Low Level Design & Implementation
Writing High-Quality Routines

- What is a routine?
  - A function, procedure, method, subroutine, etc.

- What is a **high-quality** routine?
  - Let's start by looking at what a high-quality routine is NOT
void HandleStuff( CORP_DATA & inputRec, int crntQtr,
    EMP_DATA empRec, float & estimRevenue, float ytdRevenue,
    int screenX, int screenY, COLOR_TYPE & newColor,
    COLOR_TYPE & prevColor, STATUS_TYPE & status,
    int expenseType )
{
    for ( int i = 1; i <= 100; ++i ) {
        inputRec.revenue[ i ] = 0;
        inputRec.expense[ i ] = corpExpense[ crntQtr, i ];
    }
    UpdateCorpDatabase( EmpRec );
    estimRevenue = ytdRevenue * 4.0 / (float)crntQtr ;
    newColor = prevColor;
    status = Success;
    if ( expenseType == 1 ) {
        for ( int i = 1; 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 ];
    }
}
Reasons for Creating Routines

- Reducing complexity
  - Decompose complicated algorithms into smaller pieces so you don't have to think about the entire problem at once
  - Combine smaller routines to implement the larger algorithm
  - Deep nesting of code indicates a need for further decomposition - put the nested part in a separate routine

- Avoiding duplicate code
  - Duplicated code is a design error
  - Avoiding duplicate code makes the program easier to modify, debug, and optimize
  - Promotes code reuse
Reasons for Creating Routines

- Limiting effects of changes
  - Isolate code that is likely to change in a routine so that the effects of changes will be limited in scope
- Hiding sequences
  - Sequences of statements that appear frequently should be placed in a separate routine
- Hiding implementation details of data structures
  - Create a class that represents the abstract data type
  - All details of the ADT are hidden inside the ADT class
- Hiding global data
  - If you can't resist using global data, hide its implementation details inside a class
Reasons for Creating Routines

- Making code more readable

```java
if ( node != null ) {
    while ( node.next != null ) {
        node = node.next;
    }
    leafName = node.name;
}
else {
    leafName = "";
}
leafName = GetLeafName( node );
```
Reasons for Creating Routines

- Improving portability
  - Isolate non-portable code in routines, making it easier to port the code to a new platform

- Isolating complex operations
  - Tricky boolean tests
  - Complex pointer operations
Modularity

- Routines should be like a "black box"
- You know what goes in
- You know what comes out
- You have no idea what goes on inside
- The interface is
  - simple
  - very well defined
  - easy to understand
Good Routine Names

- Procedure names (no return value)
  - PrintReport, CalcMonthlyRevenues, CheckOrderInfo, RepaginateDocument
  - The general form is Verb-Object
  - In OO languages, only the verb is needed because the object is implicit in the method call

- Function names (has return value)
  - A function name should describe the function's return value
  - NextCustomerId, IsPrinterReady, CurrentPenColor

- Avoid meaningless verbs
  - HandleCalculation, PerformServices, DealWithInput
  - If a routine is hard to name, it is probably not well designed because it is hard to describe what it does
Good Routine Names

- The name should describe **everything** that the routine does
  - PrintReportAndInitPrinter rather than just PrintReport
- Make routine names long enough to be easily understandable
  - 15 - 20 characters is not unreasonable
- Establish conventions for naming routines
  - Boolean functions - IsReady, IsLeapYear, …
  - Getters/setters - GetName, SetName, …
  - Initialization - Init, Setup, …
Strong Cohesion

- Cohesion - how closely related are the operations performed by a routine
- A perfectly cohesive routine does one and only one thing
  - GetCustomerName, EraseFile, CalcLoanPayment
- Routines that do too much should be obvious if we are naming them properly
  - DoDishesAndWashClothesAndSweepFloor
Cohesion Examples

- A routine calculates an employee's age, given a birth date
- A routine that calculates an employee's age and time to retirement, given a birth date
- A routine prints a report and then reinitializes the data structures on which the report is based
- A routine prints a quarterly expense report, a monthly expense report, or a daily expense report, depending on the value of a control flag that's passed in
Loose Coupling

