Routines & Functions

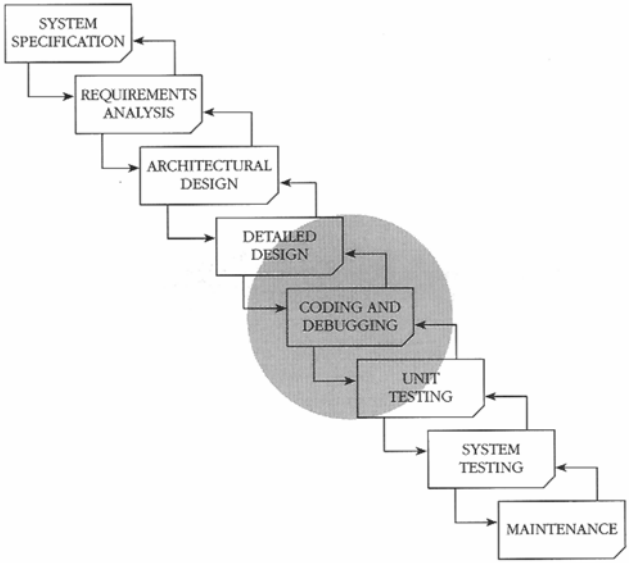
Lecture 3

Professor Peter Cheung
EEE, Imperial College

- In this lecture*, we will
  - Examine the process of software development
  - Discuss the design of routines/functions using pseudo-code or program design language (PDL)
  - How to design and construct good quality routines

* CODE COMPLETE 2nd Edition, chapters 5,7,9

Software Development Process



Phases in software design

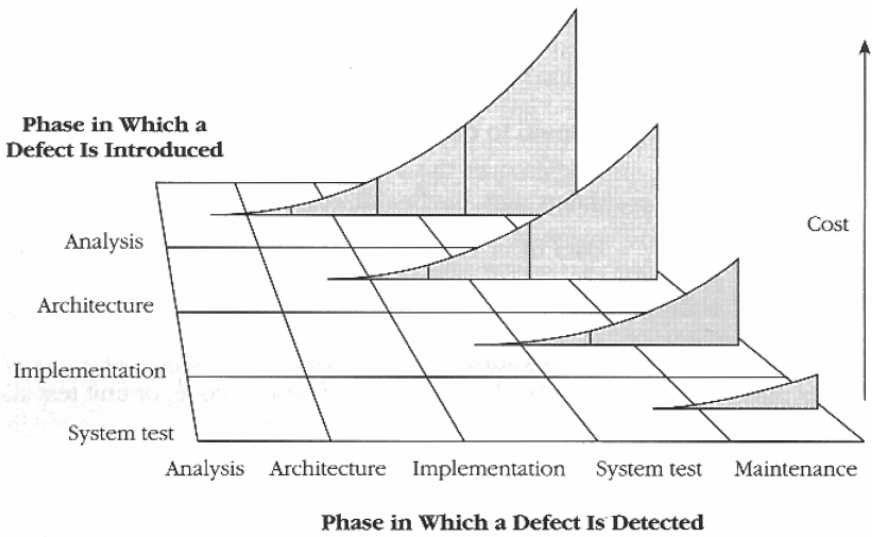
- **Architectural design**: Identify sub-systems.
- **Abstract specification**: Specify sub-systems.
- **Interface design**: Describe sub-system interfaces.
- **Component design**: Decompose sub-systems into components and modules.
- **Data structure design**: Design data structures to hold problem data.
- **Algorithm design**: Design algorithms for problem functions.

Cost of fixing bugs!

	Time Introduced		
Time Detected	**Requirements**	**Design**	**Code**
Analysis	1	—	—
Design	2	1	—
Passive tests	5	2	1
Structural tests	15	5	2
Functional tests	25	10	5

- Best time to find bugs: as early as possible.
- If you find a requirement error (i.e. in specification) during functional tests, the cost can be 25 times higher than if you find it during the analysis phase.

Cost of fixing bugs!



