9. Low Level Design and Unit Testing

Last Mod: 2024-06-22 © 2024 Russ Tront

Low level design is also called “Detailed Design”. We’ll discuss this partly in this section, and partly in the next.

But let’s cover Unit Testing first.

Required Readings: None

Optional References: See the references sub-section.

Table of Contents

9.1 Unit Testing

“**Unit Testing**” is a term used to describe testing of a *single compilation unit*.

- I.e. A module.
- In C++, a .cpp file.
  - And essentially also it’s .h file that *advertises* the .cpp file’s interface.
  - The .h is both an advertisement to calling programmers, and also to the calling compiler to check for correctness.
    - So don’t permit, say, passing a string to a parameter that is **`int`**.

Though it’s not the best idea to test your own code (other programmers are more independent). Nonetheless, programmers often initially programmatically test their own code by writing a small main() function that calls the “**unit under test**” (UUT).

Independent: means that other programmers are able to notice ways to test your code that you would have not imagined

9.1.1 Test Driven Development

There is even a technique, respected but not widely practiced, called

“**test driven development**” (TDD).

Before you graduate you should read:

- Book “Test-Driven Development by Example” [Beck 2002].
- Or [Wikipedia – “Test Driven Development”]

In TDD, you write a unit test for the function or module you are about to write, before writing the function or module.

One advantage of doing “**glass box testing**”, where the test writer can look at the code being tested (or who possibly even wrote it themselves), is the test writer can arrange to test every branch path and switch case!

And state that her test covers 60% of “execution paths”.

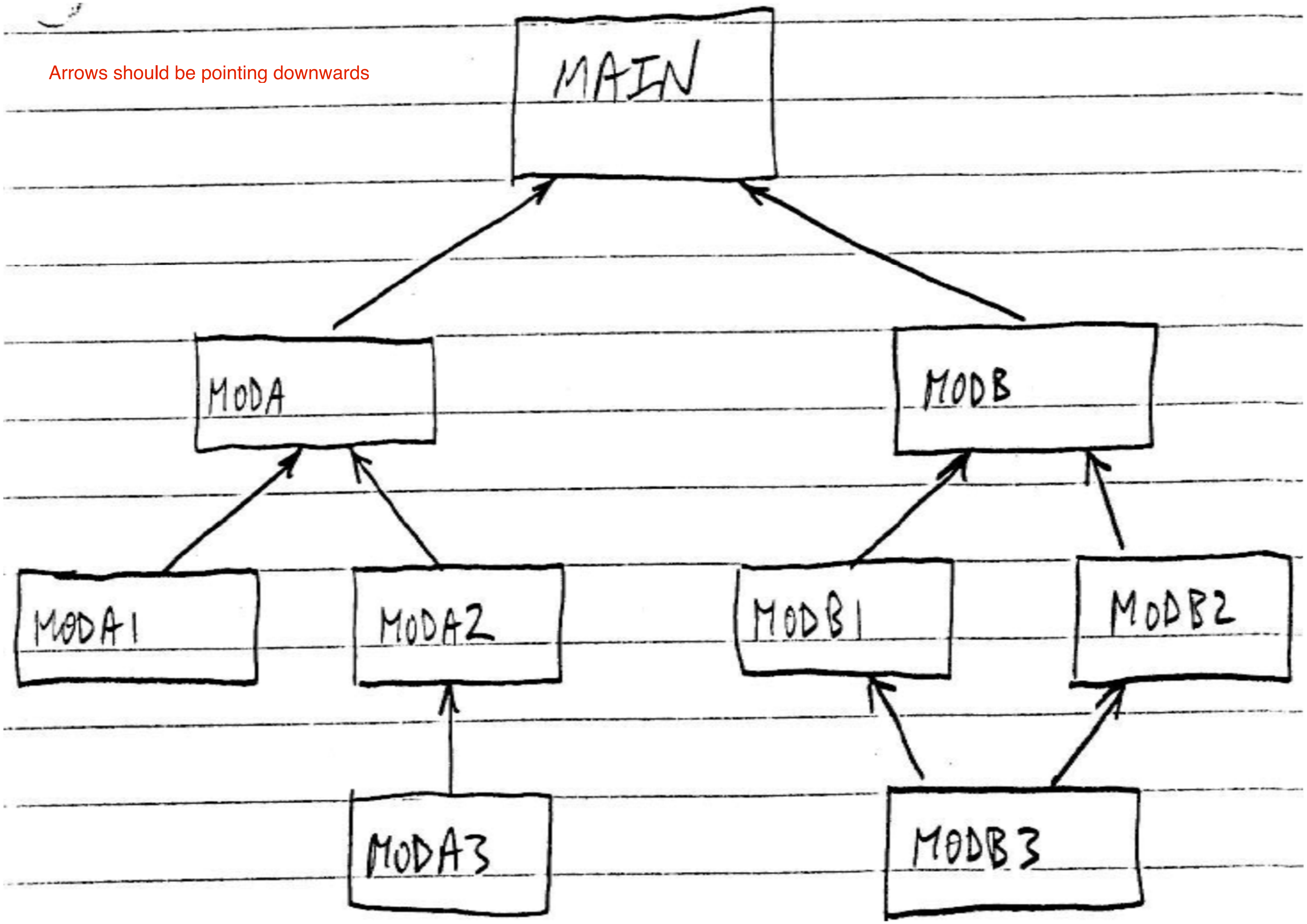
- Some test frameworks used in Quality Assurance departments of medium to large companies often have software that, after running hundreds of unit or system tests, will calculate automatically and report that you have, say, 43.7% branch path coverage.

