Software Testing and Maintenance
Modifying Code

Jeff Offutt

SWE 437
George Mason University
2008

Thanks to Jeff Lei, Susan Eisenbach, Jonathan Hardwick and Rolf Howarth

Programming for Maintainability

1. Change Overview
2. Understanding the Program
3. Configuration Management
4. Programming for Change
5. Program Style

Maintenance vs Development

- Maintenance change must reflect the goals and style of the existing system
  - Adding a new room costs more than adding a room in the first place
- We have to understand an existing system before changing it
  - How to accommodate the change ?
  - What are the potential ripple effects ?
  - What skills and knowledge are required ?

Major Maintenance Activities

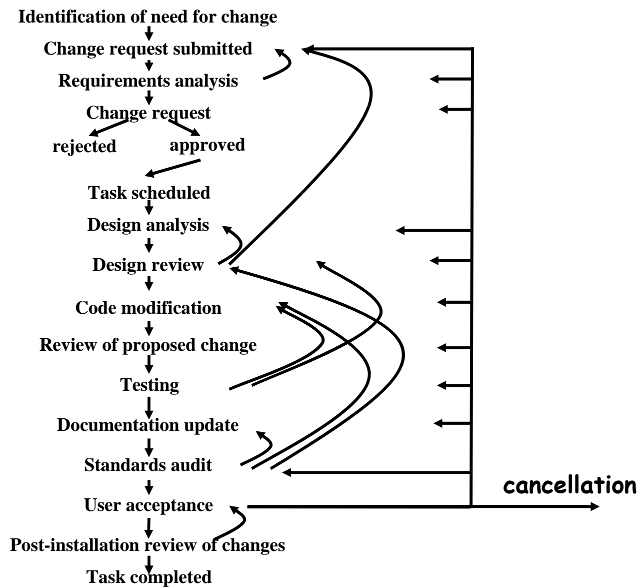
- Change identification
  - What to change, why to change
- Program understanding
  - How to make the change, determining the ripple effect
- Carrying out the change and testing
  - How to actually implement the change and ensure it is correct
- Configuration management
  - How to manage and control the changes
- Management issues
  - How to build a team

Development Models

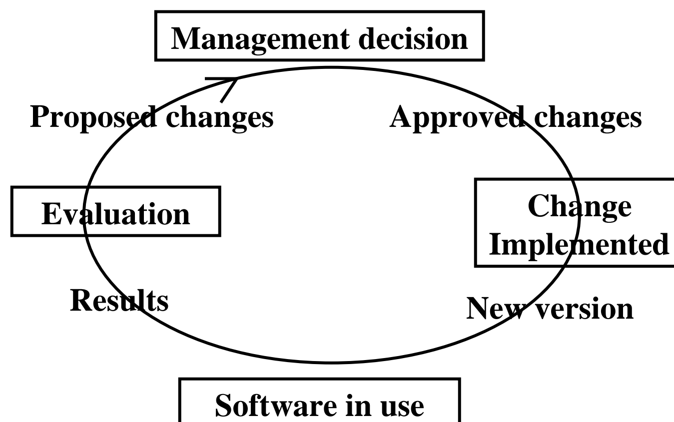
- Code-and-Fix
  - Ad-hoc, not well-defined
- Waterfall
  - Sequential, does not capture the evolutionary nature of software
- Spiral
  - Heavily relies on risk assessment
- Iterative
  - Incremental, but constant changes may erode system architecture