Program - Modules - Routines

Project (e.g. logic_sim.prj)

Module_1 (e.g. sim_engine.cpp)

```
main() {
  ....
}
```

```
ReadCircuitFile() {
  ....
}
```

.....

```
ReportErrors() {
  ....
}
```

Module_2 (e.g. graphics_library.cpp)

```
PlotTimeAxis() {
  ....
}
```

.....

```
PlotWaveForm() {
  ....
}
```


Procedures vs Functions

- Pascal has procedures and functions.
- C++ only has functions. Procedures in C++ are those functions that returns nothing (i.e. void functions).
- In general use **function** if the routine returns only ONE thing.
- Use **procedure** (i.e. void function) if the routine returns multiple
- It is also a common programming practise to have a function that operates as a procedure, but returns a status value:

```
if (FormatOutput (Input, Formatting, Output) == Success) { ...}
```

vs

```
FormatOutput (Input, Formatting, Output, Status);
If (Status == Success) { ... }
```


Design & Modularity

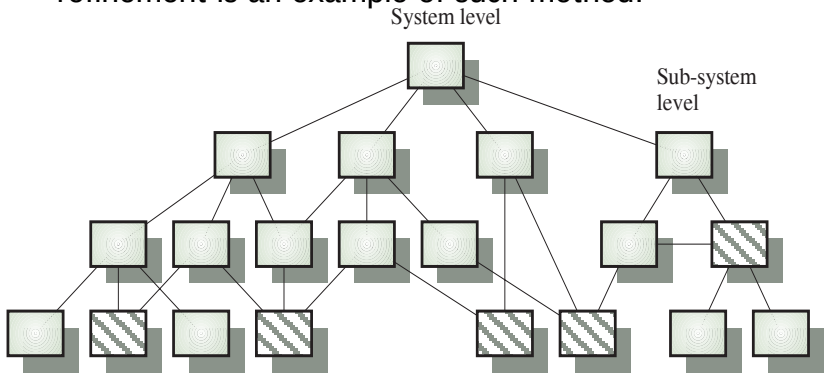
- A software package is composed of multiple, interacting **modules**.
- **Modularity** has long been seen as a key to cheap, high quality software.
- Need to determine:
  - What the modules are?
  - How the modules interact with one another?
- Modularity means that each module (and/or routine) must be:
  - **autonomous**
  - **coherent**
  - **Robust**
- The most obvious design method involve **functional decomposition**.
  - This leads to programs in which procedures represent distinct logical functions in a program.
  - Examples of such functions:
    - *DisplayMenu()*
    - *GetUserOptions()*
- This is called **procedural abstraction**

Characteristics of modules

- Five criteria to help evaluate modular design methods:
  1. Modular **decomposability**;
  2. Modular **composability**;
  3. Modular **understandability**;
  4. Modular **continuity**;
  5. Modular **protection**.

1. Module decomposability

- This criterion is met if the design method supports the decomposition of a problem into smaller sub-problems, which can be solved independently.
- Top-down design methods fulfil this criterion; stepwise refinement is an example of such method.



2. Module composability

- A method satisfies this criterion if it leads to the production of modules that may be freely combined to produce new systems.
- **Composability** is directly related to the issue of reusability. Examples are:
  - The Numerical Algorithm Group (NAG) libraries contain a wide range of routines for solving problems in linear algebra, differential equations, etc.
  - The Unix shell provides a facility called a pipe, written "–", whereby the standard output of one program may be redirected to the standard input of another; this convention favours composability.
- Composability is often at odds with decomposability
  - top-down design tends to produce modules that may not be composed in the way desired
  - top-down design leads to modules which fulfil a specific function, rather than a general one

3. Module understandability

- A design method satisfies this criterion if it encourages the development of modules which are easily understandable.
  - COUNTER EXAMPLE 1. Take a thousand lines program, containing no procedures; it's just a long list of sequential statements. Divide it into twenty blocks, each fifty statements long; make each block a module.
  - COUNTER EXAMPLE 2. "Go to" statements.
- Related to several component characteristics
  - Can the component be understood on its own?
  - Are meaningful names used?
  - Is the design well-documented?
  - Are overly complex algorithms used?
- Informally, high complexity means many relationships between different parts of the design.