9.1.2 <u>Unit Testing A Portion Of Call Tree</u>

Consider the following call structure, where the boxes represent either

- functions or
- modules.

Arrows should be pointing downwards



To unit test ModA3, you need to write a small program, called a **driver**. It calls and tests a subset (or all) of ModA3's functions and other features.

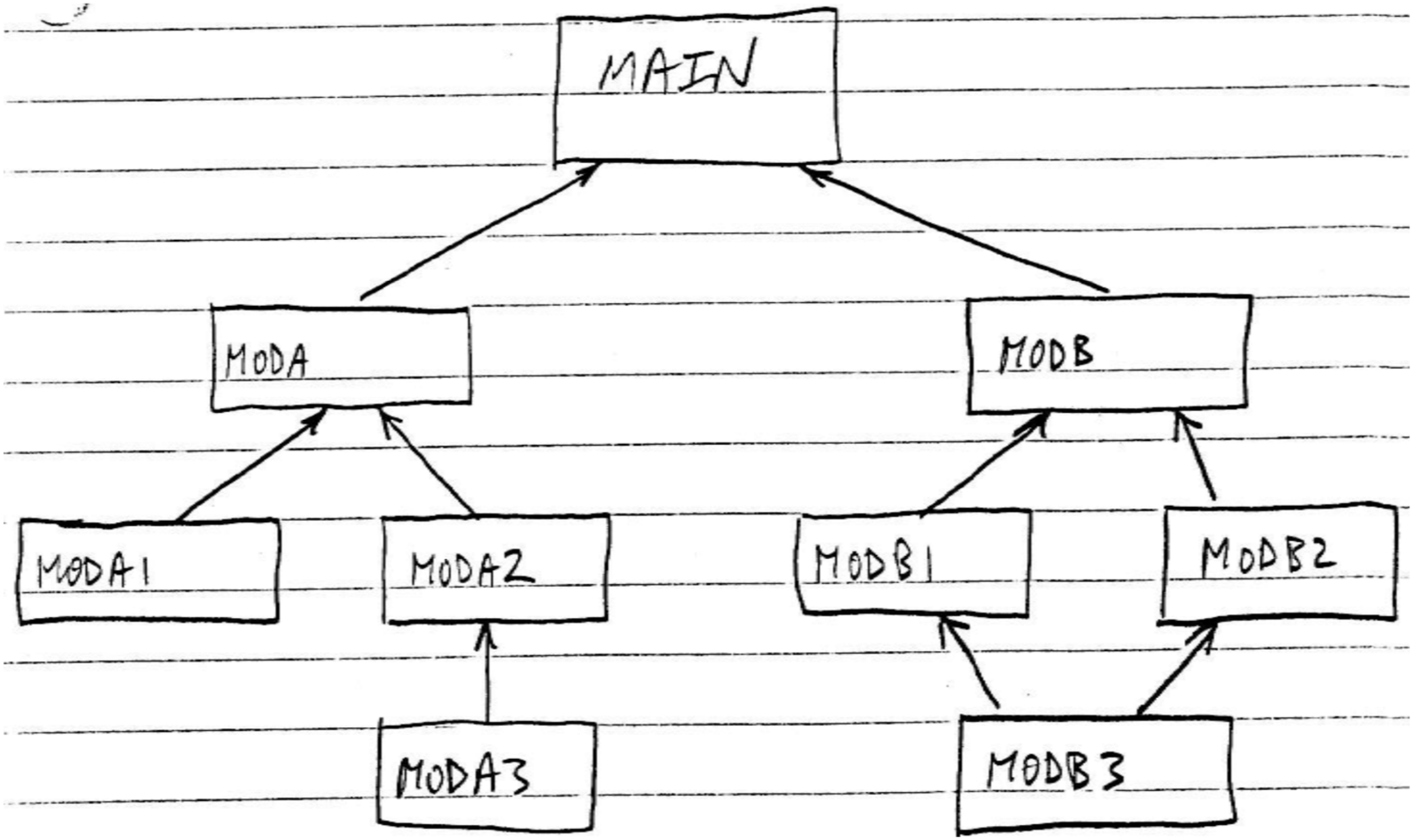
A driver in simplest terms is a **`main( )`** program.

