McGill

Introduction to Software Engineering

ECSE-321

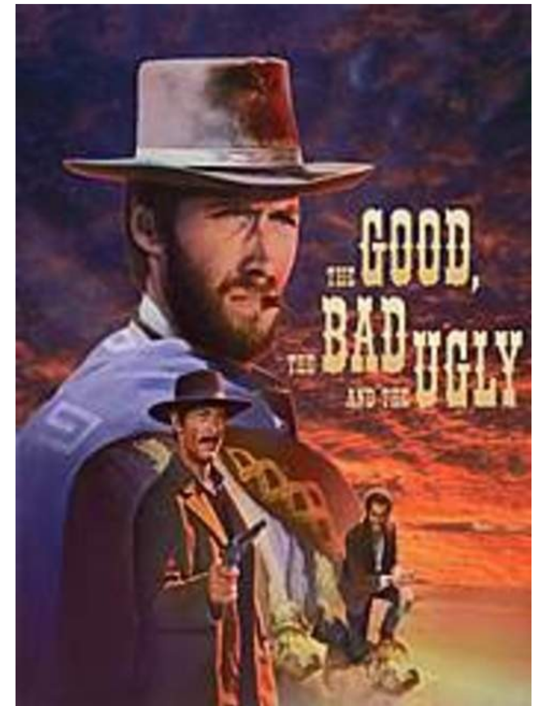
Unit 2 - Coding conventions

(from Prof. Rabbat's slides)

Course contents - Lectures

Introduction
Conventions and practices
Project management basics
Modeling with UML (basics)
Requirements
Analysis
System Design
Design Patterns
Object Design
Design to code
Quality assurance and testing
SE at large

Agenda



- Why coding conventions?
- Code conventions:

  Layout, comments, naming, practices

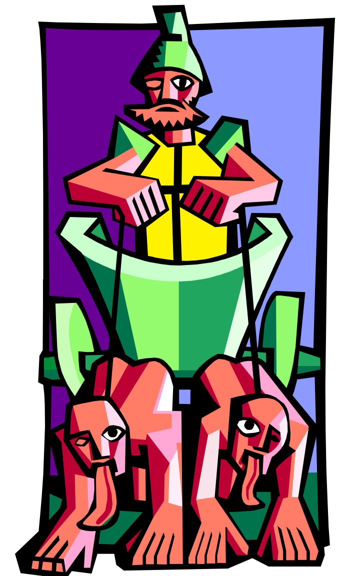
  - Not only reasons and principles
- Technical documentation

Why coding conventions?

- 80% of the lifetime cost of a piece of software goes to maintenance
  - Maintenance by multiple authors
  - Maintenance by a single author
  - Better readability → faster and deeper understanding
- Easier to integrate
- Facilitate reuse (porting to other contexts)
- Easier to find what you want
- **Communications**: code readers, communication by code, outsourcing, multiple teams
- Source code is sometimes a product
- Improves code quality

The catch

- For the conventions to work, every person writing software must conform to the code conventions. Everyone.
- People would rather quit than change style
- Consistency matters
- In this course we enforce strict coding conventions described in this unit



List of conventions

- Layout and indentation
- Comments
- Declarations and statements
- Naming conventions and practices

- Borrowed from Java SUN standard
  - http://java.sun.com/docs/codeconv/

Layout concepts

- Highlight logical structure of the code
  - Proximity: keep related things close together
  - Maintainability: ease the editing
  - Consistency: follow a set of rules
  - Compactness: make every word count

File names

- Java source : xxx.java
- Java bytecode: xxx.class

- Other “common” files:
  - README
  - GNUmakefile, makefile

Files

- Divide to sections
  - Separate sections by blank lines
  - Add comments identifying each section
- Max length is 2KLOC
- In a file:
  - One file, one public class
  - One file, one interface
  - Exception: private classes and interfaces used by a public class. Public class should always be the first class
- Java source files have the following ordering:
  1. Beginning comments
  2. Package and Import statements
  3. Class and interface declarations

Beginning comments

- All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:
- Don't overdo it
- Don't use endless version control information
- In this course – you can omit copyright notice

```
/*
* Classname
*
* Version information
*
* Date
*
* Copyright notice
*/
```

Package and Import statements

- The first non-comment line of most Java source files is a package statement
- Then list all import statements