4. Module continuity

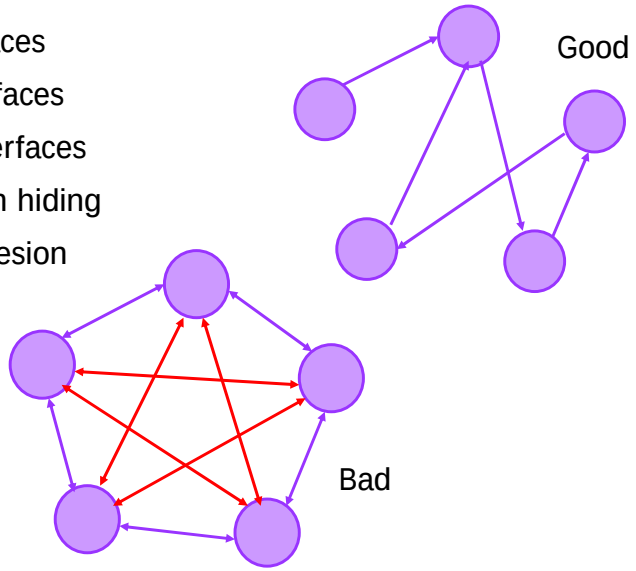
- **A method satisfies this criterion if it leads to the production of software such that a small change in the problem specification leads to a change in just one (or a small number of) modules.**
- EXAMPLE. Some projects enforce the rule that no numerical or textual literal should be used in programs: only symbolic constants should be used
- COUNTER EXAMPLE. Static arrays (as opposed to open arrays) make this criterion harder to satisfy.

5. Module protection

- **A method satisfied this criterion if it yields architectures in which the effect of an abnormal condition at run-time only affects one (or very few) modules**
- EXAMPLE. Validating input at source prevents errors from propagating throughout the program.
- COUNTER EXAMPLE. Using int types where subrange or short types are appropriate.

Five good principles in module design

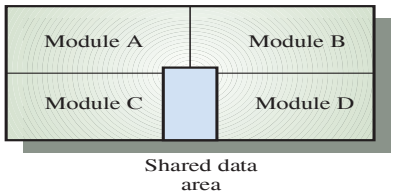
- Few interfaces
- Small interfaces
- Explicit interfaces
- Information hiding
- Strong cohesion



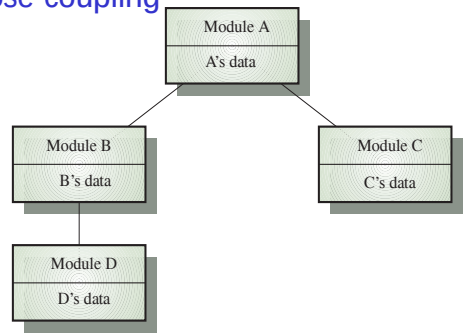
Small interfaces (Loose coupling)

- This principle states:
  - **If any two modules communicate, they should exchange as little information as possible.**
  - COUNTER EXAMPLE. Declaring all instance variables as **public**!
- **Coupling** is a measure of the strength of the inter-connections between system components.
- **Loose coupling** means component changes are unlikely to affect other components.
  - Shared variables or control information exchange lead to **tight coupling** (bad).
  - Loose coupling can be achieved by **state decentralization** (as in objects) and component communication via parameters or message passing.

Tight coupling



Loose coupling



- **Object-oriented systems** are loosely coupled because there is no shared state and objects communicate using message passing.
- However, an object class is coupled to its super-classes. Changes made to the attributes or operations in a super-class propagate to all sub-classes.

Explicit interfaces

- If two modules must communicate, they must do it so explicitly and can **easily be seen** and noticed.
  - If modules A and B communicate, this must be obvious from the code of A and B.
- Why? If we change a module, we need to see which other modules may be affected by these changes.

