Documenting Conventions

Lecture 9

Documentation: conventions & style

- Documentation is essential for reuse and maintenance…
  …undocumented code is almost useless,
  (…undocumented assignments remain un-graded.)
- What to document?
- How to write documentation?
- Tool support?
- Semi-formal documentation?
  - Doug Lea's, Draft Java Coding Standard,
  - JavaSoft's, Code Conventions for the Java Programming Language
  - Pike & Kernigan, The Practice of Programming, Ch 2

Structure and documentation

- Packages
  - A new java package for each self-contained project or group of related functionality
  - An index.html file in each directory briefly outlining package purpose and structure

Structure and documentation

- Program Files
  - Place each class in a separate file. This applies even to non-public classes
  - except in the case of one-shot usages where the non-public class cannot conceivably be used outside of its context.

  - Begin each file with:
    1. file name & related information including copyright
    2. history table listing dates, authors, and summaries of changes.
    3. if the file is principal entry point for a package, briefly describe rationale for the package.
  - Immediately follow with:
    - package name
    - import list.

Structure and documentation

- Program Files
  - Example:

```
/*
   File: Example.java
   Date       Author       Changes
   Sep 1   95 Doug Lea     Created
   Sep 13 95 Doug Lea      Added new doc conventions
*/

package demo;
import java.util.NoSuchElementException;
```

Documenting classes and interfaces

- Classes and Interfaces
  - Write all /** … **/ comments using javadoc conventions.
  - Preface each class with a /** … **/ comment describing purpose of class, guaranteed invariants, usage instructions, & examples.
  - include any reminders or disclaimers about required or desired improvements.
  - Use HTML format,

```
<pre> … </pre>           -- preformatted
             <b> … </b>            -- bold
<em> … </em>             -- emphasis
             <code>… </code>       -- fixed width font
             <p>                   -- paragraph break
```

  - with added tags:

```
@author author-name
@version version number of class
@see string
@see URL
@see classname#methodname
```

Documenting classes and interfaces

- Classes and Interfaces

  Example:

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *      Window win = new Window(parent);
 *      win.show();
 * </pre>
 *
 * @see          awt.BaseWindow
 * @see          awt.Button
 * @version      1.2 31 Jan 1995
 * @author       Bozo the Clown
**/
class Window extends BaseWindow {
        ...
```

Documenting classes and interfaces

- Class Variables
  - Use javadoc conventions to describe nature, purpose, constraints, and usage of instances variables and static variables.
  - Use HTML format with added tags:

```
@see string
@see URL
@see classname#methodname
```

Example:

```
/**
 * The current number of elements.
 * must be non-negative, and less than or equal to capacity.
 **/
protected int count_;
```

Documenting classes and interfaces

- Methods
  - Use javadoc conventions to describe nature, purpose, preconditions, effects, algorithmic notes, usage instructions, reminders, etc. Use
  - HTML format, with added tags:

```
@param paramName description
 @return description of return value
 @exception exceptionName description
 @see string
 @see URL
 @see classname#methodname
```

Documenting classes and interfaces

- Methods
  - Be as precise as reasonably possible in documenting effects. Here are some conventions and practices for semi-formal specs.

`@return condition: (condition)`

- describes postconditions and effects true upon return of a method.

`@exception exceptionName IF (condition)`

- indicates the conditions under which each exception can be thrown. Include conditions under which uncommon unchecked (undeclared) exceptions can be thrown.

`@param paramname WHERE (condition)`

- indicates restrictions on argument values. Alternatively, if so implemented, list restrictions alongside the resulting exceptions, for example IllegalArgumentException. In particular, indicate whether reference arguments are allowed to be null.

Documenting classes and interfaces

- Methods

  `WHEN (condition)`
  - indicates that actions use guarded waits until the condition holds.

  `RELY (condition)`
  - describes assumptions about execution context. In particular, relying on other actions in other threads to terminate or provide notifications.

  `GENERATE T`
  - to describe new entities constructed in the course of the method.

  `ATOMIC`
  - indicates whether actions are guaranteed to be uninterfered with by actions in other threads (normally as implemented via synchronized methods or blocks).

  `PREV(obj)`
  - refers to the state of an object at the onset of a method.

Documenting classes and interfaces

- Methods

  `OUT(message)`
  - describes messages (including notifications such as notifyAll) that are sent to other objects as required aspects of functionality, or referrred to in describing the effects of other methods.

  `foreach (int i in lo .. hi) predicate`
  - means that predicate holds for each value of i.

  `foreach (Object x in e) predicate`
  - means that the predicate holds for each element of a collection or enumeration.

  `foreach (Type x) predicate`
  - means that the predicate holds for each instance of Type.

  `-->`
  - means `implies'.

Documenting classes and interfaces

- Methods
  `unique`
  - means that the value is different than any other. For example, a unique instance variable that always refers to an object that is not referenced by any other object.

  `fixed`
  - means that the value is never changed after it is initialized.

  `EQUIVALENT to { code segment }`
  - documents convenience or specialized methods that can be defined in terms of a few operations using other methods.

Documenting classes and interfaces

- Methods
  - Example:

