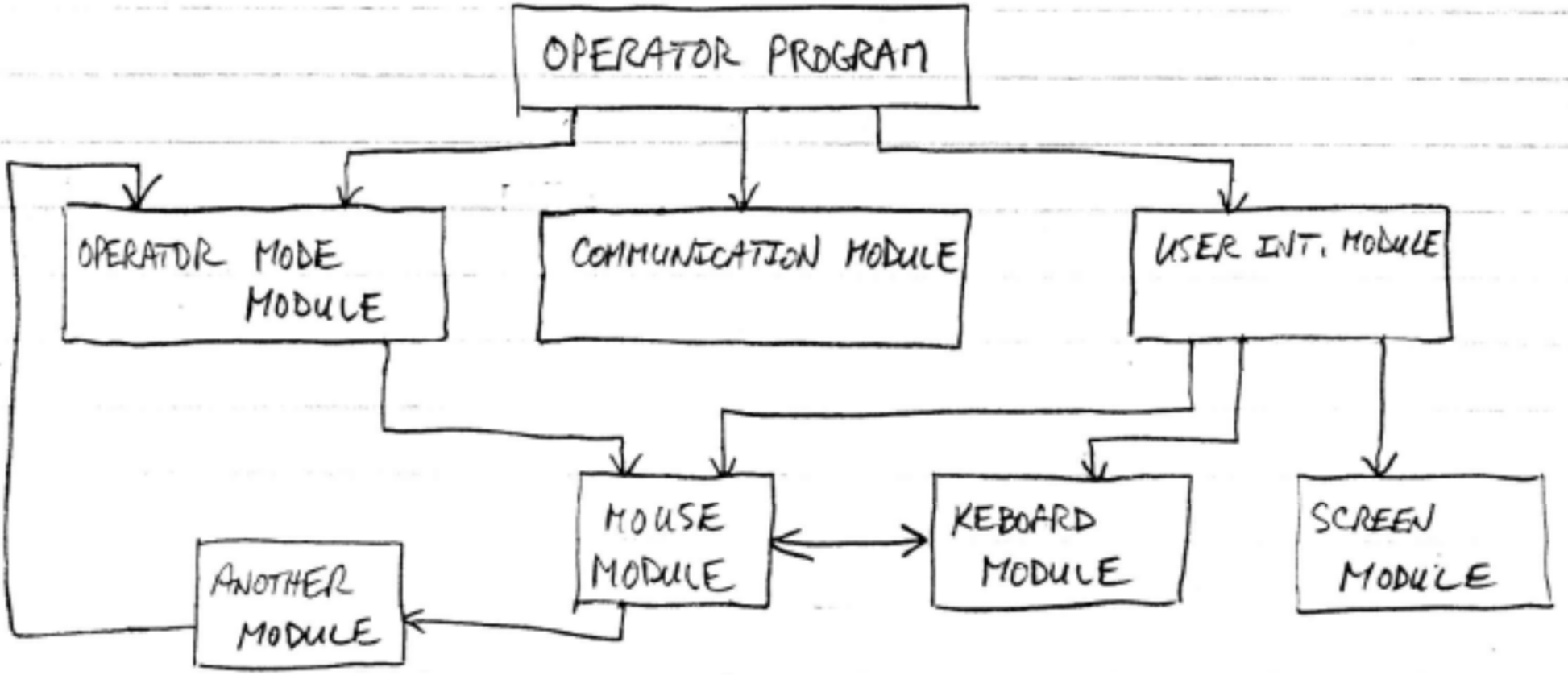
By 'modules' I mean software programming files.

- 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.

7.2.5 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.

7.2.6 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.

7.3 FILE SPACE VS. ACCESS SPEED

This subsection is an aside from Architectural Design.

Instead, it expands and completes the previous coverage of 2NF vs. 3NF, and subclass entities. In particular, prepares you to answer midterm questions regarding space needed for trade-offs between alternative storage methods for the above concepts.

It also briefly covers trade-offs in access speed and insert/update/delete anomalies.

Full normalization '*usually*' results in the least use of space, but also usually results in slower access of the data that was split off into separate tables (i.e. files) by the normalization process.

- Why? Because have to do disk accesses in *two* different files.

In addition to this performance penalty, occasionally normalization can actually *increase* space.

This occurs when certain combinations of relative entity size and relative frequency (i.e. relative populations of the related entities) occur.

In these cases, if dealing with databases the size of Air Canada's, it may save you Gigabytes of disk space (and increase speed) if the data model is 'denormalized'.

The denormalization drawbacks are:

- Updates, on what would have been one row of a normalized table, may now require updates to multiple rows in order to update the redundant copies.
  - This is particular true in 1NF. But also 2NF.
  - E.g. Storing professor's office # in every student record for the course offering.
- Possible re-introduction of inconsistencies among the redundant copies (if the inputs just come typed in rather than 'selections' from scrolling lists.

In the next subsection, we will look at the 2NF vs. 1NF trade-off in an algebraic manner.

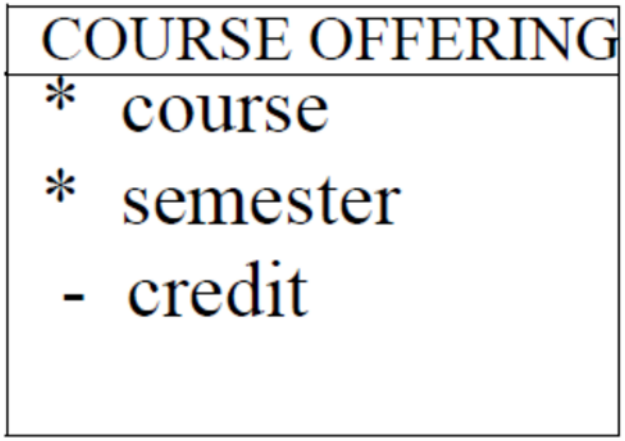
Then, we look at the 3NF vs. 2NF problem.

Lastly, we look at options for implementing class hierarchies.

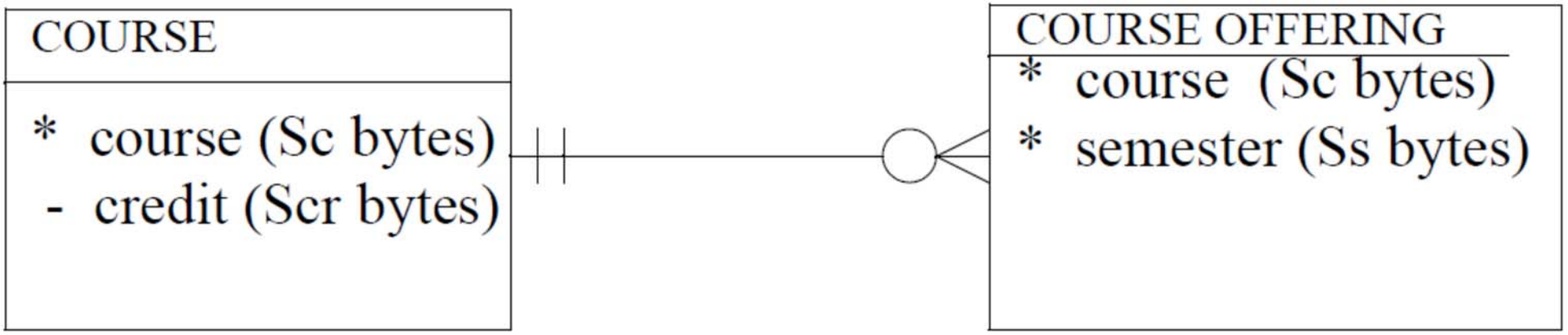
- And see some language and methodological issues posed by persistent objects.

7.3.1 2NF vs. 1NF Calculation

Let us assume this 1NF entity:



Credit is dependent on only the course, and not what semester the course is offered in:



This normalization to 2NF *usually* reduces the amount of space required.

- Because credit attribute doesn't have to be stored for every instance of the course *offering* (I.e. for every semester a particular course is offered).

However, in 2NF course name must be stored more than just once.

- But not as much as twice.
- Amount depends on the number of times (semesters) an 'average' course in the database has been offered.

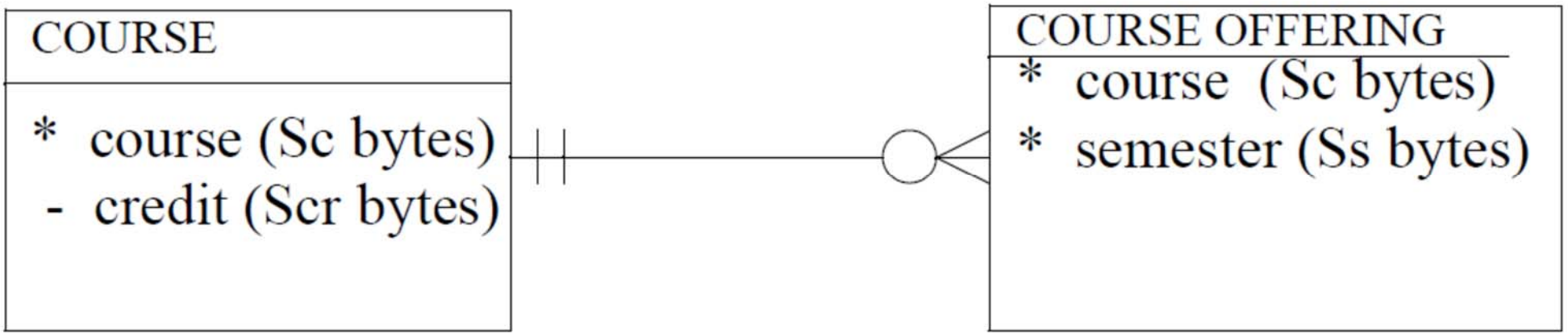
If it is a one to many relationship, would the name only be stored n + 1 times?

We'll assume that the 2NF course offering has no other attributes.

- If it did, their space usage would cancel out of both sides of the calculations.

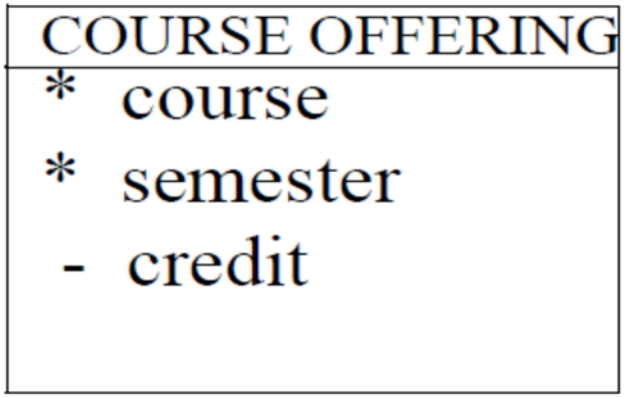
Let's assume that the typical course in the database has been offered N times. **Note that N need not be an integer**, as it is the number of offerings per course averaged over all courses.

We proceed as follows:



In 2NF the average total space for a course is

= (Sc + Scr) + N (Sc + Ss)



For 1NF, the average total space for a course is:

= N (Sc + Ss + Scr)

For 2NF to use less space, we must have:

$(Sc + Scr) + N (Sc + Ss) < N (Sc + Ss + Scr)$

$(Sc + Scr) < N*Scr$

$Sc < (N-1)Scr$

If Sc = 7 bytes, Scr = 2 bytes, and N=5:

$7 < (5-1)2 = 8$

$7 < 8$ which is true!