Information hiding

- This principle states:

  All information about a module, (and particularly how the module does what it does) should be private to the module unless it is specifically declared otherwise.
- Thus each module should have some interface, which is how the world sees it anything beyond that interface should be hidden.
- The default rule:
  - Make everything private unless explicitly publish the information

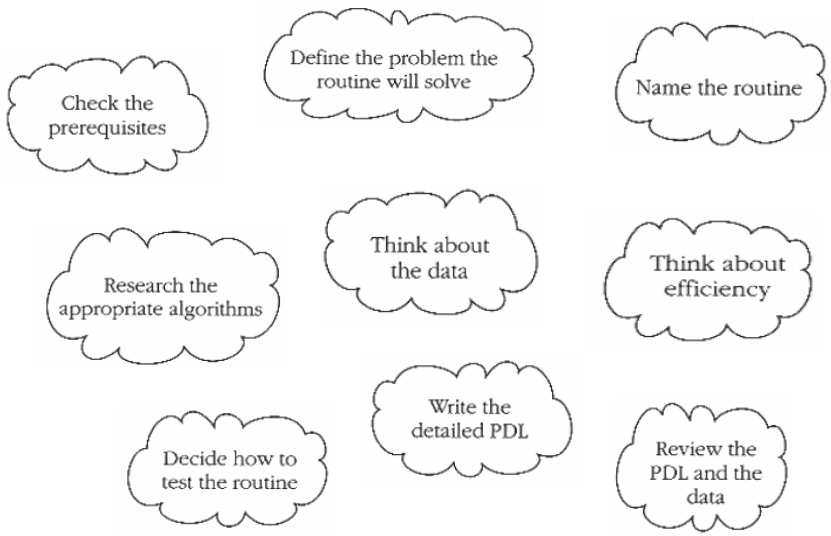
Cohesion

- A measure of how well a component "**fits together**".
- A component should implement a **single logical entity** or function.
- Cohesion is a desirable design component attribute as when a change has to be made, it is **localized** in a single cohesive component.
- Various **levels of cohesion** have been identified.

Cohesion Levels

- **Coincidental cohesion** (weak)
  - Parts of a component are simply bundled together.
- **Logical association** (weak)
  - Components which perform similar functions are grouped.
- **Temporal cohesion** (weak)
  - Components which are activated at the same time are grouped.
- **Communicational cohesion** (medium)
  - All the elements of a component operate on the same input or produce the same output.
- **Sequential cohesion** (medium)
  - The output for one part of a component is the input to another part.
- **Functional cohesion** (strong)
  - Each part of a component is necessary for the execution of a single function.
- **Object cohesion** (strong)
  - Each operation provides functionality which allows object attributes to be modified or inspected.

Steps in building a routine



The use of pseudo-code or PDL

- NEVER write the code of a routine or function right away – you will get it wrong!
- Use **pseudocode** or program description language (PDL) (which is a form of pseudo English) to help the design of the routine.

Here are some guidelines of using pseudo-code

- Describe specific operations precisely with **English-like statements**.
- **Avoid** the use of target **programming language**.
- Focus on the **intention** and **meaning**, NOT implementation.
- Write in low enough level to help generating code.

Example of BAD pseudo-code

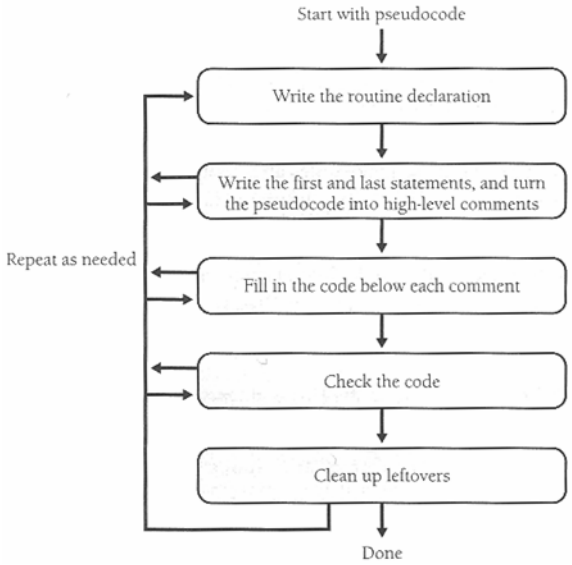
```
Increment resource number by 1
Create a new structure using make_struc
If make_struc() returns NULL then return 1
Invoke OSrsrc_init to initialize a resource for the operating system
*hRsrcPtr = resource number
Return 0
```