```
/**
  * Insert element at front of the sequence
  *
  * @param element the element to add
  * @return condition:
  * <PRE>
  *  size() == PREV(this).size()+1 &&
  *  at(0).equals(element) &&
  *  foreach (int i in 1..size()-1)
  *      at(i).equals(PREV(this).at(i-1))
  * </PRE>
 **/
 public void        addFirst(Object element);
```

Documenting decls stmts & exps

- Local declarations, statements, and expressions
  - Use /* ... */ comments to describe algorithmic details, notes, and related documentation that spans more than a few code statements.
  - Example:

```
/*
 * Strategy:
 *     1. Find the node
 *     2. Clone it
 *     3. Ask inserter to add clone
 *     4. If successful, delete node
 */
```

  - Use Running // comments to clarify non-obvious code.

Documenting decls stmts & exps

- Local declarations, statements, and expressions
  - Do not comment obvious code; instead try to make code obvious!
  - Example:

```
int index = -1; // -1 serves as flag meaning the index isn't valid
```

Or, often better:

```
static final int INVALID= -1;
int index = INVALID;
```

  - Use any consistent set of choices for code layout, including:
    - Number of spaces to indent.
    - Left-brace ("{") placement at end of line
    - Maximum line length.
    - Spill-over indentation for breaking up long lines.
    - Declare all class variables in one place (at the top of the class).

Naming conventions

packages

- lowercase.
- recommended domain-based conventions described in the Java Language Specification, page 107 as prefixes.

files

- same base name as the public class they define.

classes

- CapitalizedWithInternalWordsAlsoCapitalized

Exception class

- ClassNameEndsWithException.

Interface

- When necessary to distinguish from similarly named classes: InterfaceNameEndsWithIfc.

Class

- When necessary to distinguish from similarly named interfaces:
  - ClassNameEndsWithImpl OR
  - ClassNameEndsWithObject

Naming conventions

constants (finals)

- UPPER_CASE_WITH_UNDERSCORES

private or protected

- firstWordLowerCaseButInternalWordsCapitalized OR
- trailingUnderscore_, OR
- thisVar (i.e. prefix with this), OR
- myVar (i.e. prefix with my), OR
- fVar (i.e. prefix with f)

static private or protected

- firstWordLowerCaseButInternalWordsCapitalized OR
- twoTrailingUnderscores__

local variables

- firstWordLowerCaseButInternalWordsCapitalized OR
- lower_case_with_underscores

Naming conventions

methods

- firstWordLowerCaseButInternalWordsCapitalized()

factory method for objects of type X

- newX

converter method that returns objects of type X

- toX

method that reports an attribute x of type X

- X x() or X getX().

method that changes an attribute x of type X

- void x(X value) or void setX(X value).

Javadoc

- Example

```
/*
 * @(#)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
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */
package java.blah;
import java.blah.blahdy.BlahBlah;
```

Javadoc

```
/**
 * 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;
```

Javadoc

```
    /** instanceVar1 documentation comment */
    public Object instanceVar1;

    /** instanceVar2 documentation comment */
    protected int instanceVar2;

    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;