Code-and-Fix Model

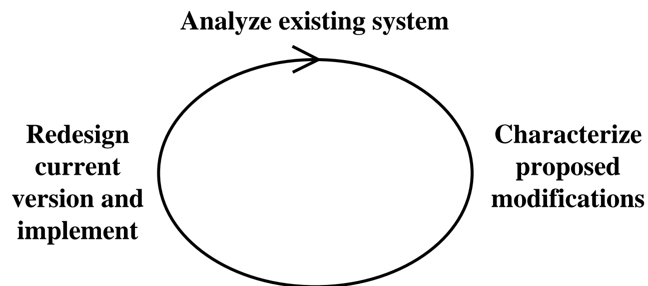
Osborne's Waterfall Model



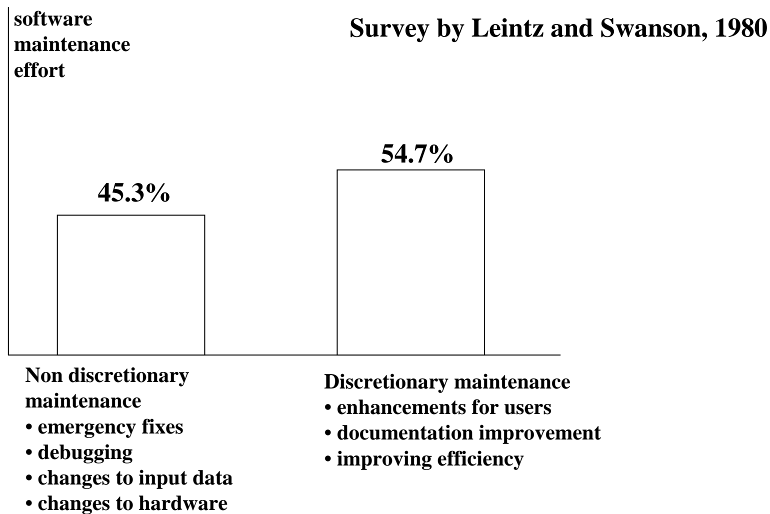
Boehm's Spiral Model



Iterative Enhancement Model



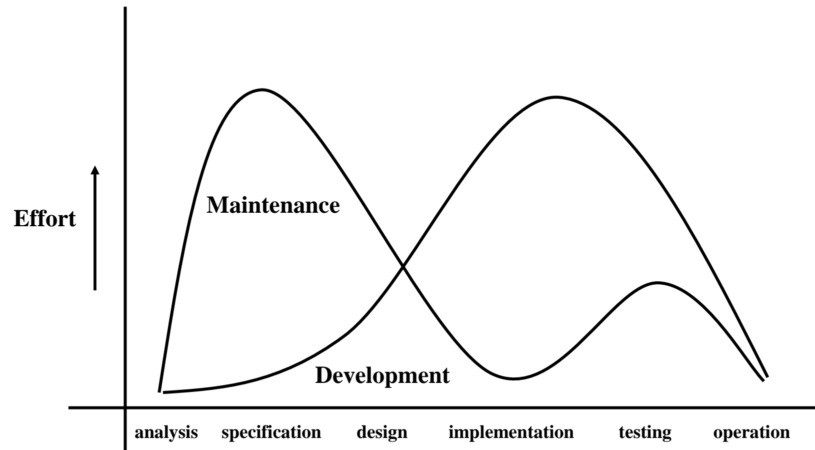
Maintenance Effort



Non discretionary maintenance
- **emergency fixes**
- **debugging**
- **changes to input data**
- **changes to hardware**

Discretionary maintenance
- **enhancements for users**
- **documentation improvement**
- **improving efficiency**

Maintenance vs Development Effort



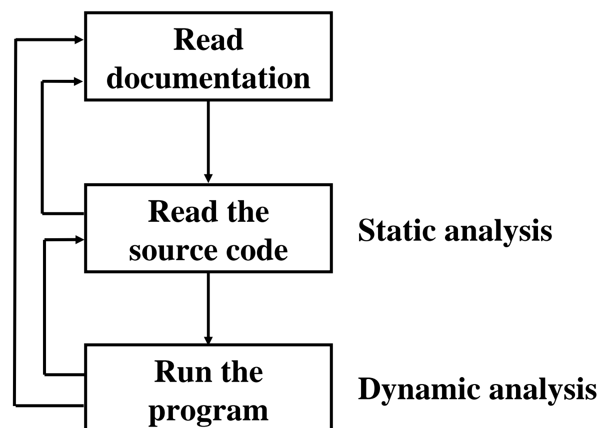
Programming for Maintainability

1. Change Overview
2. Understanding the Program
3. Configuration Management
4. Programming for Change
5. Program Style

What to Understand

- **Problem Domain**
  - Capture domain knowledge from documentation, end-users, or the program source
- **Execution Behavior**
  - Input-output relations, knowledge of data flow, control flow, and core algorithms
- **Cause-Effect Relations**
  - How different parts affect and depend on each other
- **Product-Environment Relation**
  - How the product interacts with the environment