For this case, 2NF is better.

With a little manipulation, you can show that the range of N for which 2NF is better is: N > (Sc/Scr) + 1

The above types of simple computations are possible on an Cmpt 276 exam.

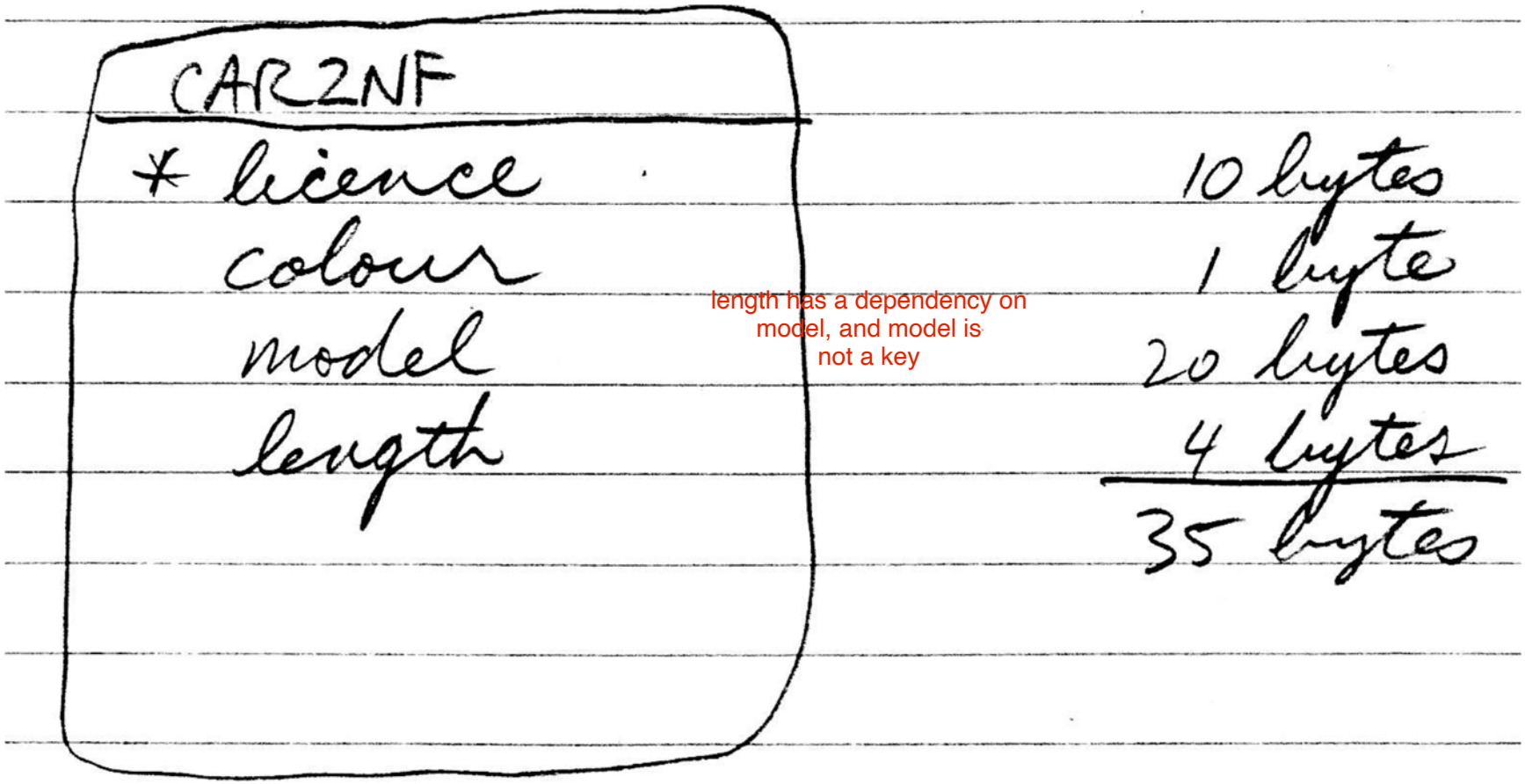
Caution reminder:

- If you multiply *or* divide both sides of an inequality by a negative number, you **must** reverse the direction of the inequality!

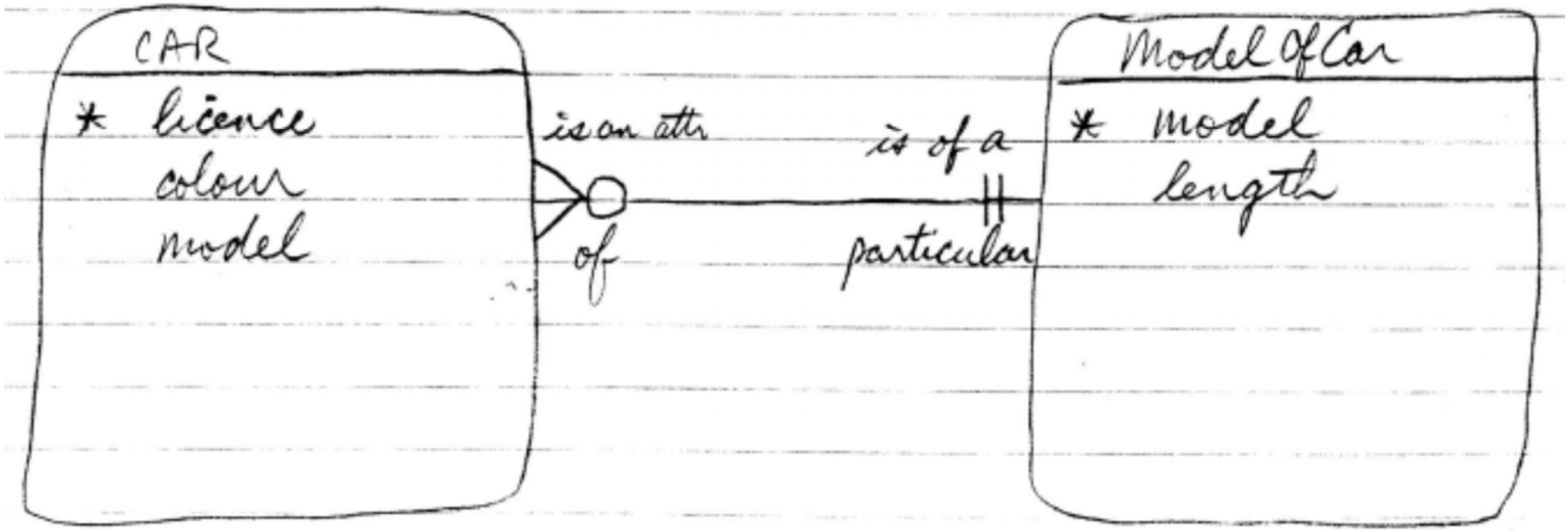
7.3.2 3NF vs. 2NF Calculation

Data in 3NF is good because it *usually, but not always,* saves space and various anomalies.

Consider the following 2NF entity:



Below in 3NF move length of car model to a separate entity. I then needn't be repeated with each car instance.



3NF eases updates since it reduces the number of extra copies of a field like length in 2NF to be updated.

3NF space:

Car		Model of Car	
licence	10 bytes	model	20 bytes
colour	1 byte	length	4 bytes
model	20 bytes		
total	31 bytes	total	24 bytes

Consider 100 cars, with 4 each of 25 different models.

3NF space: (100 x 31)+(25 x 24) = 3700 bytes

In contrast, the 2NF organization has 35 byte records in just 1 file:

2NF space:

100 x 35 = 3500 bytes (saves 200 bytes).

So, **in this particular case and only a few others**, 3NF does not save space!

2NF has advantage only when:

- Models (and particularly the model primary key), are large compared to model 'length' attribute.
  - Model key unfortunately present in both 3NF files.
- And when significant ratio of model entities to car entities (> ¼).
  - So model key duplication in two files unfortunately somewhat significant.

- 2NF has speed advantage of one vs. two file lookup is significant.
  - E.g. Need to find car length *given the licence*.
  - And file access 10,000x slower than RAM.
  - But this not worrisome for human user interfaces!
    - Extra disk lookup is 100x times faster (.01 sec) than human user (1 sec).

And, remember, in the 'less elegant' 2NF:

- Every time someone adds a new car they must:
  - Type in (error prone), or look up the length of that model.

    When looking at the length of the model, could we not have that two different models have the same length?
  - Note, entering a length is far less necessary in 3NF.
- To change a model's length requires changing it in every car instance of that model type.

During design, you will normally be told or choose to optimize for:

- Space,
- Speed, or
- Ease of Data Entry/Consistency.
- Or a balance of these.

In order to do this, you must know:

- Approximate size of each attribute.
- Approximate average time for file record access.
  - Depend on file size and access method:
    - plain unsorted,
    - sorted, or
    - has index tree.
- **Relative** numbers of the objects (e.g. 4 cars/model)

Sometimes helps to solve such problems by saying “Assume 1000 of the most populous objects in the 3NF”.

- Or alternatively 1000 of the least populous entity types.

Research/ask for the ratio of expected populations of them to each other, and employ that.