Example of GOOD pseudocode

```
set default status to "fail"
look up message based on error code

if error code is valid
    if doing interactive processing,
        display error message interactively and declare success
    if doing command line processing,
        log error message to command line and declare success
if error code is not valid, notify user that internal error detected
Return status information
```


Steps in coding a routine



Step 1: Write the routine declaration

```
/* This routine outputs an error message based on an error code
   supplied by the calling routine.  The way it outputs the message
   depends on the current processing state, which it retrieves on its
   own.  It returns a value indicating success or failure.
*/

Status ReportErrorMessage (
        ErrorCode  errorToReport
        )
```


Step 2: Turn Pseudocode into high-level comments

```
.....
Status ReportErrorMessage (
        ErrorCode  errorToReport
        ) {
  // set default status to "fail"
  // look up message based on error code

  // if error code is valid
      //  if doing interactive processing,
      //      display error message interactively and declare success

      //  if doing command line processing,
      //      log error message to command line and declare success

  // if error code is not valid, notify user that internal error detected

  // Return status information
}
```

Step 3: Fill in code below comments

```
.....
    // set default status to "fail"
    Status errorMessageStatus = Status_Failure;

    // look up message based on error code
    Message errorMessage = LookupErrorMessage (errorToReport);

    // if error code is valid
    if (errorMessage.ValidCode() ) {    //determine the processing method
        ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();

        //  if doing interactive processing,
        //      display error message interactively and declare success
        if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
            DisplayInteractiveMessage( errorMessage.Text() );
            errorMessageStatus = Status_Success;
        }

        //    if doing command line processing,
        //      log error message to command line and declare success
        .......
```


Step 3: More ...

```
.....
        //    if doing command line processing,
        //      log error message to command line and declare success
    else if ( errorProcessingMethod == ProcessingMethod_CommandLine) {
            CommandLine  messageLog;
            if ( messageLog.Status() == CommandLineStatus_ok ) {
                messageLog.AddToMessageQueue ( errorMessage.Text() );
                messageLog.FlushMessageQueue ();
                errorMessageStatus = Status_Success;
            }
        }
    }
    // if error code is not valid, notify user that internal error detected
    else {
        DisplayInteractiveMessage ("Internal Error: Invalid error code in ReportErrorMessage()");
    }

    // return status information
    return errorMessageStatus;
}
```

Candidate for new routine

Step 3: More ...

```
..... // if error code is valid
    if (errorMessage.ValidCode() ) {    //determine the processing method
        ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();

        //  if doing interactive processing,
        //      display error message interactively and declare success
        if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
            DisplayInteractiveMessage( errorMessage.Text() );
            errorMessageStatus = Status_Success;
        }

        //    if doing command line processing,
        //      log error message to command line and declare success
    else if ( errorProcessingMethod == ProcessingMethod_CommandLine) {
            erroMessageStatus = DisplayCommandLineMessage();
        }
    }
    // if error code is not valid, notify user that internal error detected
    else {
        DisplayInteractiveMessage ("Internal Error: Invalid error code in ReportErrorMessage()");
    }
    ......
```


Good reasons for creating a routine

- **Reduce Complexity** – hide details, replace deep nested loops
- Introduce an intermediate, understandable **abstraction**
  - Makes the code a lot more readable
- **Avoid duplicate** code
- **Encapsulate** a sequence of operations
- **Hide pointer** operations
- **Improve portability** – hide nonportable parts in routines
- **Simplify** complicated **Boolean** conditionals
  - Details of test hidden; descriptive function name documents code
- **Improve performance** – optimise code in fewer places

