Writing Quality Code

CS 240 – Advanced Programming Concepts

```
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 ];
  }
}
```

Quality Code Example

- If you would like to see an example of generally well-written code, look at the Job Scheduler program

Strong Cohesion

- Just like classes, methods should be highly cohesive
- A cohesive method does one and only one thing, and has a name that effectively describes what it does
  - getCustomerName, eraseFile, calculateLoanPayment
- Methods that do too much become obvious if we name them properly
  - doDishesAndWashClothesAndSweepFloor

Good Method Names

- A method name should clearly and completely describe what the method does
  - If a method prints a report and re-initializes the printer, it should be named printReportAndInitPrinter , not just printReport
- Method has no return value (i.e., void)
  - Name should be a verb or verb phrase
- Method has return value (i.e., non-void)
  - Name can be a verb or verb phrase
  - Or, it can describe what the method returns instead of what it does
  - isPrinterReady, currentPenColor, nextCustomerId

Good Method Names

- Avoid meaningless verbs
  - handleCalculation, performServices, dealWithInput
  - Methods that are not cohesive are often difficult to name
- Make method names long enough to be easily understood (don't abbreviate too much)
- Establish conventions for naming methods
  - Boolean functions - isReady, isLeapYear, …
  - Initialization - Initialize/Finalize, Setup/Cleanup, …
  - Getters/setters - getName, setName, …
  - add/remove, insert/delete

Reasons for Creating Methods

- One of our primary tools in writing quality code is knowing when and why to create new methods

- Classes are abstractions that represent the “things” in a system
- Methods are abstractions that represent the “algorithms” in a system

- There are many reasons to create methods; we will focus on a few:
  - Top-down decomposition of algorithms
  - Avoiding code duplication
  - Avoiding deep nesting

Algorithm Decomposition

- Long or complex methods can be hard to understand
- Long or complex methods can be simplified by factoring out meaningful sections of code into well-named sub-methods
- The original method becomes a "driver" that calls the sub-methods (it becomes shorter, simpler, and easier to understand)
- The extracted methods may be placed on different classes (i.e., put sub-methods on the class that contains the data they use)
- Decomposition continues until methods are sufficiently short and simple
- Example: Schedule's getNextWorkDay, isWeekendDay, and isHoliday methods

Comments

- If you feel a need to comment a paragraph of code, consider putting that section of code in a method of its own with a descriptive name
- This may do away with the need for the comment, and result in highly-readable code
- Do whatever makes the code the most readable
  - Factor out into separate method with a good name, or
  - Leave code inline with a comment
- If a method is heavily commented, that might indicate that further decomposition is necessary

Avoid Code Duplication

- Avoiding code duplication is one of the most important principles of software design
- Duplicated code makes software maintenance difficult and error-prone
- If the same code is needed in multiple places, put the code in a method that can be called wherever the code is needed
- Inheritance can also be used to avoid code duplication (inherit shared code from a common superclass)

Deep Nesting

- Excessive nesting of statements is one of the chief culprits of confusing code
- You should avoid nesting more than three or four levels
- Creating additional sub-methods is the best way to remove deep nesting

Parameters

- Use all of the parameters
- The more parameters a method has, the harder it is to understand
- The fewer parameters the better
- One rule-of-thumb is that you should limit parameters to no more than 7, and that many should be rare
- Order parameters as follows: in, in-out, out

Guidelines for Initializing Data

- Improper data initialization is one of the most fertile sources of error in computer programming
- Initialize variables when they're declared
- Declare variables close to where they're used
  - Variables don't have to be declared at the top of the method
- Check for the need to reinitialize a variable
  - Counters, accumulators, etc.
- Compiler warnings can help find un-initialized variables
- Java automatically initializes instance variables and doesn’t allow you to use uninitialized local variables

Code Layout

- The physical layout of the code strongly affects readability
  - Imagine a program with no newlines
  - Imagine a program with no indentation
- Good layout makes the logical structure of a program clear to the reader
- Good layout helps avoid introducing bugs when the code is modified
- Pick a style that you like and consistently use it
  - If your organization has a standard, use it (even if you prefer a different style)

Whitespace

- Use whitespace to enhance readability
  - Spaces, tabs, line breaks, blank lines
- Organize methods into "paragraphs"
  - Paragraph = a group of closely related statements
  - Separate paragraphs with one or more blank lines
- Indentation
  - Use indentation to show the logical structure (i.e., nesting)

Expressions

- Arithmetic and logic expressions can be hard to understand
- Over-parenthesize arithmetic expressions
  - Enhance readability
  - Make clear the order of operator evaluation
- Insert extra spaces between operands, operators, and parentheses to enhance readability

```
while (startPath+pos<=length(pathName)&&
        pathName[startPath+pos]!=';') {
   …
}

while (((startPath + pos) <= length(pathName)) &&
        pathName[startPath + pos] != ';') {
   …
}
```

Expressions

- Put separate conditions on separate lines

```
if (('0' <= inChar && inChar <= '9') || ('a' <= inChar &&
   inChar <= 'z') || ('A' <= inChar && inChar <= 'Z')) {
   …
}
```

```
if (('0' <= inChar && inChar <= '9') ||
    ('a' <= inChar && inChar <= 'z') ||
    ('A' <= inChar && inChar <= 'Z')) {
   …
}
```