Comprehension Process



Comprehension Process

- Knowledge base : Expertise and background knowledge
- Mental model : Encodes the current understanding
- Assimilation : Obtain information from various sources
- Comprehension Strategies :
  - Top-down : Start with the big picture, and then gradually work towards understanding the low-level details
  - Bottom-up : Start with low-level semantic structures, and then group them into high-level, more meaningful structures
  - Opportunistic : A combination of top-down and bottom-up

Factors that Influence Understanding

- Expertise : Domain knowledge, programming skills
- Program structure : Modularity, level of nesting
- Documentation : Readability, accuracy, up-to-date
- Coding conventions : Naming style, design patterns
- Comments : Quality, shall convey additional information
- Program presentation : Good use of indentation and spacing

Example of a Maintenance Task

- Small scale
- Illustrates the steps
- Quality measures (Meyer)
  - Maintainability
  - Efficiency
  - Conciseness
- These measures are frequently in conflict
- Many C programmers short-sightedly emphasize conciseness and efficiency over maintainability
  - This increases future programmer cost

Analysis Stage

- The parser should accept files with a .for extension, rather than files with .f or .F extensions
- Track down the C function responsible for validating the file names of input code (using ls, grep and vi)
- Determine detailed changes needed

Analysis
- Accept files ending with .for
- Find the C function
- Decide what to change

What Does This Code Do?

```
char *
c_name(s,ft)
char *s;
{       char *b, *s0;
        int c;
        b=s0=s;
        while(c=*s++)
                if (c==‘/’) b=s;
        if (--s<s0+3 || s[-2]!=‘.’
                    || ((c=*--s)!=‘f’&&c!=‘F’)) {
                infname=s0;
                Fatal("filename must end in .f or .F");
                }
        *s=ft;
        b=copys(b);
        *s=c;
        return b;
        }
```

Analysis

- **Accept files ending with .for**
- **Find the C function**
- **Decide what to change**

Looking at the Implementation

- **What the program component does**
  - **What is its role in the given change?**
- **How the component performs its function**
  - **How can we change it to perform its new role?**
- **Why the component operates in the way it does**
  - **Why is the system going to fall apart when you change this component?**

What? How? Why?

- **What ?**
  - **Function performs file-name filtering**
  - **Removes leading path names and replaces the first character of the suffix with the (integer, character?) held in ft**
- **How ?**
  - **Expand the program into a more comprehensible pseudo-C notation**
- **Why ?**
  - **Look at the users of the result of the function**
- **Try to do preventative maintenance at the same time so future alterations will be easier**
  - **Replace obscure code by either explicit code or comment it thoroughly**

Rewritten Code

```
s0 = s;
b  = s0;
c  = *s;
s  = s+1;
while (c>0) do /* while cur char not null */
   if (c=='/')
      b = s;
/* After the loop, b points to beginning of
   the file name, ignoring directory names */
c = *s;
s = s+1
s = s-1;
if (s < s0+3)
{
   infname = s0;
   Fatal ("filename must end in .f or .F")
}
```

```
elsif (s[-2] != '.')
{
   infname = s0;
   Fatal ("filename must end in .f or .F")
}
else
{
   s = s-1;
   c = *s;
   if (c != 'f' and c != 'F')
   {
      infname = s0;
      Fatal ("filename must end in .f or .F")
   }
}
*s = ft;
b  = copys(b);
*s = c;
return b;
```

Improvements

- **This clarifies the functionality by separating control flow from expression evaluation**
  - **More clarity could be provided by replacing the mixed use of s as an array and a pointer by an explicit array and array index.**
- **Reformatting (consistent indentation and spaces) makes it easier to read**
- **Variables whose purpose we don't understand cannot be changed**

  `*s = ft` and `b = copys(b);` cannot be changed
- **After changing, must run tests to check that the old suffixes are not accepted and the new suffix is accepted**
- **Tests may uncover additional faults (files of the form `xx.f` were accepted by the original program)**
- **The understanding about the function should be recorded in comments in the original code and the simplified code should be kept in a maintenance log**