    /**
     * ...constructor Blah documentation comment...
     */
    public Blah() {
        // ...implementation goes here...
    }
```

Javadoc

```
    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomethingElse documentation comment...
     * @param someParam description
     */
    public void doSomethingElse(Object someParam) {
        // ...implementation goes here...
    }
}
```

How To Write Unmaintainable Code

Last updated Saturday, 12-Feb-2000 11:06:30 PDT

Roedy Green

©1997-2000 Canadian Mind Products.

How To Write Unmaintainable Code

General Principles

To foil the maintenance programmer, you have to understand how he thinks. He has your giant program. He has no time to read it all, much less understand it. He wants to rapidly find the place to make his change, make it and get out and have no unexpected side effects from the change.

He views your code through a tube taken from the center of a roll of toilet paper. He can only see a tiny piece of your program at a time. You want to make sure he can never get the big picture from doing that. You want to make it as hard as possible for him to find the code he is looking for. But even more important, you want to make it as awkward as possible for him to safely ignore anything.

You might get the idea that every language feature makes code unmaintainable -- not so, only if properly misused.

How To Write Unmaintainable Code

Specific Techniques

1. Lie in the comments. You don't have to actively lie, just fail to keep comments as up to date with the code.
2. Pepper the code with comments like /* add 1 to i */ however, never document wooly stuff like the overall purpose of the package or method.
3. Make sure that every method does a little bit more (or less) than its name suggests. As a simple example, a method named isValid(x) should as a side effect convert x to binary and store the result in a database.
4. Use acronyms to keep the code terse. Real men never define acronyms; they understand them genetically.
5. In the interests of efficiency, avoid encapsulation. Callers of a method need all the external clues they can get to remind them how the method works inside.
6. If, for example, you were writing an airline reservation system, make sure there are at least 25 places in the code that need to be modified if you were to add another airline. Never document where they are. People who come after you have no business modifying your code without thoroughly understanding every line of it.

How To Write Unmaintainable Code

7. In the name of efficiency, use cut/paste/clone/modify. This works much faster than using many small reusable modules.
8. Never put a comment on a variable. Facts about how the variable is used, its bounds, its legal values, its implied/displayed number of decimal points, its units of measure, its display format, its data entry rules (e.g. total fill, must enter), when its value can be trusted etc. should be gleaned from the procedural code. If your boss forces you to write comments, lard method bodies with them, but never comment a variable, not even a temporary!
9. Try to pack as much as possible into a single line. This saves the overhead of temporary variables, and makes source files shorter by eliminating new line characters and white space. Tip: remove all white space around operators. Good programmers can often hit the 255 character line length limit imposed by some editors. The bonus of long lines is that programmers who cannot read 6 point type must scroll to view them.

How To Write Unmaintainable Code

10. Cd wrttn wtht vwls s mch trsr.

When using abbreviations inside variable or method names, break the boredom with several variants for the same word, and even spell it out longhand once in while. This helps defeat those lazy bums who use text search to understand only some aspect of your program. Consider variant spellings as a variant on the ploy, e.g. mixing International colour, with American color and dude-speak kulerz. If you spell out names in full, there is only one possible way to spell each name. These are too easy for the maintenance programmer to remember. Because there are so many different ways to abbreviate a word, with abbreviations, you can have several different variables that all have the same apparent purpose. As an added bonus, the maintenance programmer might not even notice they are separate variables.

How To Write Unmaintainable Code

11. Never use an automated source code tidier (beautifier) to keep your code aligned. Lobby to have them banned them from your company on the grounds they create false deltas in PVCS (version control tracking) or that every programmer should have his own indenting style held forever sacrosanct for any module he wrote. Insist that other programmers observe those idiosyncratic conventions in "his " modules. Banning beautifiers is quite easy, even though they save the millions of keystrokes doing manual alignment and days wasted misinterpreting poorly aligned code. Just insist that everyone use the same tidied format, not just for storing in the common repository, but also while they are editing. This starts an RWAR and the boss, to keep the peace, will ban automated tidying. Without automated tidying, you are now free to accidentally misalign the code to give the optical illusion that bodies of loops and ifs are longer or shorter than they really are, or that else clauses match a different if than they really do. e.g.