- For example:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

Class and Interface Declarations

1. Class/interface documentation
2. Class/interface statement
3. Class/interface implementation comment (/*...*/), if necessary
4. Class (static) variables:
   1. public class variables
   2. protected
   3. package level (no access modifier)
   4. private
5. Instance variables
   1. public
   2. Protected
   3. Package level
   4. private
6. Constructors
7. Methods – group by functionality, not accessibility (other conventions exist)

```
/*
* @(#)Blah.java 1.82 99/03/18
*
* Copyright (c) 1994-1999 Sun Microsystems, Inc.
* 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
* All Rights Reserved.
*
* This software is the confidential and proprietary information of Sun.
*/


package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 *
 * @version 1.82 18 Mar 1999
 * @author Firstname Lastname
 */

public class Blah extends SomeClass {
    /* A class implementation comment can go here. */

    /** classVar1 documentation comment */
    public static int classVar1;

     /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;
```

Interface documentation

Interface statement

[Implementation comment]

Class and Interface Declarations

1. Class/interface documentation
2. Class/interface statement
3. Class/interface implementation comment (/*...*/), if necessary
4. Class (static) variables:
   1. public class variables
   2. protected
   3. package level (no access modifier)
   4. private
5. Instance variables
   1. public
   2. protected
   3. package level
   4. private
6. Constructors
7. Methods – group by functionality, not accessibility (other conventions exist)

Reminder:
Public = everybody
Protected = package+subclasses
Package level = only package
Private = only class

Indentation

- Spaces and tabs: Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).
- Line Length: Avoid lines longer than 80 characters
  - **Note:** documentation should be even shorter—generally no more than 70 characters.
- Wrapping Lines:
  - When an expression will not fit on a single line, break it according to these general principles:
    - Break after a comma
    - Break before an operator
    - Align the new line with the beginning of the expression at the same level on the previous line
    - If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead
- Here are some examples of breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);
var = someMethod1(longExpression1,
                someMethod2(longExpression2,
                        longExpression3));
```

Indentation II

- Two examples of breaking an arithmetic expression. Which is preferred?

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6; // PREFER
longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longname6; // AVOID
```

Indentation can be the source of many bugs, especially in algorithmic parts.

Indentation III

- Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronizedWorkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

- Always use common sense

Indentation IV

- Line wrapping for if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt(); //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Ternary expressions

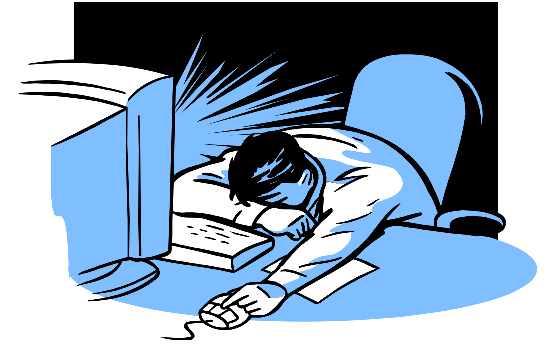
- Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;
alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

- A good method to write conditionals

White space



- Blank lines:
  - Two blank lines :
    - Between sections of a source file
    - Between class and interface definitions
  - One blank line:
    - Between methods
    - Between the local variables in a method and its first statement
    - Before a block or single-line comment
    - Between logical sections inside a method to improve readability

- Blank spaces:
  - A **keyword** followed by a parenthesis should be separated by a space

```
while (true) {
    ...
}
```

  - No blank space between a method name and its opening parenthesis
  - A blank space should appear after commas in argument lists
  - All binary operators (except “.”) should be separated from their operands by spaces
  - Unary operators (-,++, --) should never be separated from operands

```
a += c + d;
a = (a + b) / (c * d);
a = d++;
prints("size is " + foo + "\n");
```

  - The expressions in a for statement

```
for (expr1; expr2; expr3)
```

  - Casts should be followed by a blank space

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

List of conventions

- Layout and indentation
- Comments
- Declarations and statements
- Naming conventions and practices

Comments

- Implementation comments:
  - /* … */
  - // ….
  - Commenting out code
  - Particular implementation notes
- Documentation comments
  - /** … */
  - Extends to html documentation using javadoc http://java.sun.com/j2se/javadoc/
  - Describe the specification of the code
  - Implementation-free (what, not how)

To comment or not to comment?