A driver in "unit test framework" terminology might be a function that seems like a **`main( )`**, but fits into a suite of test cases.

This is "bottom first", or the start of "bottom-up", unit testing.

In contrast, to do top-down unit testing of your main.cpp you want to simply run your actual main module

To do this, you need to compile and link main.cpp with ModA and ModB.



But what if either:

- ModA and ModB don’t exist yet?
- Or aren’t stable.
- Or have complex parts?
- Or need to interact with external networks that would complicate your testing of the main module?

Solution: Program **stubs** to “stand in” for ModA and ModA.

Stubs may be:

- Do-nothing stubs.
- Or minimal-capability stubs.
- Or stubs that *fake* connection to networks or files.

Stubs are fairly easy to construct. E.g. Make a copy of their .h file, and just add '{' and '}' after each function signature.

Stubs may also log they've been called by printing to screen or to log a string like:

  - "Currently in ModA.cpp" or
  - "Currently in ModA_func( )".

[You could even write a utility to parse header files and automatically generate a linkable, runnable, logging stub for it.]

Stubs can have a bit more, but still fake functionality:

- Example #1: A stub for reading user input may just for all cases just:
  `return "test string from ModA_func()";`
- Example #2: A stub for **`cos(angle)`** may just return 0.5 no matter what angle is input,
  if this not critical to the caller.

- Example #3: Alternately, the stub could be written to return a different string depending on the whim of the person running the test.