Starting with *either* a count/estimate of

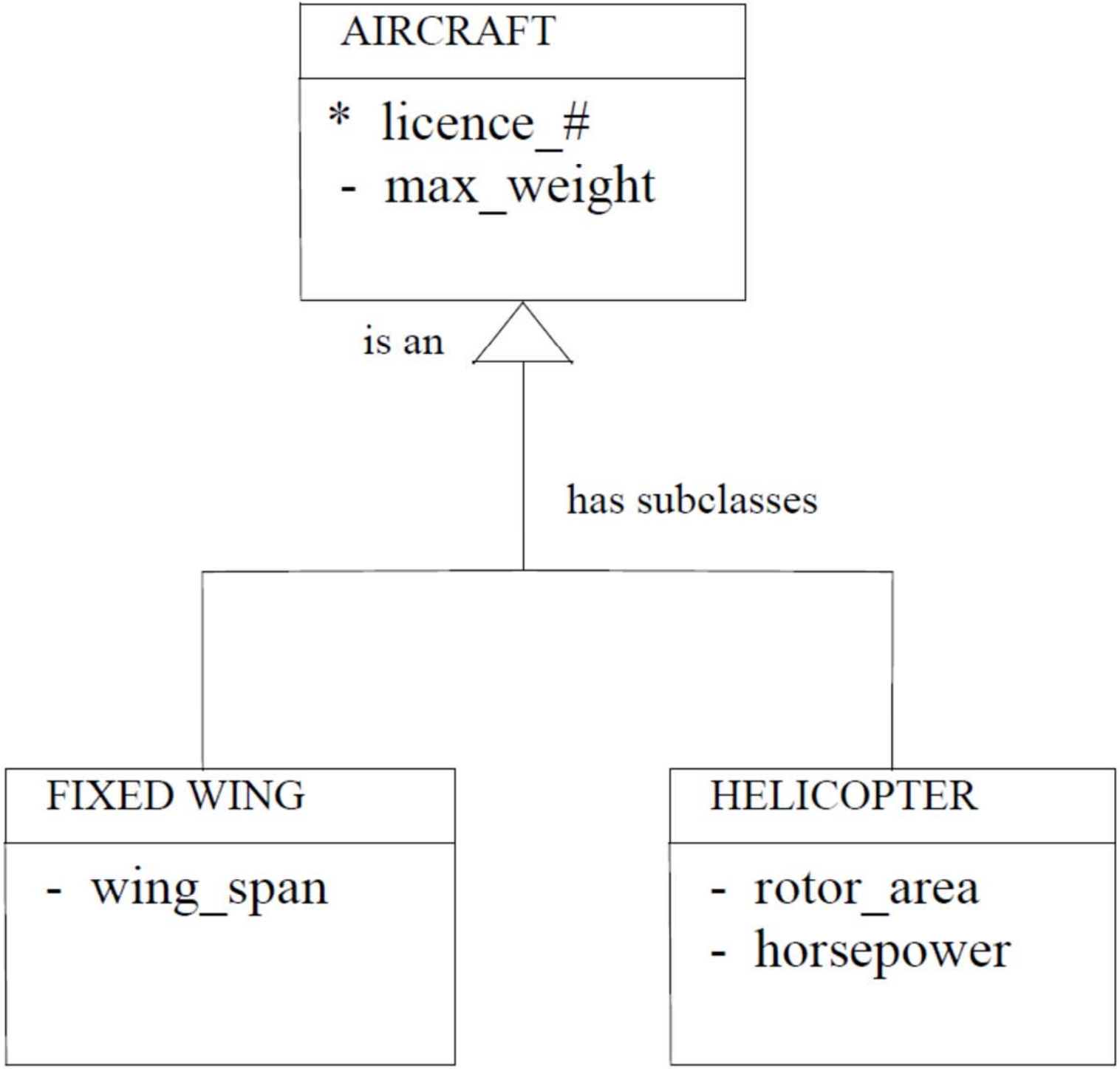
- least, or
- most,

populous entity will give you correct results.

7.4 SUB-CLASS STORAGE IMPLEMENTATIONS

Storing instances of objects from a class hierarchy on disk is necessary but difficult.

- Subclass instances usually occupies a more space than their parents.
- And a different amount of space compared to its siblings and children.


AIRCRAFT
* licence_#
- max_weight
is an
has subclasses
FIXED WING
- wing_span
HELICOPTER
- rotor_area
- horsepower

First, there are three possible classes above.

- Fixed wing aircraft (3 attributes),
- helicopter (4 attributes), and
- ‘generic’ aircraft (only 2 attributes).

If your application has no need of instances of ‘generic’ aircraft, then it’s an **‘abstract’ class**.

- An abstract class extracts (i.e. abstract out) the common attributes and behavior of a family of classes. This is often the case.

7.4.1 Roll Up, Roll Down, or Split

The **roll up** strategy basically suggests storing all instances from the same family of classes in one file.

If fast access is to be obtained by using fixed length records, the records must be long enough for the longest possible instance. In the above case, likely the helicopter is the longest type class.

[This is one use of the C++ **`union`** feature.]

AIRCRAFT
* licence_# - max_weight - wing-span OR rotor_area - horsepower

Obviously an unacceptable strategy unless either:

- classes nearly similar in length, or
- relative number of instances of the short classes expected in the database is minimal. Question: Can you explain why?

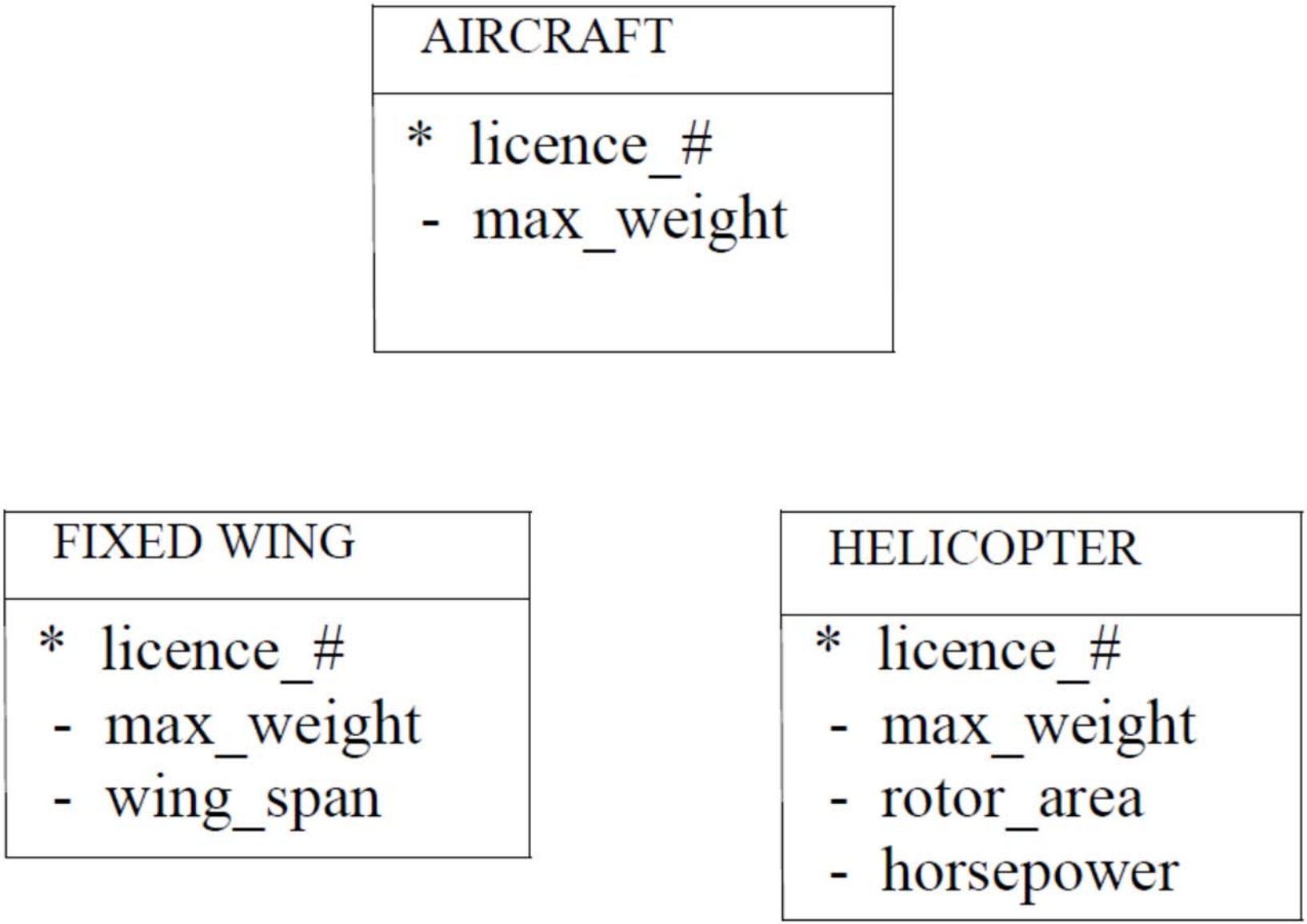
If the number of short instances is low, then you are reducing the amount of memory wasted for storage

Answer: Short instances waste space if stored as records of the size of the biggest class.

The **roll down** strategy is where instances of each class are stored in separate files.

More specifically, all the attributes of the subclasses are stored in individual subclasses' files.

- For the above example, the helicopter file would contain it's full set of attributes.
- Other subclasses would contain their lesser number of attributes.


AIRCRAFT
* licence_#
- max_weight
FIXED WING
* licence_#
- max_weight
- wing_span
HELICOPTER
* licence_#
- max_weight
- rotor_area
- horsepower

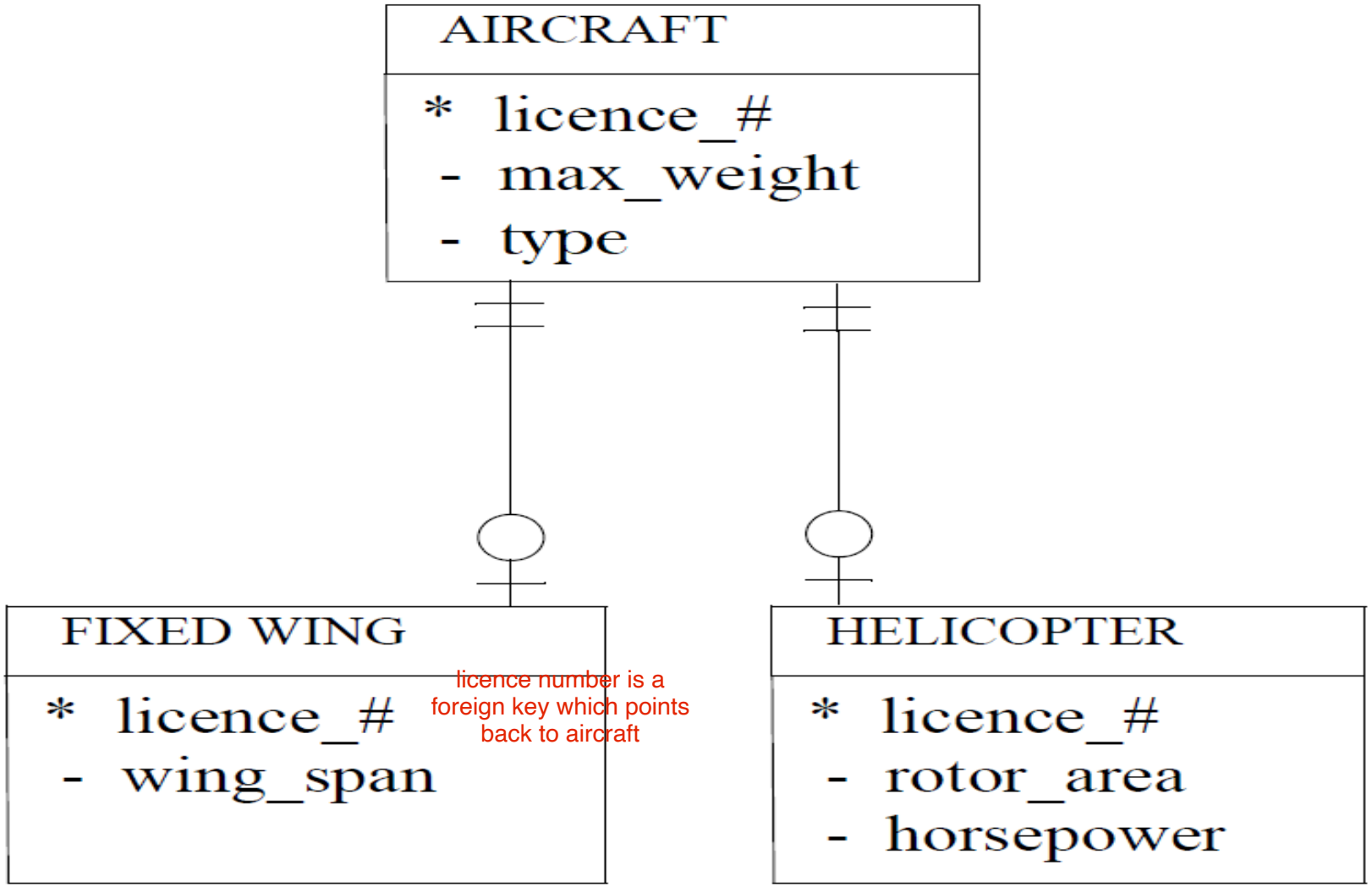
The resultant "implementation" ERD (an ERD showing how the actual storage is implemented) has **no relationships** between them!