Post-Maintenance Conditional

```
if (s < s0+5)
{
   infname = s0;
   Fatal ("filename must end in .for");
}
else if (s[-4] != '.')
{
   infname = s0;
   Fatal ("filename must end in .for");
}
else
{
   s = s-3;
   c = *s;
}
if !(c=='f' && s[1]=='o' && s[2]=='r')
{
   infname = s0;
   Fatal ("filename must end in .for");
}
```

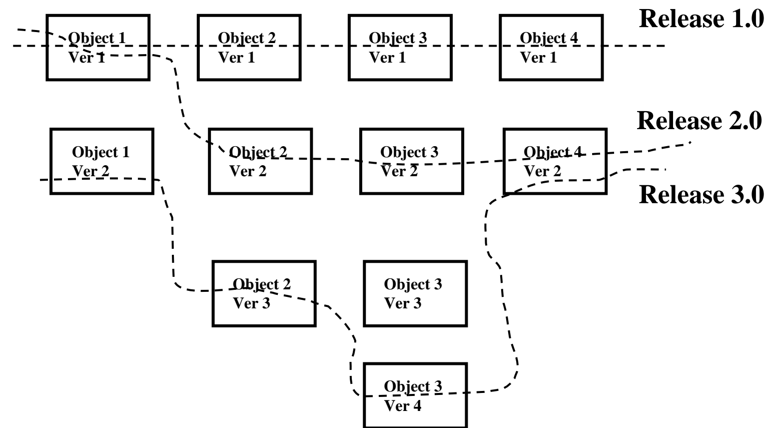
Programming for Maintainability

1. Change Overview
2. Understanding the Program
3. Configuration Management
4. Programming for Change
5. Program Style

Configuration Management

- **We must keep track of different versions of software components**
- **CM allows different releases to be made from the same code base**
- **CM allows us to retrieve old versions of a component**
- **CM allows us to "*freeze*" a version that is in use, then keep making modifications after the freeze**
- **CM supports effective team work, auditing, and accounting**
- **Major activities :**
  - **Identifying components and changes**
  - **Controlling the way changes are made**
  - **Auditing changes – making the current state visible**
  - **Status accounting – recording and documenting all activities**

Version Control



Building

Compiling and linking dozens or hundreds of software components to make a complete system

- **Performed very frequently**
  - **Often nightly**
  - **Must be followed by a standard set of tests**
- **Incremental building : only rebuild objects that have been changed or have had a dependency change**
- **Consistency : must use appropriate versions of the source files**
- ***Makefiles* are often used to declare dependencies between different modules**

Change Control

- Decide if a requested change should be made
  - Is it valid?
  - Does the cost of the change outweigh its benefit?
  - Are there any potential risks?
- Mange the actual implementation of the change
  - Allocate resources, record the change, monitor the progress
- Validate that the change is done correctly
  - Ensure that adequate testing be performed
- All change should be approved by
  - Team leader
  - Project manager
  - Software architect

Programming for Maintainability

1. Change Overview
2. Understanding the Program
3. Configuration Management
4. Programming for Change
5. Program Style

Programming for Maintainability

- **In 1980 – the most important concerns were about speed and size**
  - **Computer hardware was slow**
  - **We never had enough memory**
  - **Screens were small**
  - **Programming tools were primitive (only text editors)**
- **In 2008 – the overriding concern is to make it easier for someone to change the program later**
  - **Most importantly : readability !!**
  - **Even debugging is much easier if you can read the program**
  - **Readable code is more reliable and secure**
- **Sometimes the goals directly conflict**
- **If you happen to be writing real-time, embedded software … speed and size are still important**
  - **But reliability can save lives in safety-critical software**

