Thinking about Large Programs

What is a "large program"?

- One too large to be written by one person in a few days.

Why are large programs different?

- There are many tiny details.
- It is difficult, often impossible to keep all of those details in mind all the time.
- When there are several programmers, nobody is familiar with everything in the program.
- If the program is written over a period of more than a few days, people forget the details of what they have done.
- Errors in one part often affect other parts.

How should we respond to these problems?

- We organise programs into *modules.*
- Production of a module is a piece of work for a programmer or a group of programmers.
- Modules can be subdivided into smaller modules.
- Even individuals organise their work in modules.

Is modularisation an engineering issue?

- Some treat it as management: assigning people work.
- There are technical issues: issues independent of the people available.
- Dividing a program into modules is the subject of this lecture.

What is a module?

Historically modules were simply a unit of measure, e.g. *3.27 square meters.*

Manufacturers learned to build parts that were one unit large.

The word now means *the parts themselves.*

Modules are usually relatively self-contained systems that are combined to make a larger system.

Design is often the assembly of many previously designed modules.

The constraints on modules

If the modules are hardware, how you put them together is obvious; there are well-known physical constraints. There is a well-identified time at which modules are assembled to get the larger system.

If the modules are software, there are no obvious constraints. Theoretically software modules can be arbitrarily large, their interfaces arbitrarily complex.

During software development there are several different times at which parts are combined to form a whole.

During software development there are several different ways of putting parts together to get the whole.

Modules of software--when are parts put together?

A. While writing software
- parts: work assignments for programmer(s)
- when: files containing programs combined before compilation or execution.

B. When "linking" object programs.
- parts: separately assembled (compiled) programs with "relative" addressing
- when: addresses are inserted to provide links before execution.

C. When running a program in limited memory.
- parts: memory loads data and programs that refer to each other by memory addresses
- when: while the program is running.

The word "module" is used in programming literature for all three of these!

The ambiguity in the word leads to confusion.

In this course, we talk only about the first meaning.

The constraints on the three structures

Write time Constraints:
- Intellectual coherence for programmer
- Ability to understand, verify
- Ease of change

Assembly time constraints:
- duplicate names
- time to re-assemble

Run time constraints
- size of memory
- frequency of reference to items outside segment
- time to load/unload memory

The three sets of constraints are independent and they have only the word “module” in common. These are three different design concepts.

Myth of over-modularisation in TSS/360

TSS/360 was a major effort by IBM to build a time sharing system in the 60’s.

It was very very slow.

A well known IBM researcher attributed this to over-modularisation - the modules were too small and there were too many.

Previous popular wisdom -modules should be as small as possible

Researcher, “Too many small modules led to memory thrashing”.

Belief in many small modules was based on work assignment interpretation of that word.

Implementation used memory management interpretation.

Two meanings of module were confused.

The effects of confusing these meanings

Inefficiency results from forcing coincidence.

Write time modules may not be good memory load modules.

Write time modules need not be compiled separately. One may use macro substitution.

The three (or more) meanings must not be confused

In this lecture, modules are **always** write-time or change-time entities. We want them to have the following properties:

- They can be designed and changed independently.
- They can be divided into modules.

When do we stop subdividing modules?

- When they are so small that it is easier to write a new one than to change it.
- When the cost of specifying the interface exceeds any future benefit from having smaller modules.

The concept of "module" as a work assignment is only a definition. We need guidelines for designing the module structure of large programs.

The KWIC INDEX Example

Conventional structure

(1) Input Module

INPUT INTERFACE: Input format, marker conventions
OUTPUT INTERFACE: Memory format

(2) Circular Shift Module

INPUT INTERFACE: Memory format
OUTPUT INTERFACE: Memory format, perhaps the same

(3) Alphabetising Module

INPUT INTERFACE: Memory format
OUTPUT INTERFACE: Memory format

(4) Output Module

INPUT INTERFACE: Memory format
OUTPUT INTERFACE: Paper format, conventions, etc.

(5) Master Control Module

INTERFACE: names of the program to be invoked

The KWIC INDEX Example

Which program design decisions are most likely to change?

1. Input format
2. Memory formats
3. The decision to sort all the output before starting to print results.
4. Output formats

KWIC INDEX Example: An Alternative Structure

Line Holder Module

- A special purpose memory to hold lines of KWIC index

It consists of the following programs:

GET_CHAR (lineno, wordno, charno)

SET_CHAR (lineno, wordno, charno, char)

CHARS (lineno, wordno)

WORDS (lineno)

DELETE_LINE (lineno)

DELETE_WORD (lineno, wordno)

The KWIC INDEX Example: Alternative Structure

Input Module

- reads from input medium;
- *calls line-holder programs* to store in memory

Interface program: "INPUT"

The KWIC INDEX Example: Alternative Structure

Circular Shift Module

- Creates a "virtual" list of circular shifts.
- Uses line holder programs to get data from memory.
- It may, may not, create an actual table.

Interface programs:

(1) "CS_SETUP"

(2) "CS_CHAR (lineno, wordno, charno)"

The KWIC INDEX Example: Alternative Structure

Alphabetiser Module

- Does the actual sorting of the circular shifts.
- May or may not produce a new list.
- If it doesn't, it makes a directory.

Interface programs:

ALPH

ITH (lineno)

The KWIC INDEX Example: Alternative Structure

Output Module

- Does the actual printing.
- Calls ITH and circular shift programs.

Interface program:

OUTPUT

The KWIC INDEX Example: Alternative Structure

Master Control Module

- Links all the modules together to do the job.
- Is the main program, but very simple.
- Calls INPUT, CS_SETUP, ALPH, and OUTPUT.