Assuming that Aircraft is not abstract, you need an aircraft file. But it wouldn't have any helicopters in it.

This is not a bad storage implementation. However:

- If trying to find an aircraft with a particular licence_#, I **don't know which of 3 files to look in**!
- Essentially no polymorphism.
  - (Not that polymorphism actually exists on disk.)

In a **split** implementation strategy, the licence_# and max_weight are stored for every type of aircraft in the AIRCRAFT aircraft file.



The two extra files store the "*added*" attributes of the subclasses.

The subclasses' files need a key, so each must have licence_#.

To tell whether a given licence_# is for an aircraft, fixed wing aircraft, or helicopter, a "type" attribute has been added to the base class file.

I have draw relationships instead of subclass arcs, as this is an implementation diagram.

Drawbacks of split:

- licence_# wastefully stored in two places.

- type attribute needed. But compared to “roll down” this reduces need to search if licence_# is in more than one file.

7.5 GENERAL DESIGN PRINCIPLES

Design is the phase where you try to transfer your analysis models (ORD, DFD, FSM) into a hierarchy of importing software interfaces.

- E.g. To C++ .h file advertising a class or cohesive group of classes.

Then can choose data structures and algorithms before actually implementing the code.

Design is the most subjective, blurry, and difficult task to so in software engineering.

Though several supposedly-helpful, step-by-step "methodologies" to help the transformation from

Architectural Design to implementation, they at every step require “subjective judgement”!

Unfortunately, judgement is required as only some quantitative measures (e.g. file space trade-offs) are available to compare the worth of two possible competing designs.

Often a cleaner, more maintainable or portable design causes more space or less performance. : (

- How do you trade-off say portability vs. performance?
- How do you measure advantage of portability?
  - Lost time to marked on other platforms?
  - Cost?
  - More errors due to port.

To gain subjective judgement, let's discuss design principles.

- Abstraction:
  - Hierarchical abstraction
  - Control abstraction
  - Data abstraction
- Cohesion
- Coupling
- Encapsulation
- Design (info) Hiding
- Scope of Effect/Control
- Fan-in

And later:

- Top down vs. bottom up design
- Stepwise Refinement.

7.5.1 Design Decomposition

Let's examine principles used in design decomposition and trade-off decisions.

Recall that decomposition is necessary:

- So resultant pieces small enough and independent enough that they can be assigned to different people to design separately.
- And pieces are small and 'independent' enough that they're modifiable/correctable during the design and coding without unduly affecting the rest of the subsystems.

Separateness/independence requires an architectural structure that has **low and well-defined coupling** between components.

Control information: could be boolean flags

Avoid passing excessive control information and data between the components.

Avoid shared assumptions amongst the components. coupling?

- Put pieces that share the same assumptions (say about data structure) in the same subsystem, module, or class!

Localization of assumptions has been called **encapsulation**.

Inside the encapsulation there's strong/high **cohesion**.

Cohesion:
the components
that are there
have good reason
to be there

7.5.2 Design Hiding

Design Hiding:

- Is deceptively called “information hiding” by many textbooks.
  - E.g. Is **`File`** a type name for a:
    - (functionless) struct/record,
    - object,
    - typename for pointer to record or object?

    Do you even need to know?
- Is a heuristic design principle where inner details/implementation of one component that don’t need to be known by other components, are hidden/inaccessible.

- Cf. Private members of a class, but also applies at a larger scale.
- Other components ask the first for services, and trusts the first is implemented properly to handle this.
- This prevents 'brittle' designs.
- Allows the internal design of the first component to be changed with little likelihood of other subsystems/modules/classes needing changing.
- Since one of Tront's Principles is to expect change, hiding internal detail is a worthwhile goal.

Enforcing hiding could be:

- Making so invisible to other programmers.
  - E.g. Not visible at all in C++ .h file.

- More commonly, not usable by other programmers.
  - E.g. **`private`**
  - Only need to make public the *subset* of functionality that other programmers need to:
    - Manipulate/employ the entity, and if necessary:
    - output the entity (e.g. C++ operator<<(), or some helper (non-member or member) function that will print the object (e.g. print a date).
    - test two for equality (e.g. C++ operator==() or some equal() function)
    - assign one to another (e.g. C++ operator=() or other equivalent assignment function).

    - Copy one to a new one (e.g. C++ copy constructor or Java clone function).

- The point is to enforce hiding so other programmers can't couple to the class's internal details, but still allow other programmers to employ your class.

Design (information) Hiding can be used during top-down design to delay resolution of structural details.

Top level design allows you to leave the implementation of components for later

- And decouple those details from the higher level design.

What should be hidden:

- Difficult design decisions that are likely to change.
- Controllers that sequence activity generally.
  - E.g. Useful to hide whether synchronous versus (perhaps later) employ multi-threaded, multi-core "parallel" programming.
    - Most CPU chips are multi-core, but most programs don't lever this for performance.
- Character code representations (e.g. ASCII, Unicode, EBCDIC).
- Bit manipulation – since often used for machine-dependent tweeking.

- Class and module implementation for which there are several choices of complexity, performance, memory trade-offs.

7.5.3 Encapsulation

Encapsulation is putting a **fence** around several design aspects for one subsystem, module, class, or bounded unit.

- Data types, algorithms that work on data, communication protocols and their packet types, should be placed together.
- Good because if some assumption shared by those things are changed, only the things in that module need be reprogrammed.

Fence could be a:

- class,
- .cpp module,
- subsystem, or
- program (if there are interacting programs).

Encapsulation doesn't necessarily mean a solid fence.

- Could be see-through.
- But it means putting all the stuff related to a design aspect in one place (class, module, or program).

7.5.4 Cohesion

Cohesion is a heuristic measure of how well the various elements/things in an software element belong together. There are two elements of cohesion:

Remember purity and completion

- A component has all the functions and data associated with a particular implementation issue in it, and
- Also, no other extraneous things (consts, types, vars, functions, tasks) are in the same **encapsulation container**. E.g. In same class.

I.e. Completeness and purity.

Purity:
only things related to what you are designing

A stack class is cohesive if it embraces:

- The behavior of a stack (purpose)
- Everything about a stack (completeness)
- Nothing but stack stuff (purity).

Strong cohesion implies later modifications will only require one or few design entities.

Types of Cohesion (weakest to strongest)

(NOTE: Do not memorize):

- None or coincidental cohesion.
- Logical cohesion: Similar in nature, like all input functions.
- Temporal: Must/will be executed at same time, or in a prescribed time order.
- Procedural: When elements must be executed (or executed upon) in a specified order which may vary depending on circumstances.
  - Cf. Builder and Chain of Responsibility patterns.
- Communications: All elements access or share the same data.

- Sequential: The output of one element is the input of the next.
- Functional: All are necessary for the existence and function of the software element.

Comment: The aspects of a cohesive subsystem/module/class usually "share some assumption(s)" about the data representation/organization, or functioning.

E.g. Stack class.

7.5.5 Coupling

Coupling is a measure of how intertwined two elements/components are by their

- shared assumptions,
- shared data,
- share function calling dependencies.

Coupling is just a measure of how much interdependency there is between two elements/components of a design.

Strong coupling to outside is poor.

Strongly coupled modules or classes share *many detailed* assumptions about one and other.

Obviously, modules must rely on the services of each other. But in a good design this is minimized and pure.

- Few shared data types.
- Few shared data structures and files.
- *One class shouldn't know how to traverse another's structures, or even know where they are!*
- One module doesn't know much about the decisions and ordering of another.
- A service provides service to whomever calls it.
  - Weird example: tan(angle) doesn't care about the name of the actual callers parameter.

- A client program element shouldn't care how (stack as array or list) or who (laser vs. inkjet printer) provides a service.

Low coupling promotes code re-use and easy re-arrangement.

Types of Coupling: (worst to best)

(Do not memorize):

- Content coupling: One module makes use of or modifies data or control info of another.
- Common coupling: Modules share a common data area, not in either. E.g. Global variables.

  Shared global variables in a multi-threaded program are a concurrent programming nightmare!
- External coupling: Coupling to external device characteristics, network protocols and packet formats.

- Control coupling: When control flags (e.g. Booleans) are passed that modify decisions or sequencing in another module.
- Stamp coupling: Shared data record “types” or structures passed as parameters.
  - Except function caller must share knowledge of types to be passed. But don’t assume the called function will store your data in the incoming type.
  - E.g. Pass in an string date, which is broken up and stored as 3 integers. Caller shouldn’t have knowledge of how stored.
- Data coupling: shared simple types only (e.g. int, float).

Comment: Weakly coupled systems interact only with abstract views of one and other.

Comment: If only an abstract view is exposed, weak coupling is enforced by design hiding.

High cohesion promotes weak coupling as all things about a design are contained in one place.

Incomplete cohesion implies extra coupling to rest of needed aspects.

- One aspect of “spaghetti” code. Tangled.

Note: Code which starts fairly cohesive, encapsulated, and uncoupled, can after much maintenance become coupled.

- And hard to do further maintenance on.
- Especially happens in language without enforceable design hiding (e.g. private) and strong typing.
- Easiest way to patch such code is to violate anti-coupling principles.
- But then hard to know if changing something will break something else elsewhere.
- Result is sometimes a whole sub-system must be discarded and re-written from scratch.

If a design:

- cohesively encapsulates things together in a wise architectural layout, and
- forcefully hides the design issues and assumptions made by individual components,

then each component will have a nice simple abstract nature.

And even if used in a design in many ways, its design will have relatively **low coupling** with the rest of the components.

- This will prevent the architecture from being brittle in the face of change, and

- make the components (like integrated circuit chips) re-usable in many situations.

It also allows us to replace a component with a different one that has the same interface.

7.5.6 Abstraction

Abstraction is the most powerful technique humans have to handle complexity.

We've already seen two forms of abstraction:

- hierarchical classification and
- hierarchical composition.

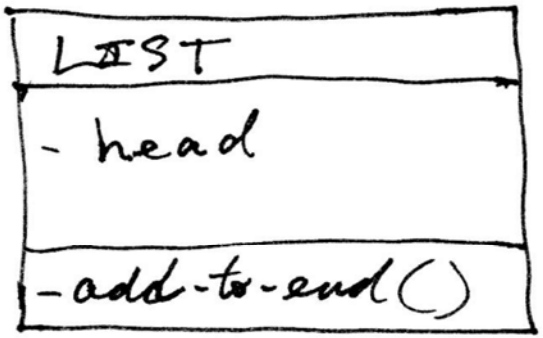
Abstraction is basically just being able to refer to

- the more general or more composed (i.e. car) thing,
- without needing to reference the subclasses or many parts of a car.