```
if(a)
    if(b)x = y;
else x = z;
```

How To Write Unmaintainable Code

13. Never put in any { } surrounding your if/else blocks unless they are syntactically obligatory. If you have a deeply nested mixture of if/else statements and blocks, especially with misleading indentation, you can trip up even an expert maintenance programmer.

15. Use very long variable names or class names that differ from each other by only one character, or only in upper/lower case. An ideal variable name pair is swimmer and swimner. Exploit the failure of most fonts to clearly discriminate between ilI1| or oO08 with identifier pairs like parseInt and parseInt or D0Calc and DOCalc. l is an exceptionally fine choice for a variable name since it will, to the casual glance, masquerade as the constant 1. Create variable names that differ from each other only in case e.g. HashTable and Hashtable.

How To Write Unmaintainable Code

16. Wherever scope rules permit, reuse existing unrelated variable names. Similarly, use the same temporary variable for two unrelated purposes (purporting to save stack slots). For a fiendish variant, morph the variable, for example, assign a value to a variable at the top of a very long method, and then somewhere in the middle, change the meaning of the variable in a subtle way, such as converting it from a 0-based coordinate to a 1-based coordinate. Be certain not to document this change in meaning.
17. Use lower case l to indicate long constants. e.g. 10l is more likely to be mistaken for 101 that 10L is.
18. Ignore the conventions in Java for where to use upper case in variable and class names i.e. Classes start with upper case, variables with lower case, constants are all upper case, with internal words capitalised. After all, Sun does (e.g. instanceof vs isInstanceOf, Hashtable). Not to worry, the compiler won't even issue a warning to give you away. If your boss forces you to use the conventions, when there is any doubt about whether an internal word should be capitalised, avoid capitalising or make a random choice, e.g. use both inputFileName and outputfilename. You can of course drive your team members insane by inventing your own insanely complex naming conventions then berate others for not following them. The ultimate technique is to create as many variable names as possible that differ subtly from each other only in case.

How To Write Unmaintainable Code

19. Never use i for the innermost loop variable. Use anything but. Use i liberally for any other purpose especially for non-int variables. Similarly use n as a loop index.

20. Never use local variables. Whenever you feel the temptation to use one, make it into an instance or static variable instead to unselfishly share it with all the other methods of the class. This will save you work later when other methods need similar declarations. C++ programmers can go a step further by making all variables global.
21. Never document gotchas in the code. If you suspect there may be a bug in a class, keep it to yourself. If you have ideas about how the code should be reorganised or rewritten, for heaven's sake, do not write them down. Remember the words of Thumper "If you can't say anything nice, don't say anything at all". What if the programmer who wrote that code saw your comments? What if the owner of the company saw them? What if a customer did? You could get yourself fired.

How To Write Unmaintainable Code

22.To break the boredom, use a thesaurus to look up as much alternate vocabulary as possible to refer to the same action, e.g. display, show, present. Vaguely hint there is some subtle difference, where none exists. However, if there are two similar functions that have a crucial difference, always use the same word in describing both functions (e.g. print to mean write to a file, and to a print on a laser, and to display on the screen). Under no circumstances, succumb to demands to write a glossary with the special purpose project vocabulary unambiguously defined. Doing so would be unprofessional breach of the structured design principle of information hiding.

23. In naming functions, make heavy use of abstract words like it, everything, data, handle, stuff, do, routine, perform and the digits e.g. routineX48, PerformDataFunction, DoIt, HandleStuff and do_args_method.

24. In Java, all primitives passed as parameters are effectively read-only because they are passed by value. The callee can modify the parameters, but that has no effect on the caller's variables. In contrast all objects passed are read-write. The reference is passed by value, which means the object itself is effectively passed by reference. The callee can do whatever it wants to the fields in your object. Never document whether a method actually modifies the fields in each of the passed parameters. Name your methods to suggest they only look at the fields when they actually change them.

How To Write Unmaintainable Code

27. I am going to let you in on a little-known coding secret. Exceptions are a pain in the behind. Properly-written code never fails, so exceptions are actually unnecessary. Don't waste time on them. Subclassing exceptions is for incompetents who know their code will fail. You can greatly simplify your program by having only a single try/catch in the entire application (in main) that calls System.exit(). Just stick a perfectly standard set of throws on every method header whether they could throw any exceptions or not.