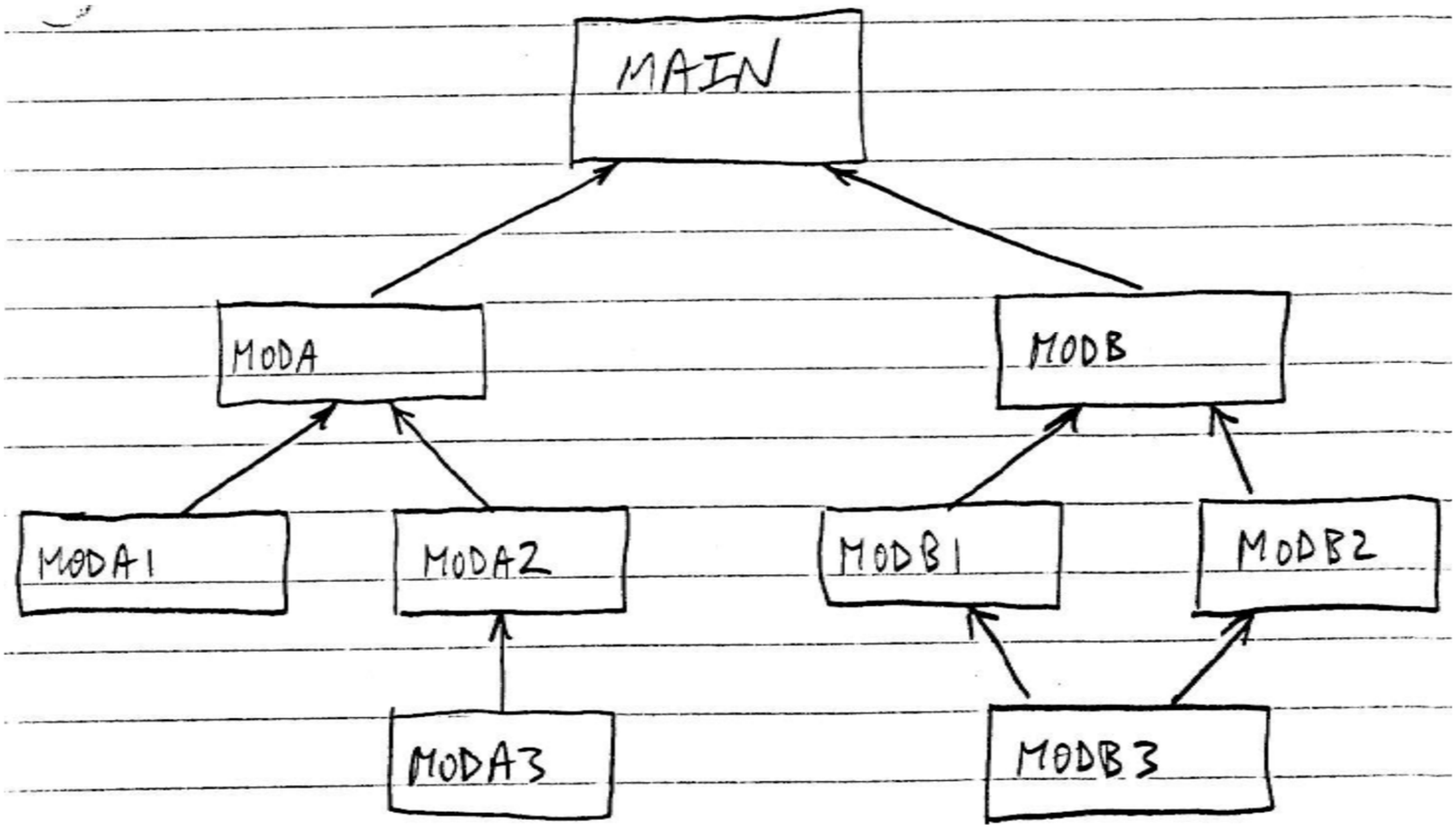
  The stub could just solicit the string from the user

  - if it’s not an automated test and there’s actually a real QA staff member running the test.

  In this case: your stub could:

```
      cin >> userInputString;
      return userInputString;
```

If you need to test a mid-level module like ModA or ModA2 below, you'll require both a driver and stub or stubs!



9.2 Low Level Design

Hopefully you have already been taught about basic data structures (e.g. linked lists) and algorithms (e.g. binary search).

Future courses will expand on those:

- Very advanced structures (B+ trees)
- Complex algorithms (e.g. quicksort)
- Multi-threaded programming.
- Data bases
- Graphics

When you mentioned that there is code you have wrongly called,
does that mean that your design of the functions does not match what you
currently have?
Yes

9.2.1 General Advice

- Use a library or framework of already tested and debugged objects/modules to maximum advantage. Rather than wasting time painfully writing your own.
  - This may require you *actually look* in the library to see what's available!
  - E.g. C++ Standard Template Library
    - There are whole books on the Standard Template Library, and on how best to use it, and what to be careful of.
    - There are good online references for when you are programming and want to quickly look something up.
    - E.g. http://cppreference.com

    - E.g. http://cplusplus.com
  - E.g. Object or GUI frameworks sometimes come with a browser to search the library.
- Use appropriate structures and algorithms. Trees are not always better. Most but not all are relatively poor for sequential access (i.e. O(log N) rather than O(1) for lists.
- Use step-wise refinement.
- Watch out for round-off error (especially if it can accumulate).
  - Discussed later.

9.2.2 <u>Selection of Data Structures</u>

Legend:

- Fast = O(1)
- Medium = O(log N)
- Slow = O(N)   [often O(1/2 N)]

	Random Access	**Seqential Access**	**Insert + Delete**
Unordered Array	Slow	Slow	Fast + Sery Slow
Ordered array	Medium	Fast	Very Slow
Ordered linked list	Slow	Fast	Slow
Tree	Medium	Medium	Medium
B+ tree	Medium	Fast	Medium

Note: The above structures can be just as easily used in files or RAM.