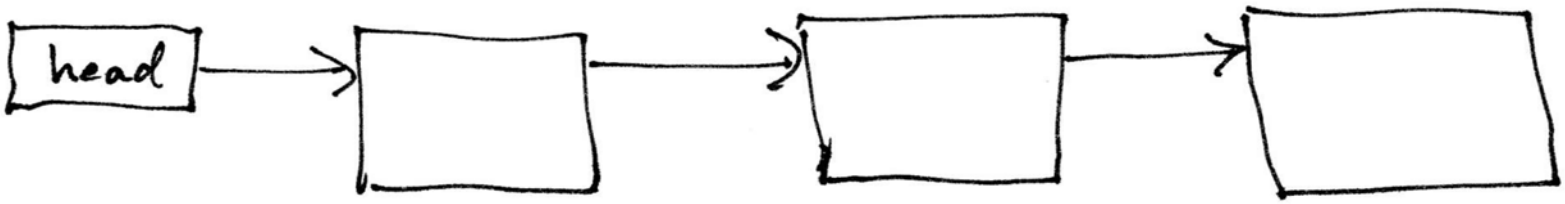
7.5.7 Hierarchical Abstraction

Hierarchical abstraction allows us to unclutter our mind of the lower level details if we just need to concentrate on the diagrams and relationships at a higher level that the component appears in.

- E.g. #1: Car rather than list of parts.
- E.g. #2: List rather than depiction of its nodes.

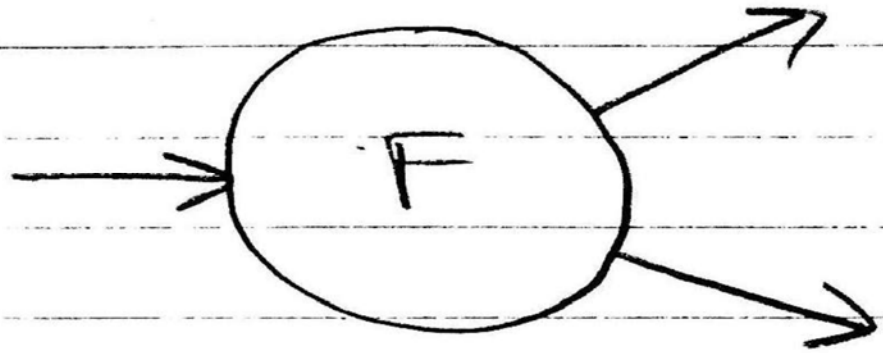


Rather than:

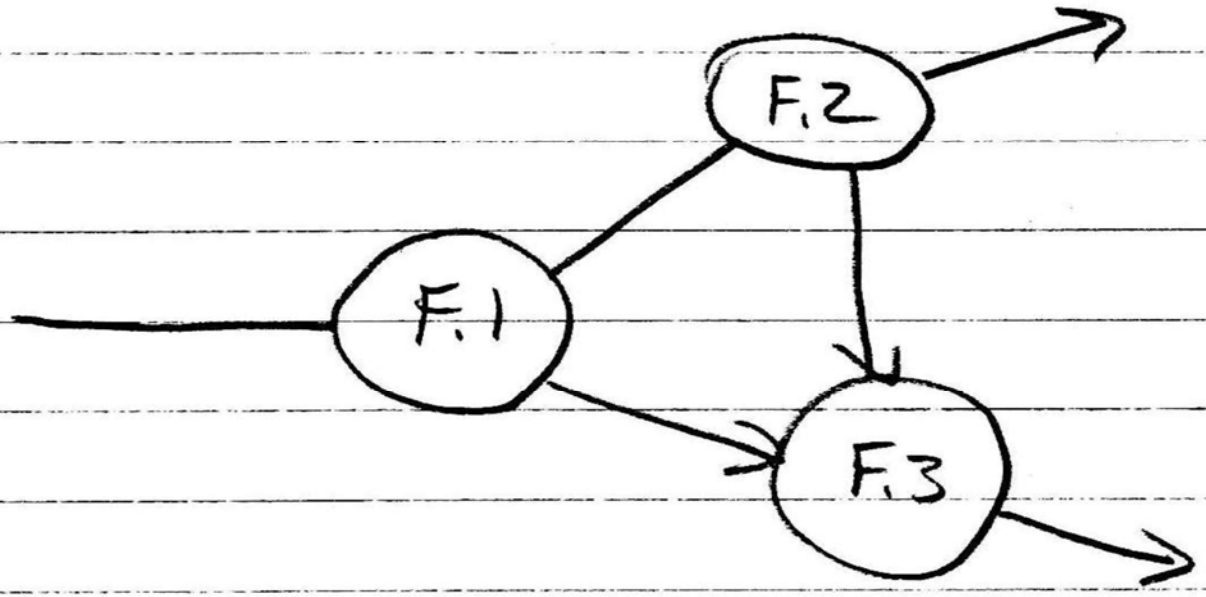


For data flow diagrams (DFDs):

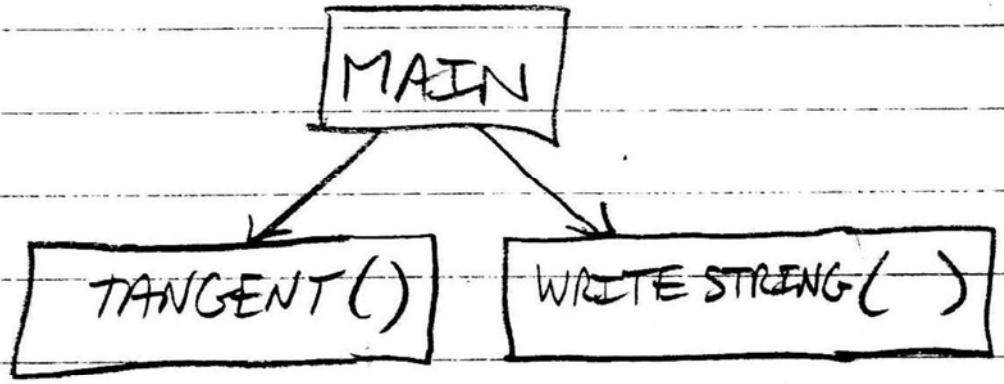
e.g. #3)



RATHER THAN

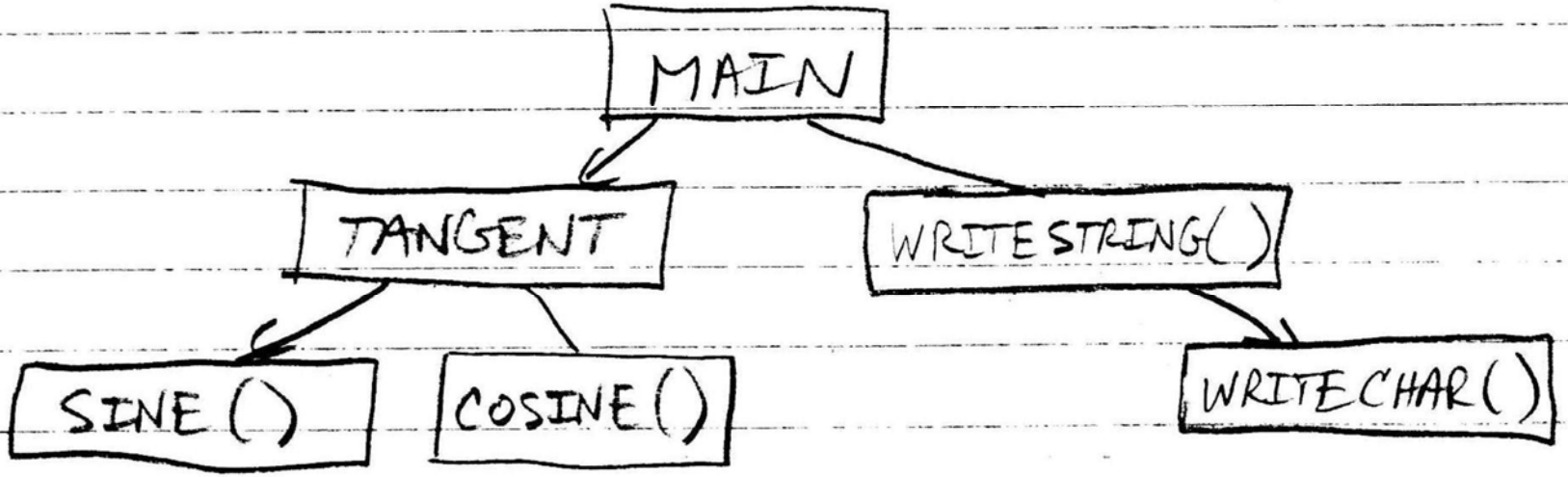


eg. #4)



CALL STRUCTURE CHART

RATHER THAN THE INDIVIDUAL ALGORITHMS AND/OR LOWER LEVEL CALLS:



Hierarchical abstraction supports human characteristic of only “critically” dealing with about 7 things at once.

Several layers of hierarchical abstractions leads to a tree of abstractions.

- Subclass and sub-subclasses.
- Programming objects that contain component objects, which contain sub-component objects.

Hierarchical decomposition of a problem is a very powerful technique referred to as either:

- Step-wise refinement
- Top-down design
- Functional decomposition

- Layered design

Each level represents an abstracted view of the lower levels.

Abstraction frees the designer to think in terms of more simple/familiar/useful objects or operations/models-of-reality.

7.5.8 Data Abstraction

Data Abstraction is much easier in modern programming languages that allow you to define your own types and records.

e.g. enum Season {spring, summer, fall, winter}

e.g. File myFile; myFile.open();

Potential midterm question:
what is data abstraction and give an example

Data abstraction advantages:

- Design (info) hiding – don't know implementation of type File.
- Simplicity – only need export degree of info needed.
- Abstraction – can deal with File at higher level (might be a record, an OS control block, or in Apple terminology a "handle").

7.5.9 Control Abstraction

Control Abstraction is decision-making, especially in the sense of deciding:

- if,
- when, and in
- what order

to do operations in.

If control of these can be abstracted so clients need not view the particulars, then you have control abstraction.

E.g. Object-oriented patterns called “Chain of Responsibility” and “Builder”.

Control abstraction is also called “behavioral” abstraction.

E.g. Simulating concurrency when it doesn’t really exist (e.g. multi-user computer system seems to each client as completely devoted to him/her).

E.g. Burying or hiding actual concurrency (so client needing concern himself with the messy timing of interacting concurrent system).

E.g. Disk arm motion scheduling in a multi-user OS doing seek/read for different users simultaneously, **not** in FIFO order.

- Instead, use elevator scheduling. Which moves most people by non-FIFO, but instead up and down sweeps (only changing direction when no need to go further in that direction).

E.g. Print spooling subsystem. Program things it's writing to the printer right now, but printer might be busy doing previous job.

Leaving decisions to lower level is also control abstraction. E.g.