The KWIC INDEX Example: Alternative Structure:

What happened?

Not necessarily getting a really different program.

Different way of cutting up a program--so that *likely* changes are confined to one person's work - one module.

System organised into set of modules based on explicit consideration of what needs to be changed.

Not necessarily better algorithms or data structures.

Simplifies interfaces

- information hiding, abstraction
- descriptions may be less familiar

In the original design, we could have used function tables to describe the interface.

In the new design, we can't because the data structure is internal, not external.

Terminology

Information-hiding modules

Identify the design decisions that are likely to change.

Have a module for each design decision that we consider very likely to change.

Requires experience and judgement. Experience tells us what is likely to change.

Terminology

The *secret* of a module

The design decision that might change--only the implementor needs to know what decision was made.

Line holder
- how lines are represented in memory

Input module
- input format

Circular shift module
- how circular shifts are represented

Alphabetiser
- sorting algorithm
- when the alphabetisation is done

Output
- output format

Terminology

Module Structure

The structure of a system is described by describing parts and their connections.

Connections: between modules are assumptions that they make about each other (interface).

Parts: work assignments

Design Errors in the Conventional Design

Flowchart boxes become modules.

There were unnecessary "connections".

- All of the modules contained code that is dependent on data structure design decisions.
- The whole program was written on the assumption of a given input format.
- The whole program was written on the assumption that printing would be done at the end.

Frequency of Switching from Module to Module

Steps-in-processing approach

- There are few transfers from module to module.
- The cost of the transfer is not significant.

Information-hiding approach

- There are many separate programs called from other modules.
- There is a high frequency of switching
- The cost of switching can be very significant.

Module access programs need not be subroutines.

The usual space-time tradeoffs apply.

Do not confuse run-time with write time!

We want to hide information about the design at write-time, not minimise data exchange at run-time.

Some designers try to reduce the information passed between modules at run-time.

This may actually increase the amount of information needed by the writers.

You can save run-time information storage and processing by making assumptions at write-time.

This will speed up your program and save memory, but may make it much harder to change.

Remember, the module structure is a write-time structure and may disappear with final assembly.

Design Procedure for Very Large Programs

A. Identify the secrets, separate as far as practical.

B. Design an information hiding interface for each:
   - Implementations of access programs are based on the secrets.
   - The interface won't change if the secrets do.

C. Specify all the interfaces precisely.

D. Implement independently
   - One module may use access programs from another.
   - Programs from one module cannot use data structures or internal programs from another.

E. Decompose the larger modules - go to A.

What is the relation between information hiding and abstract data types

Data Abstraction is a special case of information hiding. Algorithms can be hidden as well.

Data Types allow many copies of the hidden structure.

- Each variable has one copy of the hidden data structure.
- The module is the programs.
- The data structure copies are the variables.

What is the relation between information hiding and object-oriented approaches.

Modules are the code that produces objects.

Not dependent on message passing.

Class-inheritance not there.

Viewed as a family development

We are designing not one program, but a program family.

Decisions shared by all members of a family should be made early. Decisions likely to change should be postponed.

Early decisions are harder to change than later ones.

Structure decisions are early and hard to change.

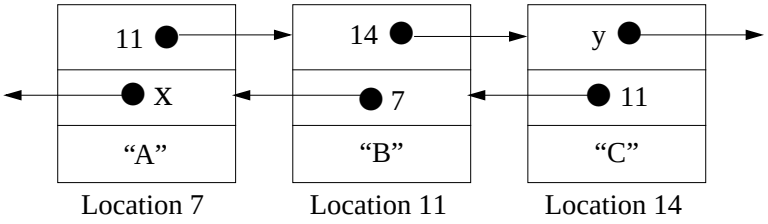
Interfaces: Should embody decisions less likely to change.

Implementation: Should embody decisions most likely to change.

The more likely a decision is to change, the more you restrict knowledge of it. Where possible decisions that are likely to be reversed should be made late in the design process.

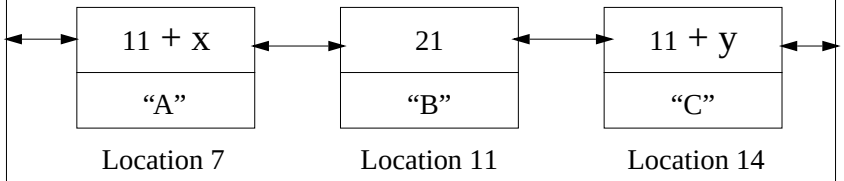
Example Linked Lists

Consider the following data structure:



This allows easy insertion and deletion.

Consider the following Alternative.



The link at each element is the sum of the forward and backward links.

Each alternative has advantages and disadvantages.The decision is likely to change.

Example: Linked List, using information hiding

(1) Make the job of maintaining the list a module.

(2) Provide the following interface programs.
- Current: returns the value of the current element.
- Moveforward: current element becomes next on list.
- Movebackward: current becomes previous on list.
- Atstart?: returns **true** if you cannot move backward.
- Atend?: returns **true** if you cannot move forward.
- Insertafter(x): insert a new element in the list after current; the value of the new element will be x.
- Insertbefore(x): insert a new element in the list before current; the value of the new element will be x.
- Remove: Deletes the current element from the list.
- Alter(x) Changes the value of the current element to x.

(3) Write the programs that use this list using only the interface programs.
- Your programs can do everything that they could do if they used the link structure directly.
- Your programs will be easier to understand and change.
- Your programs are more likely to be correct.