Use Good Tools

- **Notepad and javac are NOT good programming tools**
- ***Excise tasks* are things that do not directly solve the problem, but that must be done to support weak technology or processes**
  - **Compiling, editing, storing, viewing, …**
  - **Good tools reduce the amount of time you have to spend on excise tasks**
  - **A major goal of software engineering research is to reduce the amount of time developers spend on excise tasks**
  - **1980 : 80% excise**
  - **2008 : 25% excise**
- **IDE : *Integrated Development Environments* provide built-in editors, compilers and other tools for managing large systems**
- **Successful professionals rely on symbolic debuggers at every step**
- **Eclipse is free and very widely used**
- **High quality editors like emacs and vim allow many changes to be made very quickly**

Avoid Unnecessary Fancy Tricks

- **Write as if for a human, not for a compiler**
- **Fully parenthesize all expressions**
- **Only use optimizations if you are sure they will make a difference**
- **In 1980, the control flow of individual functions dominated the running time of a program**
  - **Hence the undergraduate CS emphasis on analysis of algorithms**
- **In 2008, the overall architecture of the system usually washes out any effect of optimizing individual methods**
  - **I once worked on a project where a colleague spent 6 weeks optimizing methods that accessed data from files – saving about 4% of execution time**
  - **I then looked at his overall design, and found that he was reading the entire file into memory each time through an outer loop – I was able to spend 4 hours and saved over 40% of execution time**
  - **However, nobody could ever understand his micro-optimizations …**

Documentation

- **`javadoc` automatically generates documentation for Java**
- **Information such as the `author` and `version` can be added into the Java source and `javadoc` will extract it automatically**
- **Add a comment every time you have to stop and think**
- **Write pseudocode as comments, then write the method**
  - **Faster**
  - **More reliable**
  - **Ready-made, free, documentation**
- **Always include header blocks for each method**
- **Document any assumptions**
- **Document all variables that can be overridden by child methods**
- **Document reliance on default and superclass constructors**

Tips for Writing Maintainable Java

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
- C.A.R. Hoare

Tips for Writing Maintainable Java

- **Programmers need to be precise**
  - **Sloppy style looks like sloppy thinking**
- **Test pieces continually – the "build and test" cycle is essential to well engineered software**
  - **Automate your tests with main methods and tools like `junit`**
- **KISS : Keep It Simple and Stupid**
  - **We write once, but read dozens of times …**
- **Do not optimize until you know what needs to be optimized**
  - **Keep the un-optimized version in comments for documentation**
- **Use design patterns when they fit**
  - **Don't bang on jigsaw puzzle pieces to make them fit …**
- **Don't test for error conditions you don't know how to handle**
  - **Let them propagate to someone who does**
- **If you cannot develop these habits, look for non-developer jobs**

Documentation Tips

- **When the design changes during implementation (and it always does!) don't forget to change the design documentation !**
  - **Not doing so is unprofessional – selfish, lazy and short-sighted**
- **When you change the implementation – Update the comments !**
- **Why a method does something is usually more important than what and what is more important than how**
- **Do not over-document – comments should supplement the implementation, not dominate it**
- **Don't waste time stating what's obvious**
  - `setList (List list); // This method sets a list`
  - **What list ? Why ? When ? What are the preconditions ?**
- **Use a version control system that maintains an edit history**
  - **Enter comments explaining why each change was made**

Java Specific Tips

- **Always implement both or neither `equals()` or `hashCode()`**
  - **Not doing so can cause some very subtle faults**
- **Always override `toString()` to produce a human-readable description of the object**
  - **If `o1.equals (o2)` is true, `o1.toString()` should equal `o2.toString()`**
- **If `equals()` is called on the wrong type, return false, not an exception**
- **If your class is cloneable, use `super.clone()`, not `new()`**
  - **`new()` will break if another programmer inherits from your class**
- **Don't keep two copies of the same data**
- **Threads are hard to get right and harder to modify**
- **Don't add error checking the VM already does**
  - **Array bounds, null pointers, etc**
  - **Let the VM raise the exception to your handler**

Java Specific Tips – Immutability

An object is *immutable* if its state cannot change after it is constructed

- **Immutable objects are simpler, safer, and more reliable**
  - **They sacrifice some speed … but not much**
- **Basic types such as keys should be *immutable* – their values cannot change after being constructed**
- **Immutable objects should be declared `final` so that they can be used just as elementary types**
- **Immutable objects are especially useful in concurrent software because they cannot be corrupted by thread interference**