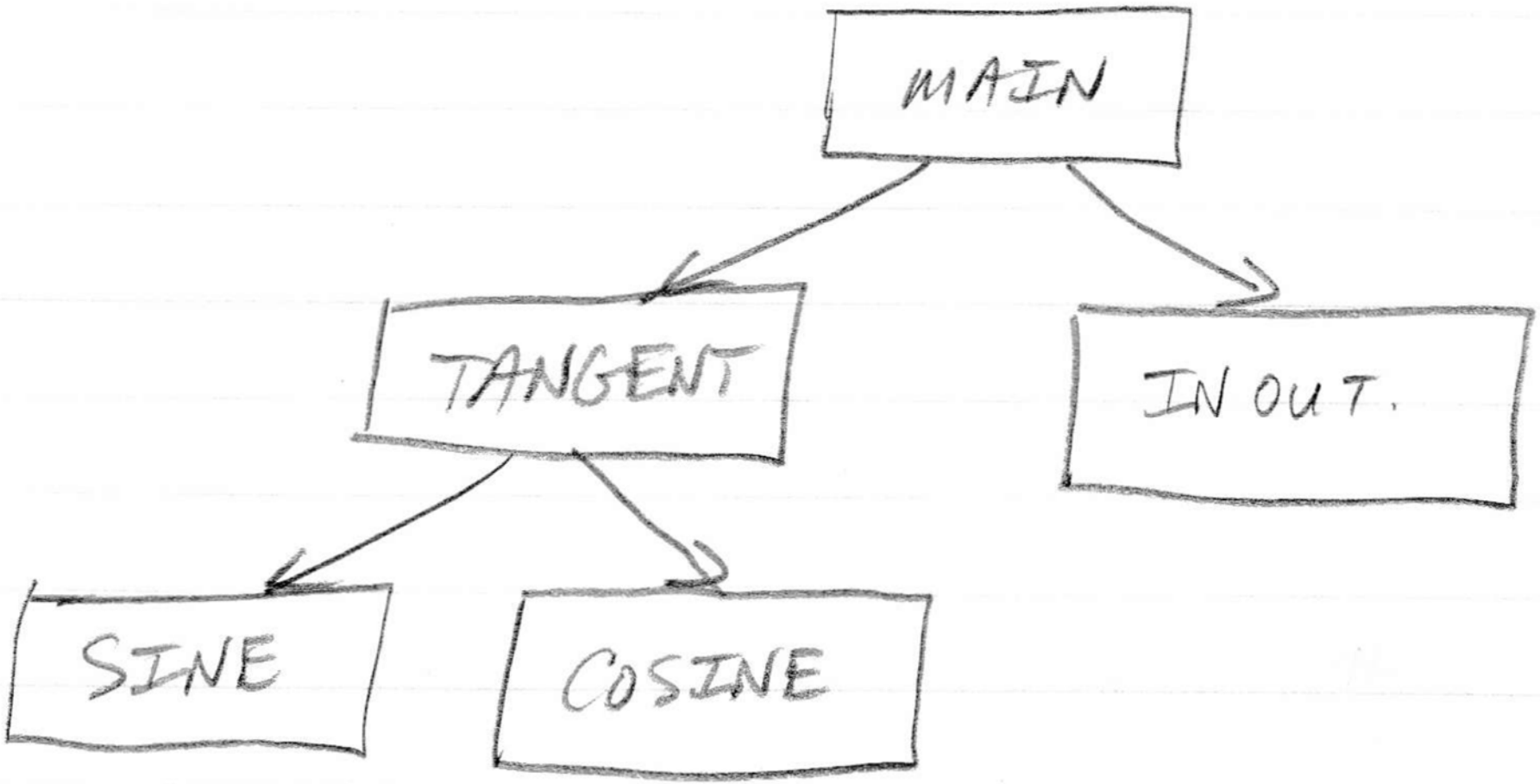
- The C++ “exception” mechanism is good for leaving detection of low level events to lower level libraries.

  But allows libraries to force higher level code to handle lower level events and errors, even if a higher level code authors are too lazy to check function return codes.

7.5.10 Scope/Effect of Control

Scope of Control is the set of all functions subordinate in a call structure chart.

CALL STRUCTURE CHART!

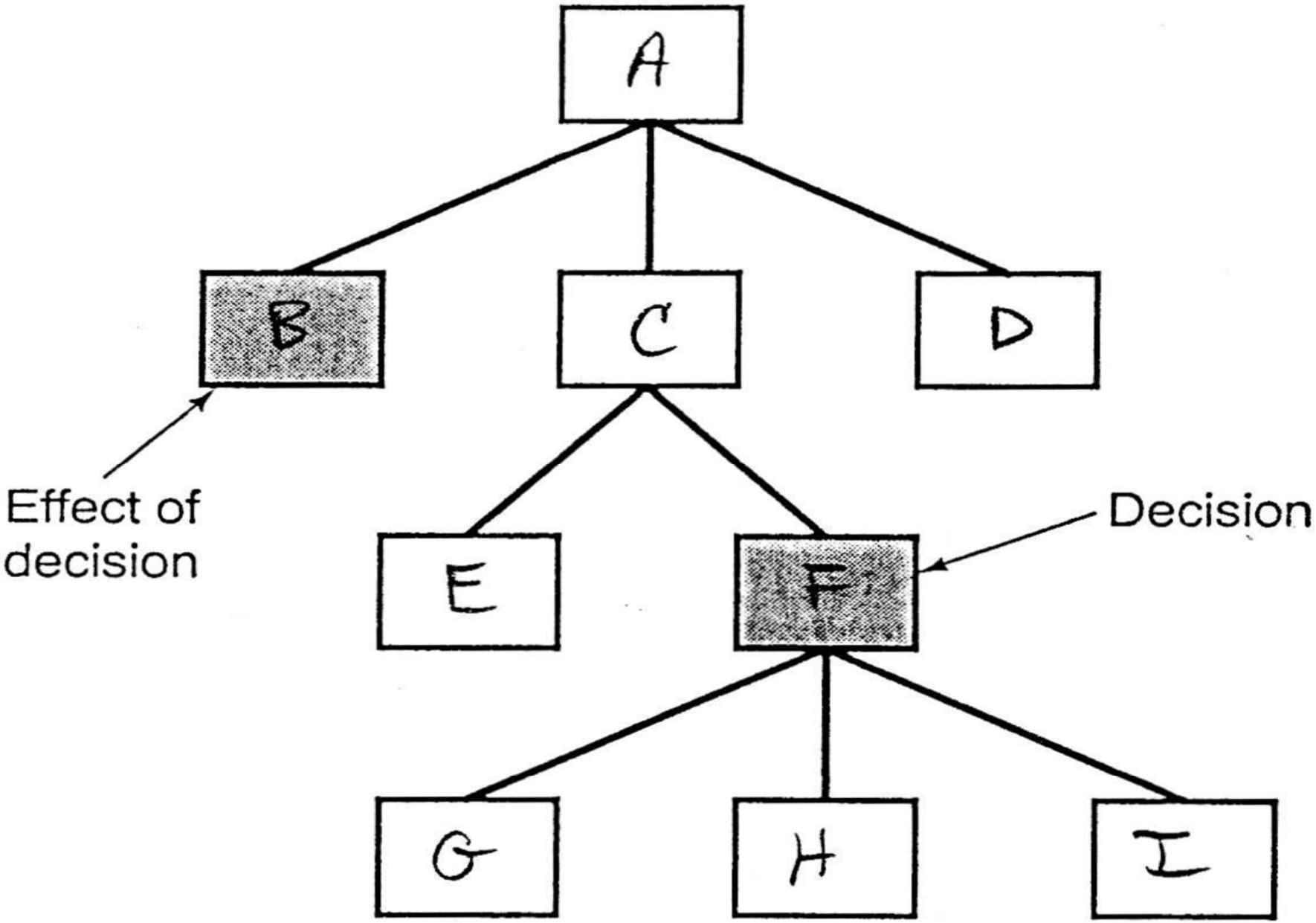


Scope of control of Tangent function is Tangent+ Sine + Cosine.

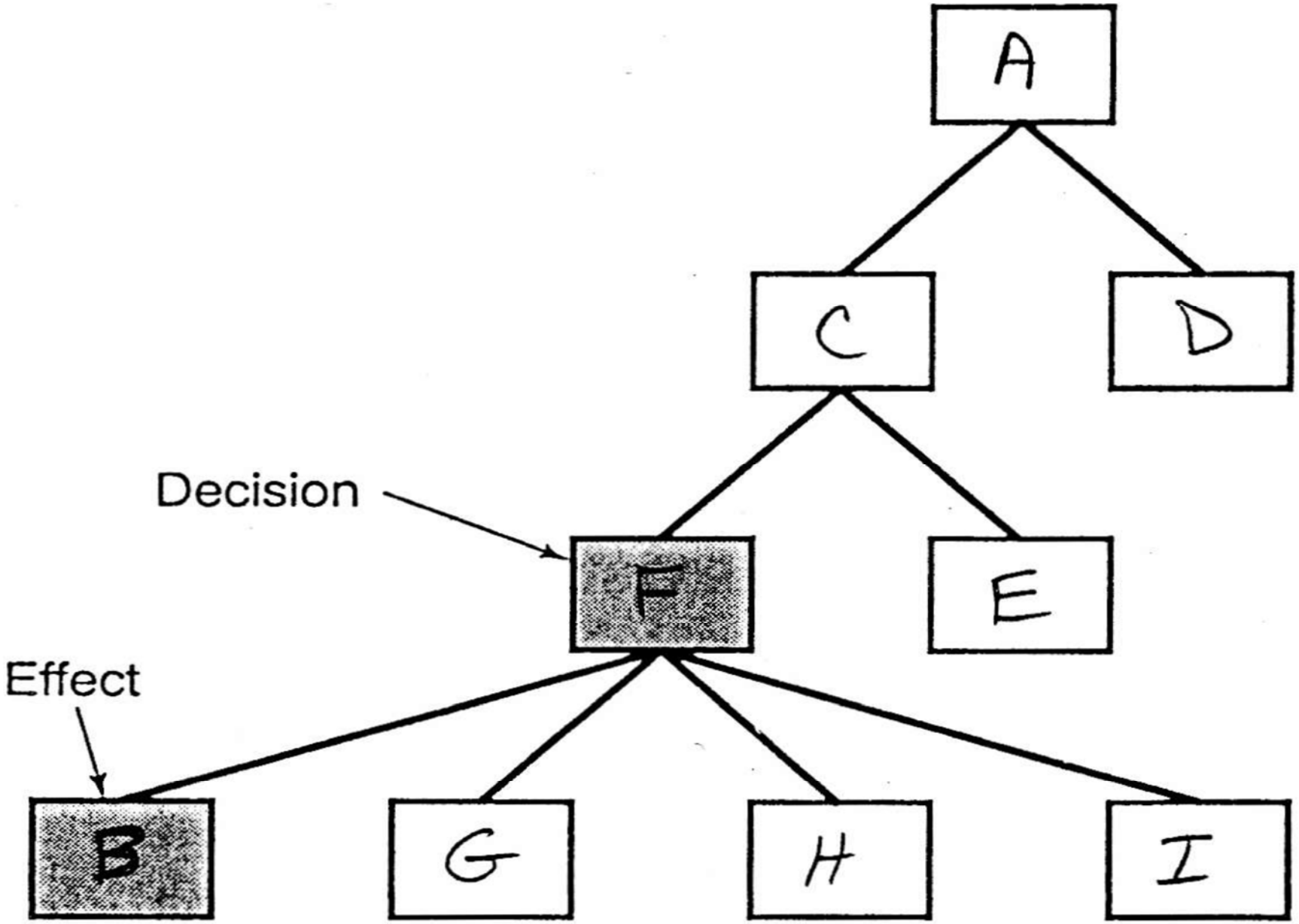
Scope of Effect of a decision is set of all functions or subsystems affected by the outcome of a decision.

Good Principle: Simpler, more maintainable systems result when the **scope of effect of a decision is inside the scope of control** of the function making the decision.