- Coupling - the strength of the connection between two routines
- We want routines to be "loosely coupled", meaning that they are as independent as possible
- When dependencies do exist, they are clearly defined and as small as possible
- Highly-coupled routines are difficult to separate and use with other routines (like Siamese twins)
- Loosely-coupled routines are easy to reuse with other routines (like model railroad cars)
Levels of Coupling

- **Simple-data Coupling (OK)**
  - Two routines are simple-data coupled if all the data shared between them is non-structured and its all passed through a parameter list.
  - Example: One routine passes a variable containing an angle in degrees to a tan() routine.
  - Example: One routine passes five variables to another routine, including a name, address, phone number, birth date, and social security number.
Levels of Coupling

- Data-structure Coupling (can be OK or bad)
  - Two routines are data-structure coupled if the data shared between them is structured and passed through a parameter list
  - Example: One routine passes an EmpRec structure to another routine, which uses all of the fields in the EmpRec (OK)
  - Example: One routine passes an EmpRec structure to another routine, which uses only the birth date and phone number fields (bad)
Levels of Coupling

- Control Coupling (bad)
  - Two routines are control-coupled if one routine passes data to the other that tells the second routine what to do
  - Example: One routine passes another routine a flag telling it whether to print a quarterly, monthly, or daily report
Levels of Coupling

- Global-data Coupling (bad)
  - Two routines are global-data coupled if they make use of the same global data
  - Tolerable if the global data is read-only
  - Example: Employee records are stored in a global array. One routine modifies an employee record and then passes the ID of the employee to another routine. The called routine uses the ID to read the employee record from the array
Levels of Coupling

- Pathological Coupling (bad)
  - Two routines are pathologically-coupled if one directly uses the code or data inside the other
  - Some languages do not allow pathological coupling
  - Example: Using the C `setjmp` and `longjmp` functions to jump from the middle of one routine to the middle of another routine
  - Example: One routine uses pointers to modify the machine instructions in another routine
Defensive Programming

- Use assertions to document and verify assumptions
- Validate all input parameters
- Check for and handle errors returned by other routines
Using Parameters

- Put parameters in the following order: in, in-out, out
- If several routines have similar parameters, put the similar parameters in a consistent order
- Use all of the parameters
- Put status or error variables last
- Don't use routine parameters as working variables
- Limit the number of parameters to about seven
- Pass only the parts of structured variables that the routine needs
Coding Style

int i;
for ( i = 1; i <= num; ++i )
MeetsCriteria[ i ] = true;
for ( i = 2; i <= num / 2; ++i ) {
  int j = i + i;
  while ( j <= num ) {
    MeetsCriteria[ j ] = false;
    j += i;
  }
}
for ( i = 1; i <= num; ++i )
if ( MeetsCriteria[ i ] ) {
  cout << i << " meets criteria." << endl;
}
Coding Style

```cpp
int primeCandidate;
for ( primeCandidate = 1; primeCandidate <= num; ++primeCandidate )
    isPrime[ primeCandidate ] = true;

int factor;
for ( factor = 2; factor <= num / 2; ++factor ) {
    int factorableNumber = factor + factor;
    while ( factorableNumber <= num ) {
        isPrime[ factorableNumber ] = false;
        factorableNumber += factor;
    }
}

for ( primeCandidate = 1; primeCandidate <= num; ++primeCandidate )
    if ( isPrime[ primeCandidate ] ) {
        cout << i << " is prime." << endl;
    }
```
Writing High-Quality Classes

- Just like for routines, the following principles apply to writing classes
  - Modularity - black box with simple, easy to use interface
  - Cohesion - all methods on a class are highly related
  - Coupling - dependencies between classes are minimized as much as possible
- These principles also apply to organizing classes into packages
Information Hiding

- Classes should hide as much of their internal implementation as possible.
- Classes are like icebergs: seven-eighths is under water, only one-eighth is visible to the outside.
- This helps minimize dependencies between classes and provides maximum freedom to change the class implementation.
- All variables and many methods should be private.
  - E.g., a class might have 25 methods, only 5 of which are public.
- This also applies to package design. Do not expose classes outside the package unnecessarily.