Modularity and Coupling

A primary goal of OO software is to reduce *coupling* – how much methods and objects depend on each other

- **The goal of reducing coupling led to most major programming advances in the last 40 years**
  - **Macro assemblers, high level languages, structured programming, ADTs, data hiding, inheritance, polymorphism, CASE tools, UML, JavaBeans, XML, the web, J2EE, …**
- **The most common theme is to increase modular components**
  - **Method, class, file, package, …**
- **It must be possible to describe each component in a very concise way**
  - **"This is a stack" …**
  - **If not, nobody will ever be able to maintain or reuse it !**

Modularity and Coupling (2)

- **If you find a module that cannot be described concisely, "wrap" another abstraction layer around it**
  - **Rewrite it later**
- **Always carefully consider which modules refer to which other modules**
  - **Changing the implementation should be independent of other modules**
  - **The implementation changes regularly**
  - **Changing the API should be very rare**
- **Hide all complexity behind simple APIs**

The real test of design is when we have to change the implementation

Keep It Simple, and Stupid

- **Long names are simple, short names are complicated**
- **Long methods are not simple**
  - **Good programmers write less code, not more**
- **Bad designs lead to more and longer methods**
- **Don't add generalizations unless you need them**
- **Ten programmers deliver twice as much code, four times as many faults, and half the functionality that 5 programmers do**

Classes and Objects

The point of OO design is to look at nouns (data) first, then verbs (algorithms and methods)

- **Think about what an object is, not what the class does**
  - **Be suspicious of classes whose names are verbs**
- **An object is defined by its state – the class defines behavior**
- **Lots of switch statements often mean the class is trying to do too many things**
  - **Consider replacing the class with a base class and children classes**
  - **Use type parameterization from Java 1.5 (Generics)**
- **If a method doesn't use any class instance variables, make it static**
- **Don't confuse inheritance and aggregation**
  - **Inheritance should implement "is-a"**
  - **Aggregation should implement "has-a"**

Programming for Change Summary

The cost of writing a program is a small fraction of the cost of fixing and maintaining it

…

Don't be lazy / selfish

…

***Be an engineer* !**

Remember that
complexity
is the number one enemy of
maintainability

Programming for Maintainability

1. Change Overview
2. Understanding the Program
3. Configuration Management
4. Programming for Change
5. **Program Style**

Don't Just Code, Style It !

- A crucial part of maintainability is readability
- To make your code readable
  - Select a set of style conventions
  - Follow them strictly !
- When making maintenance changes, follow the existing style
  - Even if you do not like it
- Lots of style conventions are available
- Very little difference – the most important thing is to be consistent

General Style Principles

- **A famous study way back in the 1960s asked "how far should we indent"**
  - **2—4 characters is ideal**
  - **Fewer is hard to see**
  - **More makes programs too wide**
- **Never use tabs – they show up differently in every editor and printer**
  - **The only worse habit : mixing tabs and spaces**
- **Use plenty of white space**
  - `newList(x+y)=fName+space+lName+space+title;`
  - `newList (x+y) = fName + space + lName + space + title;`
- **Don't put more than one statement per line**
- **Design your names so that they make sense**

Key Aspects For Guidelines

- **Case for names**
  - **Variables, methods, classes, …**
- **Guidelines for choosing names**
- **Width, special characters, and splitting lines**
- **Location of statements**
- **Organization of methods and use of types**
- **Use of variables**
- **Control structures**
- **Proper spacing and white space**
- **Comments**

Sample guidelines from:
Java Programming Style Guidelines
Geotechnical Software Services

General Naming Conventions

- **Names representing packages should be in all lower case**
- **Names representing types must be nouns and written in mixed case starting with upper case**
- **Variable names must be in mixed case starting with lower case**
- **Names representing constants (final variables) must be all uppercase using underscore to separate words**
- **Names representing methods must be verbs and written in mixed case starting with lower case**
- **Abbreviations and acronyms should not be uppercase when used as name**
- **Private class variables should have underscore suffix**
- **Generic variables should have the same name as their type**
- **All names should be written in English**
- **Variables with a large scope should have long names, variables with a small scope can have short names**
- **The name of the object is implicit, and should be avoided in a method name**