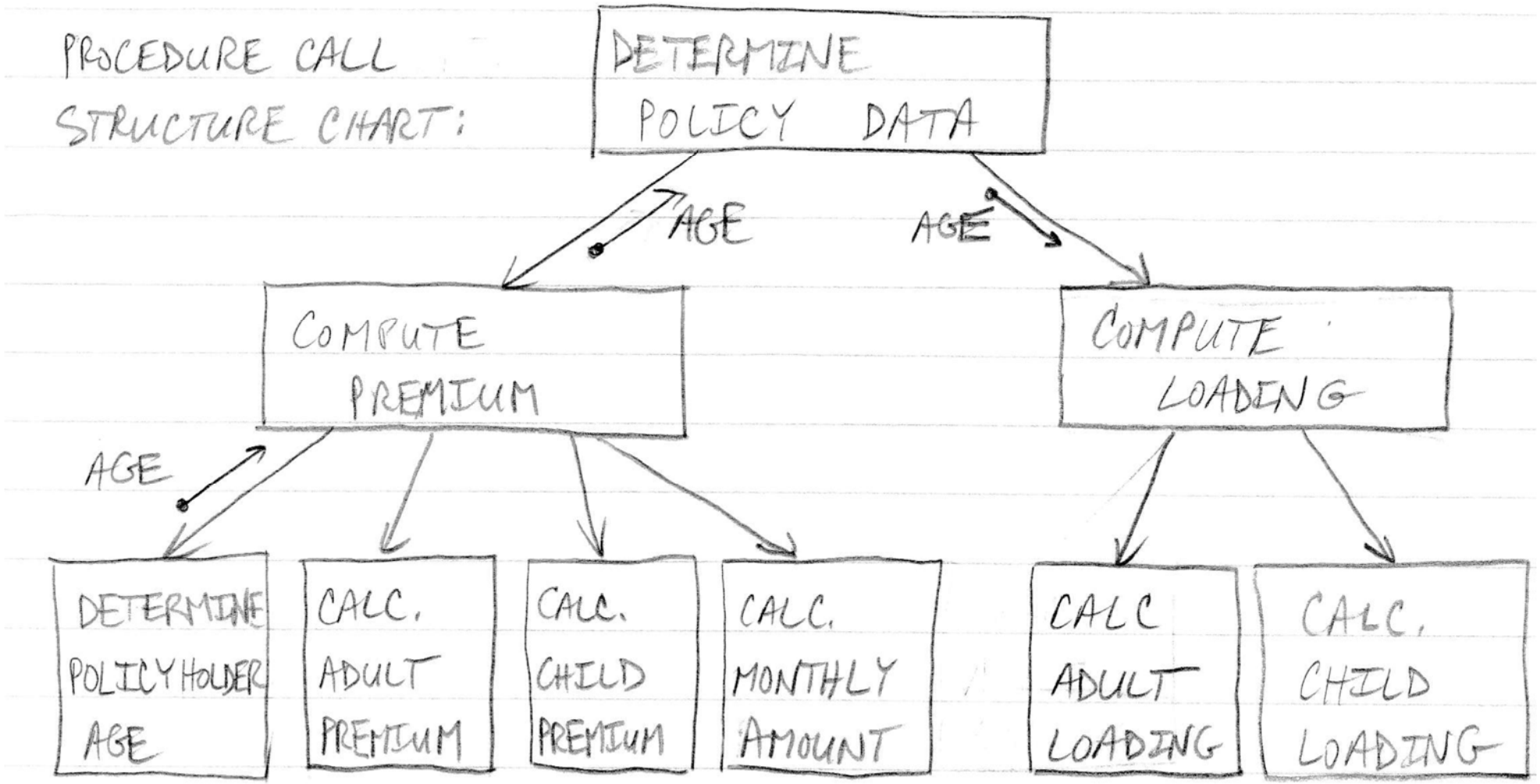
Violation of the above principle:



Re-arrangement to comply with above principle.

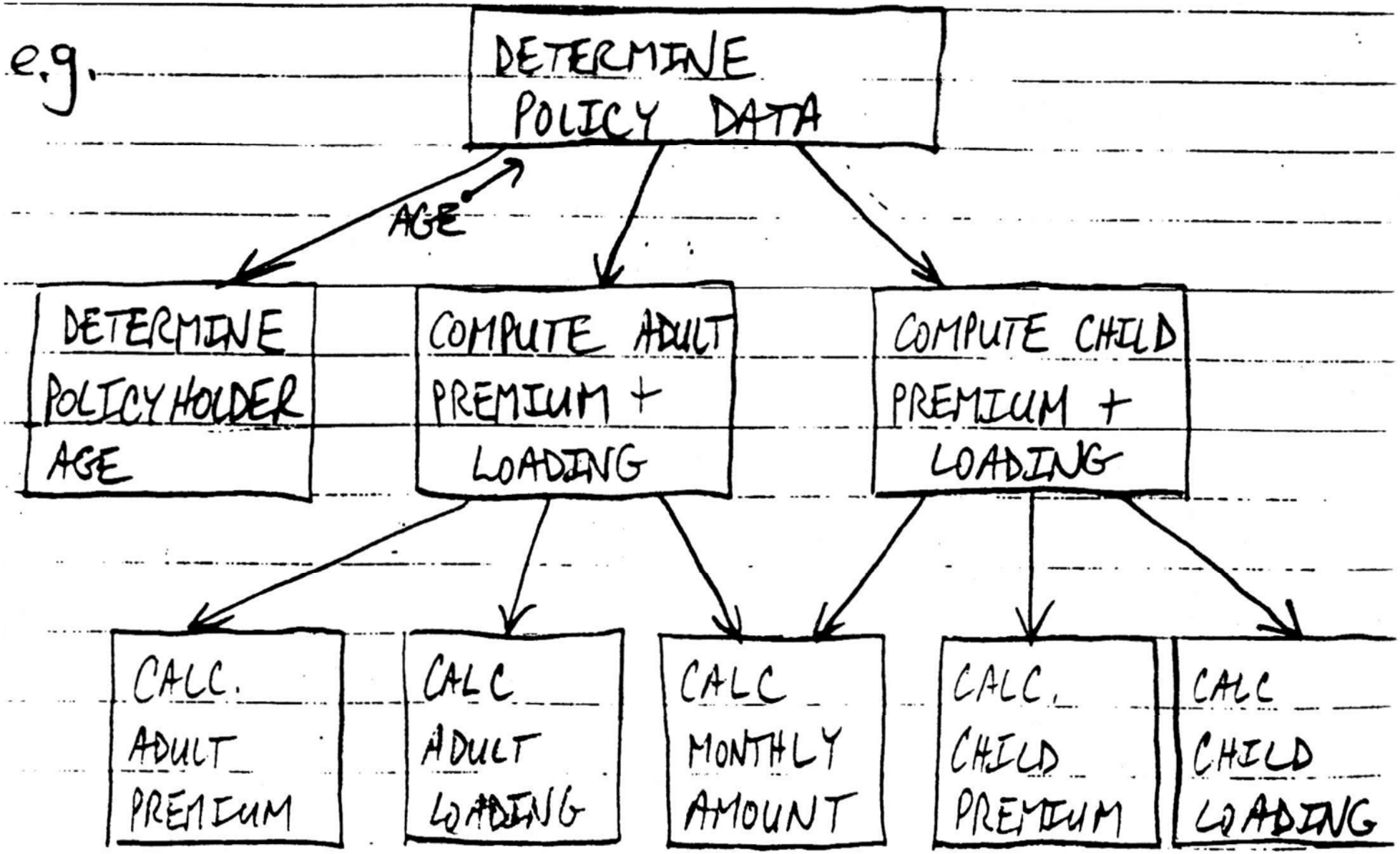


Insurance Calculation Example:



The above layout of control is poor, with age dependencies in both left and right branches.

Here's a better alternative:



Result:

- Age of adulthood decision is made only once.
- Scope of effect of adulthood decision is within the scope of control of the decision-making function/subsystem.
- Less control coupling.

Note: In original diagram, an “adulthood” boolean flag could have been determined when age was obtained.

- This would seem to contain the child/adult decision.

But would have had to:

- Pass flag *two* levels up, then down the other major call branch.
- Or left in a global variable!

Note: Passing a control variable only *one level up* is ok, as it’s not uncommon to delegate such a decision chore to a one-step-lower function.

Be careful as the first design you think of, or the way the design gradually evolves into, is:

- not the only possible design,
- and perhaps not the best.

Taking the action to conceive/think about alternative designs, and choosing the best is what "design" is all about.

Sometimes programs can be re-structured with a minor change, re-arrangement, extra function parameter, or extra variable or function.

Book:
Refactoring, by Martin Fowler

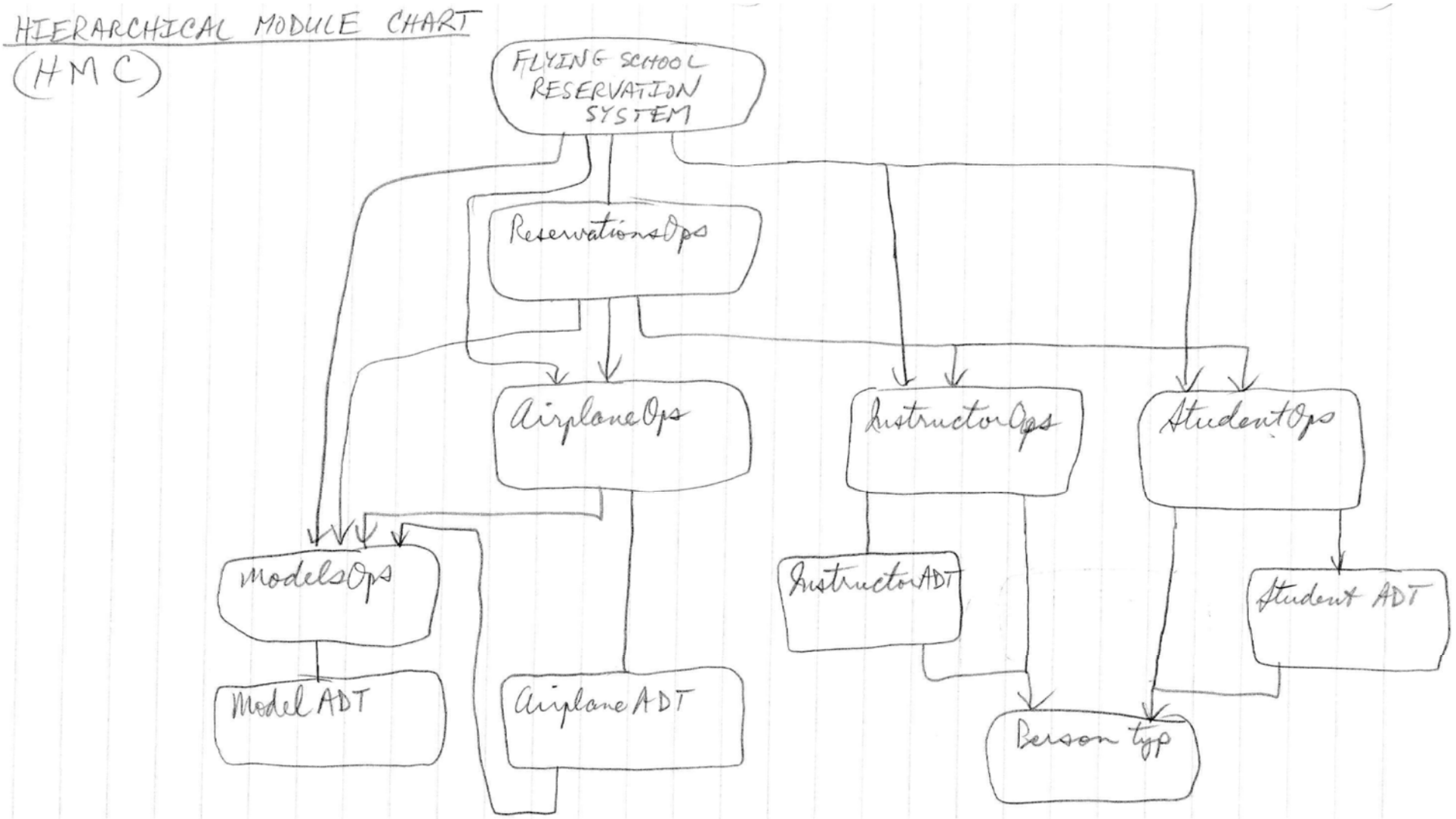
The above scope of control/effect concepts can also be used at the class or module level.