- Don’t repeat code/directory information
  - Give overviews /additional information
- Assist the reader
- Discuss nontrivial or non-obvious design decisions
- Document coding surprises and dirty tricks
- Comments are parts of your SW
  - More comments → more maintenance
- Comments should not be enclosed in large boxes drawn with asterisks or other characters – no embellishments!

Implementation comments

- Block comments – tell a story
  - Describe files, methods, data structures and algorithms
  - May be used at the beginning of each file and before each method.
  - Can be used within methods
  - Inside a method - indented to the same level as the code they describe

```
/*
* This is a block comment
* This is the second line of the block comment
*/
```

- Single-Line Comments (one liners)
  - Indented to the level of the code that follows
  - Can’t be written in a single line → block comment
  - Should be preceded by a blank line

```
If (condition){

/* Handle the condition. */
...
}
```

- Trailing Comments – very short comments
  - Use to tie to data declarations, maintenance notes
  - Appear on the same line as the code they describe
  - Should be shifted far enough to separate them from the statements
  - More than one short comment in the same chunk of code ➔ all indented to the same tab setting.

```
if (a == 2) {
    return TRUE;                    /* special case */
} else {
    return isPrime(a);              /* works only for odd a */
}
```

- The // comment delimiter
  - Use for commenting out a complete line or only a partial line.
  - Don’t use on consecutive multiple lines for text comments

```
if (foo > 1) {
                            // Do something
...
}
else{
    return false;        // Explain why here.
}
```

  - Can be used in consecutive multiple lines for commenting out sections of code

```
//if (bar > 1) {
//
//      // Do something
// ...
//}
//else{
//      return false;
//}
```

- Caveat: often lots of commented code in legacy code

Some useful types of comments

- Subtitle – say what the code does
  - More abstraction
  - PDL (Program Definition Language)

```
// set parameters
myAccount.balance = DEFAULT_BALANCE;
myAccount.overdraftProtection = false;

// link to a checking account
myAccount.linkAccount(curSavingAccount);

.....
```

- Assertion comments
  - Requires (preconditions)
  - Modifies (data changed)
  - Effect (postcondition)

```
public static void addMax (Vector v, Integer x)
          throws NullPointerException, NotSmallException
  /* REQUIRES: All elements of v are integers
   * MODIFIES: v
   * EFFECTS: If v is null throws NullPointerException; if v
   *   contains an element larger than x throws
   *   NotSmallException; else adds x to v.
   */
```

- From [L], P61

- Data comments
  - Meaning of data fields and invariants
  - At declaration (for consistency)
  - End-line layout recommended here

```
protected point lowerLeft;     // lower left corner; .x, .y >=0
protected int width;           // .width >0
protected int length;          // .length >=0
```

To comment or not to comment ?

- How to achieve good commenting?
  - Relevance

```
/* set a to b */            /* make sure a's level is min(b,3)*/
a=b;                        a=b.truncate(LEVEL_3);
```

  - Maintainability – proximity

```
int securityClearance;    // 0-5, 5 highest
```

  - Document surprises

```
int securityClearance;    // 0-6, 5 highest 6 undefined
```

- When in doubt?
- Practice makes perfect …

Documentation Comments

- Generate documentation from in-line comments
  - Avoid redundancy
  - Easy and efficient way to document
- Describe Java classes, interfaces, constructors, methods, and fields
- Javadoc
  - Each doc comment is set inside the comment delimiters `/**...*/` (as opposed to `/* ...*/, // ...`)
  - Converts to html
- One comment per class, interface, or member.
- Should appear just before the declaration:

```
/**
* The Example class provides ...
*/
public class Example { ...
```

- Never use doc inside a method (will carry to the next)
- If you need to document something but don't want it to appear in the doc – add block/single line after declaration
- Will be used in project – see sun site & tutorial for details!