31. Nest as deeply as you can. Good coders can get up to 10 levels of ( ) on a single line and 20 { } in a single method. C++ coders have the additional powerful option of preprocessor nesting totally independent of the nest structure of the underlying code. You earn extra Brownie points whenever the beginning and end of a block appear on separate pages in a printed listing.

Wherever possible, convert nested ifs into nested [? :] ternaries.

How To Write Unmaintainable Code

29. If you have an array with 100 elements in it, hard code the literal 100 in as many places in the program as possible. Never use a static final named constant for the 100, or refer to it as myArray.length. To make changing this constant even more difficult, use the literal 50 instead of 100/2, or 99 instead of 100-1. You can futher disguise the 100 by checking for a == 101 instead of a > 100 or a > 99 instead of a >= 100.

Consider things like page sizes, where the lines consisting of x header, y body, and z footer lines, you can apply the obfuscations independently to each of these and to their partial or total sums.

These time-honoured techniques are especially effective in a program with two unrelated arrays that just accidentally happen to both have 100 elements. There are even more fiendish variants. To lull the maintenance programmer into a false sense of security, dutifully create the named constant, but very occasionally "accidentally" use the literal 100 value instead of the named constant. Most fiendish of all, in place of the literal 100 or the correct named constant, sporadically use some other unrelated named constant that just accidentally happens to have the value 100, for now. It almost goes without saying that you should avoid any consistent naming scheme that would associate an array name with its size constant.

How To Write Unmaintainable Code

43. Keep all of your unused and outdated methods and variables around in your code. After all - if you needed to use it once in 1976, who knows if you will want to use it again sometime? Sure the program's changed since then, but it might just as easily change back, you"don't want to have to reinvent the wheel" (supervisors love talk like that). If you have left the comments on those methods and variables untouched, and sufficiently cryptic, anyone maintaining the code will be too scared to touch them.

45. Reverse the parameters on a method called drawRectangle(height, width) to drawRectangle(width, height) without making any change whatsoever to the name of the method. Then a few releases later, reverse it back again. The maintenance programmers can't tell by quickly looking at any call if it has been adjusted yet. Generalisations are left as an exercise for the reader.

46. Instead of using a parameters to a single method, create as many separate methods as you can. For example instead of setAlignment(int alignment) where alignment is an enumerated constant, for left, right, center, create three methods: setLeftAlignment, setRightAlignment, and setCenterAlignment. Of course, for the full effect, you must clone the common logic to make it hard to keep in sync.

How To Write Unmaintainable Code

48. Declare every method and variable public. After all, somebody, sometime might want to use it. Once a method has been declared public, it can't very well be retracted, now can it? This makes it very difficult to later change the way anything works under the covers. It also has the delightful side effect of obscuring what a class is for. If the boss asks if you are out of your mind, tell him you are following the classic principles of transparent interfaces.

55. Java offers great opportunity for obfuscation whenever you have to convert. As a simple example, if you have to convert a double to a String, go circuitously, via Double with new Double(d).toString() rather than the more direct Double.toString(d). You can, of course, be far more circuitous than that! Avoid any conversion techniques recommended by the Conversion Amanuensis. You get bonus points for every extra temporary object you leave littering the heap after your conversion.

57. Use exceptions for non-exceptional conditions. Routinely terminate loops with an ArrayIndexOutOfBoundsException. Pass return standard results from a method in an exception.

How To Write Unmaintainable Code

60. Use octal constants. Smuggle them into a list of decimal numbers like this:

```
array = new int []
        {
                111,
                120,
                013,
                121,
                };
```

72. If you cannot find the right English word to convey the meaning of a temporary variable (and you ignore the other suggestions about not giving meaningful names to variables), you may use a foreign language word as the name of the variable. For example, instead of using variable "p" for a "point", you may use "punkt", which is the German word for it. Maintenance coders without your firm grasp of German will enjoy the multicultural experience of deciphering the meaning. It breaks the tedium of an otherwise tiring and thankless job.