Good Routine Names

- For a procedure name, **use a strong verb** followed by an object
  - E.g. PrintReport(), CalcMonthlyRevenue(), CheckOrderInfo()…..
- For a function name, use a **description of the return value**
  - E.g. NextStudentID(), PrinterReady(), CurrentPenColour() …
- **Avoid** meaningless or wishy-washy verbs
  - Don't use: ProcessInput(), DealWithOutput()
  - Instead use: ReadPersonalDetails(), FormatAndPrintOutput()
- Make names of routines **as long as necessary**, but not longer
  - Research shown that around 10 – 20 characters are realistic
- Establish **conventions** for common operations
  - E.g. **Get** prefix destructive input and a **Query** prefix for non-destructive input: GetInputChar() & QueryInputChar()

How to use routine parameters?

- Put parmeters in **input-modify-output** order
- Create your own **IN** and **OUT** keywords
- Use **ALL** parameters
- Put status or error variables **last** – these are usually incidental to the main purpose of the routine

```
#define IN
#define OUT
Void InvertMatrix (
        IN Matrix originalMatrix,
        OUT Matrix *resultMatrix
);
```

- **Never use routine parameters** as working variables. Use local variables instead
- **Document all interface** careful, e.g. including all assumptions
- Limit the number of parameter to a **maximum of 7**
- Pass parameters by value in preference to passing them by reference unless:
  - It is modified by the routine
  - It is a large structure

Example of a low-quality routine

- × Bad name
- × no routine documentation
- × poor style & layout
- × inputRec should not have this name if it is modified
- × use global variables corpExpense and profit (bad!)
- × poor defence against bad data (i.e. crntQtr = 0)
- × use "magic" numbers (e.g. 100)

```
Void HandleStuff (CORP_DATA &inputRec, int CrntQtr,
   EMP_DATA emprRec, double &estimRevenue, double
   ytdReveneue, int screenX, int screenY, COLOR_TYPE
   &newColor, COLOR_TYPE &prevColor, StatusType &status,
   int expenseType)    {
int i;
for ( i = 0; i < 100; i++ ) {
    inputRec.revenue[i] = 0;
    inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
updateCorpDatabase( empRec );
estimRevenue = ydtRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
Status = SUCCESS;
If ( expenseType == 1 ) {
   for ( i = 0; i < 12; i++ )
        profit[i] = revenue[i] – expense.type1[i];
   }
Else if (expenseType == 2) {
        profit[i] = revenue[i] – expense.type2[i];
        }
Else if ( expenseType == 3)
        profit[i] = revenue[i] – expense.type3[i];
        }
```


Example of a low-quality routine

- × screenX & screenY parameters not used in routine
- × prevColor passed by reference but never changed
- × too many parameters (should be 7 or below)
- × no comments & poor documentations
- × routine does too many things

```
Void HandleStuff (CORP_DATA &inputRec, int CrntQtr,
   EMP_DATA emprRec, double &estimRevenue, double
   ytdReveneue, int screenX, int screenY, COLOR_TYPE
   &newColor, COLOR_TYPE &prevColor, StatusType &status,
   int expenseType)    {
int i;
for ( i = 0; i < 100; i++ ) {
    inputRec.revenue[i] = 0;
    inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
updateCorpDatabase( empRec );
estimRevenue = ydtRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
Status = SUCCESS;
If ( expenseType == 1 ) {
   for ( i = 0; i < 12; i++ )
        profit[i] = revenue[i] – expense.type1[i];
   }
Else if (expenseType == 2) {
        profit[i] = revenue[i] – expense.type2[i];
        }
Else if ( expenseType == 3)
        profit[i] = revenue[i] – expense.type3[i];
        }
```

What to do before the next lecture?

- Complete Exercise B (non-assessed) – See separate sheet
- Either read Lesson 3 & Lesson 4 of the following C++ Tutorial on the web:

http://www.functionx.com/cppbcb/Lesson03.htm
http://www.functionx.com/cppbcb/Lesson04.htm

Alternatively, read Chapter 3 & 4 of Savitch