Most of the above “medium” speed items above use binary search or tree traversal.

Sometimes these can be sped up using some kind of “hashing” resulting in a so-called Hash Table.

- Fast
- But waste some space.

Use the structure and algorithm that uses the

- **least space,** yet provides
- **adequate performance** for the
- **kind of access** you
- **most frequently need**.

Is your application driven by hundreds of transactions/second?

Is application for single user, and need only be faster than human interaction:

What level of performance is needed by the application?

- typing in a human-user application, or
- web interaction?

If your design is not brittle (abstraction and coupling) you could upgrade structure and algorithm as needed.

9.2.3 Stepwise Refinement

Stepwise refinement is the act of coding by first specifying the outer loop of the **`main()`** using **pseudo-code**.

- Pseudo-code is loosely-worded code that is partly comment.
- Or speculation of what names of functions need to be called.

Once you have the loop algorithm **and loop exit conditions** correct, then consider how to implement the pseudo-statements.

They may be “refined” inline to actual code. Or to finer-grained pseudo-code.

Pseudo-code statements can be considered names of functions, and be pseudo-coded/written separately.

Nonetheless, you already have candidate function names from the pseudo-code verbs.

All that remains to define is to author the function interface (e.g. number and type of parameters, and exceptions).

Niklaus Wirth, who designed the Pascal and Module-3 languages, demonstrates this technique throughout [Wirth 2002].

```
IMPLEMENTATION MODULE naturalMerge
    VAR L: INTEGER;  a, b, c: Sequence
BEGIN
    REPEAT Reset(a); Reset(b); Reset(c);
      distribute;  (*c to a and b*)
      Reset(a); Reset(b); Reset(c);
      L := 0; merge  (*a and b into c*)
    UNTIL L = 1
END NaturalMerge
```

The two phases clearly emerge as two distinct pseudo-statements. They are now to be refined, i.e., expressed in more detail. The refined description of *distribute* is

```
distribute:  REPEAT copyrun(c, a);
               IF ~c.eof THEN copyrun(c, b) END
             UNTIL c.eof
```

and that of *merge* in

```
merge:       REPEAT mergerun; L := L+1
             UNTIL b.eof;
             IF ~a.eof THEN copyrun(a, c); L := L+1 END
```

From P. 107 of "Algorithms and Data Structures" by Niklaus Wirth, Prentice-Hall, 1986.

Note: copyrun( ) is used in both the pseudocode fragment for distribute and for merge.
We can refine our definition of copyrun by supplying a body for it:

```
PROCEDURE copyrun(VAR x, y: Sequence);
BEGIN (*from x to y*)
  REPEAT copy(x, y) UNTIL x.eor
END copyrun
```

Finally, we must refine what we mean by the pseudo-code statement (used in the merge fragment) called mergerun:

```
           (*merge from a and b to c*)
mergerun:  REPEAT
             IF a.first < b.first THEN
               copy(a, c);
               IF a.eor THEN copyrun(b, c) END
             ELSE copy(b, c);
               IF b.eor THEN copyrun(a, c) END
             END
           UNTIL a.eor OR b.eor
```

From p.109 of "Algorithms and Data Structures" by Niklaus Wirth, Prentice-Hall, 1986.

9.2.4 Round-Off Error

When making computations that mix integer and floating point variables, be very careful of the error that can occur.

E.g.

```
int i = static_cast<int>(2 * 3);
```

Will calculate 5.9999, which will round to 5!

Be particularly careful when making calculations that update an old value with new factors, resulting in a new value that becomes the old value for next calculation (or next day’s calculation).

E.g. The calculation of the Vancouver Stock Exchange closing average dropped 200 points over about 9 months due to stupid update design.

- To create today's closing average used yesterday's closing average updated by the transaction during today.
- The above is quick to calculate.
  - Vs. Adding up all the stocks from scratch.

  But can cause creeping numerical error.

9.3 References

[Beck 2002] Beck, Kent, "Test-Driven Development by Example", Addison Wesley, 2002. ISBN 978-0-321-14653-3.

[Wikipedia – "Test Driven Development"] https://en.wikipedia.org/wiki/Test-driven_development

[Wirth 2002] Niklaus Wirth, "Algorithms and Data Structures", Prentice Hall, 2002.