- Not just the function call level.

Consider a group of classes that call each other with a variety of specific function calls.

Even though one class might make calls to many different functions in another class, they are coupled at a slightly coarser level in being grossly dependent on the several functions of another class or module.

Consider this diagram to represent a bunch of C modules that depend on each other (but could also represent classes that call each other).

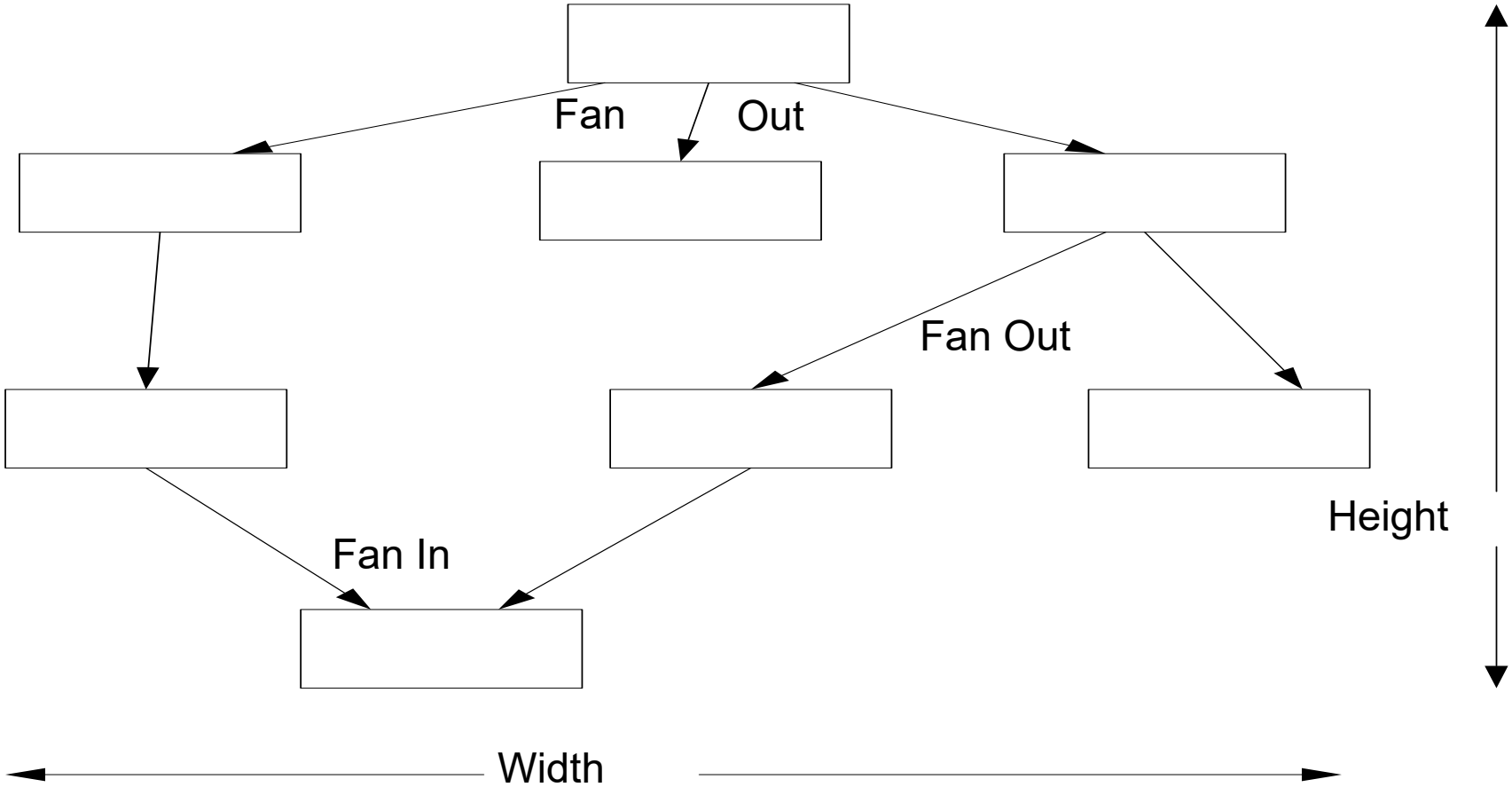


Each line in the above could represent calls to several differently-named functions in the target module/class.

And you can apply the scope of control/effect principle to at this higher level of abstraction too.

7.5.11 Fan-in/Fan-out

Call charts, and almost any connected graph, often have characteristics called **fan-in and fan-out**.



Know what the terms fan-out, fan-in, height, and width of fan-out mean!

Classically, the boxes were programming language functions.

More recently, the diagrams can illustrate the calling or interaction relationship between:

- Subsystems,
- Modules, or
- Classes.

Note: Because of fan-in, such charts are not actually trees.

Fan-out is common at the top.

Fan-in can occur at the bottom.

- Fan-in good. Illustrates that good, low level functions for subsystems have been written, which are re-used by different parts of a program.
- Bottom fan-in also good because as it allows system to be easily ported to other OS, libraries, or hardware platform.
  - Likely only a few low level functions or modules need to be re-coded/adapted for the new supporting system!

7.6 Design Specifications

More in on writing a Design Spec will be given out with Assignment #3.

As an aside: There used to be design spec samples in the "Sample Document" resource links at one of my favorite author's (Steve McConnell, author of "Code Complete") website . However, this path no longer exists.

Nonetheless, there is new fabulous material on many modern software engineering topics there:

https://www.construx.com/resources/topics/

7.7 Design Reviews

Before coding should start, for quality assurance reasons, a design review should take place. Sometimes on a big project there are two design reviews:

- An Architectural (or high level) Design review
- A Detailed (or low level) Design review.

Generally, these reviews bring out questions or problems that are then addressed and fixed.

- A later revision of the Design Specification is then signed by several managers to signal approval of the design and allow the project to go forward toward implementation.

We'll cover design reviews later in the course, but you may find it interesting to peruse review checklists for various stages (requirements, architectural design, low level design, etc.) from [Weigers98] that are posted as a Section 7 resource.

7.8 References

[Booch94] “Object Oriented Analysis and Design, 2nd ed.” by Grady Booch, Benjamin/Cummings, 1994.

[McConnell93] “Code Complete: A Practical Handbook of Software Construction, Microsoft Press, 1993.

[Pressman97] “Software Engineering, a Practitioner’s Approach, 4th ed.”, by Roger Pressman, McGraw-Hill, 1997.

[Sommerville96] “Software Engineering, 5th ed.” by Ian Sommerville, Addison-Wesley, 1996.

[Weigers98] “Software Technical Reviews” by Karl Wiegers, course notes from a course put on in Vancouver 94/4/17 by the Software Productivity Center.

7.9 Appendix A – Black Box Testing

There are certain kinds of tests than can be planned even before the code is written, at least if the User Manual has already been written.

In the past, if I ask students to write 'black box' tests as part of Assignment #4, they write the code first followed by the tests.

- This biases how they write the tests, and whether the tests actually check what the user manual says (vs. what the code does).
- So I will explain 'black box' tests, and ask you to do them in an even earlier assignment.

This appendix contains a very short introduction to ‘Black Box’ testing. Note that there exist whole books on this subject.

Definition: Black Box Testing

- Testing that can be done without knowing the implementation details.
- Cf. Unit testing, branch coverage testing, etc.

In user manual, emphasize benefit of scrolling console
You do not need to
submit a revised requirements spec until assignment 3
Russell was looking for user interface guidance in requirements spec
Focus on printing out lists the customer can read for data entry

Generally, if you have read the user manual, you should be able to write black box tests.

- Not generally written by users, but instead by Quality Assurance staff or software engineers.
- However, do not need expertise of software engineers who are actually coding the project to write black box tests.

7.9.1 Test Cases

A test case is composed of 4 parts:

1. What is being tested, and the class/type of test.
2. The EXACT hardware, file, and program state (perhaps half way through execution) prior to the test. E.g. example input data.
3. The EXACT steps of the test.
   - Includes intermediate prompts and responses, if a user-oriented test.
4. The EXACT expected final results.
   - Not a description, but an example of screen or printer output.
   - Or example of final file contents.

In more detail:

1. The goal of testing is to *find* errors.
   - Not try and prove there are none. This is impossible to prove.
   - Although for a finite state machine, can prove this in a finite number of tests.
   - Generally, find test writers who love to stress the software. Software engineers sometimes consider them 'evil' in their ability to find bugs.

Functional Test Subtypes:

- Nominal inputs,
- boundary inputs (assuming inputs have a documented range),
- special case inputs (e.g. 1x1 matrix, identity matrix, zero matrix),
- default cases work,
- various permissible order of operations work.

Performance Test Subtypes:

- Throughput
- Response time
- Above while under heavy processing load.
- Database access time
- Network delays
- %CPU time spent in each function, class, module, or thread.

Stress Test Types:

- Invalid data
- Overload of:
  - Users
  - Jobs
  - matrix size
  - memory
  - rate of network message arrival
- User operator bad sequence of operation.
- Hardware or network failure
  - E.g. power failure, disk/thumbdrive full, Wifi lost.

Test how the system fail for the above?

- Slow decay?
- Graceful error message?
- Crash?
- User interaction freezes.
- User can’t exit a query/response loop?

Other Black Box Types:

- Security tests
- Installation tests.
- Back-up tests. (Do restored back-ups actually work?)
- Test of user manual by novice users.
- Etc.

2. Pre-Conditions:
   - Exact test input for the case. E.g. The exact state of any input files or running state files.
   - The state of the program. I.e. Where along in execution is it? And possibly which path (of several possible paths?) did it take to get there before test is to begin.
   - Works on hard disk, cloud, and thumb drive?
   - What size or speed of disk?

3. Steps:
   - Exact operations by user, keypress-by-keypress, to execute the test.
   - Along with intermediate system responses.
   - Write so a secretary or Co-op student unfamiliar with the product could do the testing.
     - Useful as will likely need to be run several/many times on future releases of the program.
     - So doesn't need to steal times from valuable development programmers.

4. Example Results:

Why pretty output?

- Not just a narrative description, but exact expected pretty output. For example, if a printed output, with exact expected table headings, date, page #, etc.
  - So tester not biased by there own expectations.
- Sometimes useful to state what should not happen.

See for example a black box functional test case from a past Cmpt 275 student project.

- Posted as a reference document associated with Assignment #3.
- Comment: The posted test case is for testing a "rather long" complex use case.