Specific Naming Conventions

- **The terms get/set must be used where an attribute is accessed directly**
- **is prefix should be used for boolean variables and methods**
- **The term compute can be used in methods where something is computed**
- **The term find can be used in methods where something is looked up**
- **The term initialize can be used where an object or a concept is established**
- **JFC (Java Swing) variables should be suffixed by the element type**
- **Plural form should be used on names representing a collection of objects**
- **n prefix should be used for variables representing a number of objects**
- **No suffix should be used for variables representing an entity number**
- **Iterator variables should be called i, j, k etc**

Specific Naming Conventions (2)

- **Complement names must be used for complement entities**
- **Abbreviations in names should be avoided**
- **Negated boolean variable names must be avoided**
- **Associated constants (final variables) should be prefixed by a common type name**
- **Exception classes should be suffixed with Exception**
- **Default interface implementations can be prefixed by Default**
- **Singleton classes should return their sole instance through method getInstance**
- **Classes that creates instances on behalf of others (factories) can do so through method new[ClassName]**
- **Functions (methods returning an object) should be named after what they return and procedures (void methods) after what they do**

Files

- **Java source files should have the extension .java**
- **Classes should be declared in individual files with the file name matching the class name**
  - **Secondary private classes can be declared as inner classes and reside in the file of the class they belong to**
- **File content must be kept within 80 columns**
- **Special characters like TAB and page break must be avoided**
- **The incompleteness of split lines must be made obvious**

Statements

- The package statement must be the first statement of the file. All files should belong to a specific package
- The import statements must follow the package statement
  - import statements should be sorted with the most fundamental packages first, and grouped with associated packages together and one blank line between groups
- Imported classes should always be listed explicitly
- Method modifiers should be given in the following order :
  - <access> static abstract synchronized <unusual> final native
  - <access> modifier must be the first modifier

Types and Variables

- Type conversions must always be done explicitly
  - Never rely on implicit type conversion
- Array specifiers must be attached to the type not the variable
- Variables should be initialized where they are declared and they should be declared in the smallest scope possible
- Variables must never have dual meaning
- Class variables should never be declared public
- Arrays should be declared with their brackets next to the type
- Variables should be kept alive for as short a time as possible

Control Flow

- Only loop control statements must be included in the for() construction
- Loop variables should be initialized immediately before the loop
- The use of do-while loops can be avoided
- The use of break and continue in loops should be avoided

- Complex conditional expressions must be avoided. Introduce temporary boolean variables instead
- The nominal case should be put in the if-part and the exception in the else-part of an if statement
- The conditional should be put on a separate line
- Executable statements in conditionals must be avoided

Layout and Spacing

- Basic indentation should be 3
- Left curly braces should appear directly under the control word
- else statements should appear directly under the if
- Each case statement in a switch should be indented
- Single statement control statements can be written without brackets

White Space

- Inter-line white space
  - Operators should be surrounded by a space character
  - Java reserved words should be followed by a white space
  - Commas should be followed by a white space
  - Colons should be surrounded by white space
  - Semicolons in for statements should be followed by a space character
- Method names can be followed by a white space when it is followed by another name
- Logical units within a block should be separated by one blank line
- Methods should be separated by three blank lines
- Variables in declarations can be left aligned
- Statements should be aligned wherever this enhances readability

Comments

- Tricky code should not be commented but rewritten
- All comments should be written in English
- Javadoc comments should have the following form:
- There should be a space after the comment identifier
- Use // for all non-JavaDoc comments, including multi-line comments
- Comments should be indented relative to their position in the code
- The declaration of anonymous collection variables should be followed by a comment stating the common type of the elements of the collection
- All public classes and public and protected functions within public classes should be documented using the Java documentation (javadoc) conventions

Summary

- **The little things that programmers do have a major impact on readability**
- **Readability has a major impact on maintainability**
- **Maintainability is the primary determining factor in the long-term cost of the system**

The minor little decisions that you make as programmers determine how much money your company makes

That is what engineering means !