Expressions

- Put expressions, or pieces of them, in well-named submethods

```
if (isDigit(inChar) || isLowerAlpha(inChar) || isUpperAlpha(inChar)) {
   …
}
```

- Or, even better

```
if (isAlphaNumeric(inChar)) {
   …
}

boolean isAlphaNumeric(char c) {
   return (isDigit(c) || isLowerAlpha(c) || isUpperAlpha(c));
}
```

Placing curly braces

```
for (int i=0; i < MAX; ++i) {
    values[i] = 0;
}

for (int i=0; i < MAX; ++i)
{
    values[i] = 0;
}

for (int i=0; i < MAX; ++i)
{   values[i] = 0;
}

for (int i=0; i < MAX; ++i)
    {
    values[i] = 0;
    }
```

Placing curly braces

- What about this?

```
for (int i=0; i < MAX; ++i)
    values[i] = 0;
```

Method parameters

- Use spaces to make method parameters readable

```
WebCrawler.crawl(rootURL,outputDir,stopWordsFile);

WebCrawler.crawl( rootURL, outputDir, stopWordsFile );

WebCrawler.crawl(rootURL, outputDir, stopWordsFile);
```

One statement per line

- Declare each variable on a separate line
  - More robust under modification
  - Easier to understand

```
int * p, q;  // oops!
```

```
int * p;  // correct
int * q;
```

- Don't put multiple statements on the same line

```
x = 0; y = 0;
```

```
x = 0;
y = 0;
```

Wrapping long lines

- When should you wrap long lines?
  - When they won't fit on the screen?
  - Whose screen?
- Wrapping between 80 and 100 characters is common
  - Lines longer than that are hard to read
  - It discourages deep nesting
- Align continuation lines in a way that maximizes readability

Wrapping long lines

```
private boolean isNthDayOfWeekInMonth(Calendar date, int n,
                                      int dayOfWeek, int month) {
        ...
}


target = addDependenciesToTarget(dependencyGraph, targetName,
                                 dependencyList);


DailySchedule newDailySchedule =
    new DailySchedule(getNextSchedulableDay(today));


return (date.get(Calendar.DAY_OF_WEEK) == dayOfWeek &&
        date.get(Calendar.MONTH) == month &&
        date.get(Calendar.DAY_OF_WEEK_IN_MONTH) == n);
```

Pseudo-Code

- When writing an algorithmically complex method, write an outline of the method before starting to code
- Use English-like statements to describe the steps in the algorithm
- Avoid syntactic elements from the target programming language
  - Design at a higher level than the code itself
- Write pseudo-code at the level of intent
  - *What* more than *how* (the code will show how)
- Write pseudo-code at a low enough level that generating code from it is straightforward
  - If pseudo-code is too high-level, it will gloss over important details

Example of bad pseudo-code

```
increment resource number by 1
allocate a dlg struct using malloc
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource
      for the operating system
*hRsrcPtr = resource number
return 0
```

- Intent is hard to understand
- Focuses on implementation rather than intent
- Includes too many coding details
- Might as well just write the code

Example of good pseudo-code

```
Keep track of current number of resources in use
If another resource is available
   Allocate a dialog box structure
   If a dialog box structure could be allocated
      Note that one more resource is in use
      Initialize the resource
      Store the resource number at the location
         provided by the caller
   EndIf
EndIf
Return TRUE if a new resource was created
else return FALSE
```

- Written entirely in English
- Not programming language specific
- Written at level of intent
- Low-level enough to write code from

Choose Good Variable Names

- Too Long
  - NumberOfPeopleOnTheUSOlympicTeam
- Too Short
  - N
- Just Right
  - NumTeamMembers
- Are short variable names always bad? NO
  - Loop control variables: i, j, k, idx
  - Temporary variables: temp
  - Names that are naturally short: x, y, z

Naming Conventions

- Separating words in identifiers
  - "Camel-case"
    - WebCrawler, documentMap
  - Separate words with underscores
    - Web_crawler, document_map
- First char of class name is upper-case
- First char of method name is lower case
- First char of variable name is lower-case
- Constant names are usually all upper-case with words separated by underscore

Creating Readable Names

- Names matter more to readers of the code than to the author of the code
- Don't use names that are totally unrelated to the entities they represent (e.g., “Thingy”)
- Don't differentiate variable names solely by capitalization
  - int temp;
  - char Temp;
- Avoid variables with similar names but different meanings
  - int temp;
  - Mountain timp;

Creating Readable Names

- Avoid words that are commonly misspelled
- Avoid characters that are hard to distinguish (1 and l)
- Avoid using digits in names (e.g., File1 and File2)
  - srcFile and destFile might be better
  - Sometimes Dr. Seuss naming is the best you can do (Thing1 and Thing2)

Abbreviation Guidelines

- Only abbreviate when you have to
- Remove non-leading vowels (Computer -> Cmptr)
- Or, Use the first few letters of a word (Calculate -> Calc)
- Don't abbreviate by removing just one character from a word (use "name" instead of "nam")
- Create names that you can pronounce
- Abbreviate consistently