List of conventions

- Layout and indentation
- Comments
- Declarations and statements
- Naming conventions and practices

Declarations

- One declaration per line:

```
int level;                    // indentation level
int size;                     // size of table
```

is preferred over

```
int level, size;
```

- Do not put different types on the same line.

```
int foo, fooarray[];          //WRONG!
```

- Can use either one space between type and identifier or tabs, e.g.:

```
int       level;           // indentation level
int       size;            // size of table
Object    currentEntry;    // currently selected table entry
```

- Initialize local variables where they're declared
- Exception: initial value depends on some computation occurring first
- Put declarations only at the beginning of blocks ({block})
  - Don't wait to declare variables until their first use

```
void myMethod() {
    int int1 = 0; // beginning of method block
    if (condition) {
        int int2 = 0; // beginning of "if" block
        ...
    }
}
```

  - Exception: indexes of for loops

```
for (int i = 0; i < maxLoops; i++) { ... }
```

- Avoid local declarations that hide declarations at higher levels
- Also avoid similar names

```
int count;
...
myMethod() {
if (condition) {
    int count;      // BAD!
    ...
    }
    ...
}
```

- Classes and interfaces:
  - No space between a method name and the parenthesis “(“ starting its parameter list
  - Open brace “{” appears at the end of the same line as the declaration statement
  - Closing brace “}” starts a line by itself indented to match its corresponding opening statement
  - Except: when there is a null statement we should have “{}”

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}
    ...
}
```

- Methods are separated by a blank line

Statements

- One line one statement

```
v++;       // Correct
x--;       // Correct
v++; x--;       // Ugly - AVOID!
```

- return statement
  - avoid parenthesis unless needed:

```
return;

return myDisk.size();

return (size ? size : defaultSize);   // What does this do?
```

Compound statements

- Compound statements are lists of statements enclosed in braces

```
"{ statements }"
```

  - Enclosed statements should be indented one more level than the compound statement
  - The opening brace should be at the end of the line that begins the compound statement
  - The closing brace should begin a line and be indented to the beginning of the compound statement
- Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement
- Toughest bugs - statements without {…} accidentally introducing bugs due to forgetting to add braces

- The if-else class of statements should have the following form:

```
if ( condition) {
    statements;
}

if ( condition) {
    statements;
} else {
    statements;
}

if ( condition) {
    statements;
} else if ( condition) {
    statements;
} else {
    statements;
}
```

No more than 3 levels of nesting in if statements

- Avoid:

```
if ( condition) //Ugly - AVOID! THIS OMITS THE BRACES {}!
    statement;
```

Loops

- A for statement:

```
for (initialization; condition; update) {
    statements;
}
```

- An empty for statement (all the work is done in the initialization, condition, and update clauses) :

```
for (initialization; condition; update);
```

- A while statement:

```
while (condition) {
    statements;
}
```

- An empty while:

```
while (condition);
```

- A do-while statement:

```
do {
    statements;

}
while (condition);
```

Switch

- A switch statement should have the following form:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
    break;
}
```

- Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be
- **Every** switch statement should include a default case
- Add break in the default case - prevents a fall-through error if later another case is added

Try-catch

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

- A try-catch statement may also be followed by finally:

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

finally allows to "clean up" and is always executed

List of conventions

- Layout and indentation
- Comments
- Declarations and statements
- Naming conventions and practices

Naming conventions

- Packages: `com.sun.eng, ca.mcgill.ecse321`
  - The prefix of a unique package name is always written in all-lowercase ASCII letters and is separated by "." (prefix: com, edu, gov, mil, net, org, country code).
  - Note `java.util, java.util.map`
  - Subsequent components of the package name - project dependent (hierarchical)
- Classes: Nouns
  - First letter of each word capitalized
  - Use whole words – no nonstandard acronyms/ab/abb/abbr.

```
class  Account;
class  CheckingAccount;
```

- Interfaces – like classes

- Methods:
  - Verbs
  - in mixed case with the first letter lowercase, with the first letter of each internal word capitalized
  - Not more than 4 words

```
terminate();
terminateDownload();
terminateDownloadAfterBlueButtonPressed(); // Ugly
```

Variables

- Mixed case with a lowercase first letter, Internal words start with capital letters.
- Variable names should not start with underscore (“_”) or dollar sign (“$”) characters
- Names:
  - short yet meaningful
  - mnemonic– indicate its use to other users (no abbreviations)
- Avoid one-character variable names except “throwaways”
  - i, j, k, m, and n for integers
  - c, d, and e for characters.

```
int k;            // throwaway
int trmn8;        // cryptic
float myWidth;    // good
float myWindowWidthInInchesBiggerThan2;             // too long
```

Constants

- all uppercase with words separated by underscore (“_”)

```
static final int MIN_WIDTH = 4;
static final int MAX_WIDTH = 999;
static final int GET_THE_CPU = 1;
```

Practices

- Don’t make class variables public unless there is a reason
  - Exception: when class is really a struct
- Avoid using an object to access a class (static) variable or method. Use a class name instead.

```
classMethod();                 //OK
AClass.classMethod();          //OK
anObject.classMethod();        //AVOID!
```

- Use special comments when reviewing code:

`// XXX` (looks bogus)

`// FIXME` (a bug)

Each project should have a small set of special comments

- Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, in a for loop as counter values
- Avoid assigning several variables to the same value in a single statement.

```
fooBar.fChar = barFoo.lchar = 'c'; // Ugly - AVOID!
```

- Do not use the assignment operator if it can be confused with the equality operator.

```
if (c++ = d++) { // AVOID! (Java disallows)
    ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
    ...
}
```

- Do not use embedded assignments in an attempt to improve run-time performance

```
d = (a = b + c) + r; // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

- Parentheses: Use liberally in expressions involving mixed operators
- Avoid operator precedence problems (even if the operator precedence seems clear to you)

```
if (a == b && c == d)          // Ugly
if ((a == b) && (c == d))      // Good
```

- Returning values: make the structure of your program match the intent.

```
if ( booleanExpression) {
   return true;
} else {
   return false;
}
```

should instead be written as

```
return booleanExpression;      // Good
```

Similarly,

```
if (condition) {
   return x;
}
return y;
```

should be written as

```
return (condition ? x : y);    // Good
```

Defensive Programming

- Trust no one – no assumptions should be made
- Users are malicious
- Users are stupid
- Other programmers do not read the instructions
  - “You mean I always have to initialize this parameter?”
- Sanitize input data
- Never make code more complex than necessary. Complexity breeds bugs, including security problems
- Shell your code from other people meddling – encapsulate

Defensive Programming (Cont')

- If possible, reuse code
- Leave the code available to everyone on the Net or make sure the software was audited for security problems
- Encrypt/authenticate all data transmitted over networks
  - Use a proven encryption scheme
- All data is important until proven otherwise
- All code is insecure/unchecked/incorrect until proven otherwise
- If data is checked for correctness, verify if it's correct, not if it is incorrect

Hungarian notations

- Widely used in C/C++
- Out of fashion now, but still common
- Main idea: every variable has an identifier in its beginning explaining its type followed by the “given name”
- Examples:

`nSize` : integer

`dwLightYears` : double word

`f_interestRate` : floating point member of a class

`pFoo` : pointer

`szLastName` : zero-terminated string

`psz_Owner` : pointer to zero-terminated string, member of a class

Agenda

- Why coding conventions?
- Code conventions
- Technical documentation

Documentation

- Part of the products, often outlives code
- Includes: specifications, manuals, design, test plans, the code itself, notes, messages, rationale management, etc.
- What is important for the technical documentation for software?
  - Correctness, completeness, ...
  - Maintainability, consistency, ...
  - Conciseness, navigability, clarity, …
- Keep in mind the purpose - what needs to be done with the document
  - Consult for detailed information – precise, clear, comprehensive
  - Browse for specific information – easy to navigate
  - Modify often, keep consistent – maintainable
  - Manage rationale – concise and clear
- NOT entertainment
- Special role in many projects (more than a “technical writer”)
- **More is less**; Size doesn’t matter!

- Maintainability
  - Documentation needs to be updated and corrected, just like code
  - Documentation should be modular, just like code
- Avoid: redundant, repetitious text
  - Change ➔ consistency problems
  - Good: say something in one place, insert cross-reference in the other place
  - Bad: repeat important concepts many times
  - Redundancy is useful for quick high-level view
- Avoid: discussing multiple concepts
  - Good: isolate ideas, keep each sentence focused, summarize each paragraph in the opening sentence
- Avoid: update help files and user manuals independently
  - Good: obtain manuals automatically from help files, or obtain both from code (e.g. by javadoc)
  - Good: obtain both manually from the specification documents

- Conciseness
  - "… does not mean omission of detail"
  - Bad: quoting OO book in a design document
  - Good: rationale for your design decisions
- Navigability
  - How easy is it to find the relevant information?
  - Bad: unstructured text
  - Good: table of contents, cross-references, hyperlinks, index of terms
- Consistency
  - Consistent style helps retrieving information
  - Unlike literary writing, technical writing is not made better by style variation
- Clarity
  - E.g. avoid ambiguous expressions: "it", "this", ...