Clean Code

Topic 9

Brett Hamm, Julia Zochodne, Louay Rafih,

Ali El-Dani,Winston Chan

CONTENTS

ABSTRACT

Clean code refers to the quality of software, usually pertaining to the readability and understandability of source code. We wouldn't be where we are today without continually improving the quality of software using refactoring. One main reason for refactoring is maintainability. It is much easier to fix bugs in code when people can actually understand it, and locate things in a timely manner. Another major reason for refactoring code is to increase the extensibility of the software. It is much easier to extend the capabilities of an application through the use of well-structured design patterns, and provides another level of flexibility.

It is said that beauty is in the eye of the beholder. In some sense, the same can be said about clean code. An experienced programmer can easily spot unreadable source code. People also develop separate opinions on whether or not code is efficient, simple, or well structured. However, most programmers can probably agree on whether a piece of code is clean or not.

Although there are many factors contributing to clean code, some are universal and apply to any domain or programming language, while others are specific to the task at hand. In this paper, we will focus on general code refactoring techniques that do not change the functionality of the code, but change its structure and usability. These techniques can be used in any range of software and or scripting applications.

COMMON ASPECTS OF BAD CODE

In order to write cleaner code it is first necessary to recognize signs of bad code. We will begin with some of the more obvious features of bad code, which are often described as having a bad smell. We can define a code smell as an indication that there are problems with the quality of the source code (Atwood, 2006). The following basic code smells are some of the most general and apply to most programming languages.

Code duplication is an indication of poor quality code. It indicates that the design of the software is not sufficiently modular (McConnell, 2004). Duplication can be easily spotted with code that is identical. It can also occur when similar steps are used in code that appears to be different. Duplication also results in code that is more difficult to modify and maintain (McConnell, 2004). If changes need to be made to code that is duplicated, corrections have to be applied in multiple places, which increases the probability for mistakes.

The use of very long functions is another common aspect of foul smelling code. Functions that are hundreds of lines long are more difficult to understand and more difficult to debug (Atwood, 2006). It is more difficult to keep track of local variables that appear in very long functions as they may not have been used for hundreds of lines!

A similar issue is the use of functions with a long list of parameters. The complexity of a function increases with the number of parameters, which both increases the coupling of the code and decreases its understandability (Atwood, 2006).

There are further smells specific to object oriented programming. These are particularly relevant as many of us spend a significant amount of time programming using the object oriented paradigm.

A class that lacks cohesion, or that takes on a variety of unrelated responsibilities, is a code smell. It encounters the same issue as very long functions as it is more difficult to understand and to debug, and should be broken down into smaller more cohesive classes (McConnell, 2004).

It is also a code smell if classes are overly intimate with one another, that is, they expose too much information to one another (McConnell, 2004). A lack of information hiding means that the code is more complex. Also, if classes are overly dependent on one another the program becomes less robust and more difficult to modify or extend.

For similar reasons, a class with many public data members is a code smell: it also violates the principle of information hiding and make code more complex and less flexible.

These are only a few examples of code smells. These can be used as part of a more extensive checklist when deciding whether or not to refactor code.

COMMON EFFECTS OF BAD CODE ONTO APPLICATIONS AND CUSTOMERS

We have already considered some aspects of bad code from the programmer's perspective. These are indications of internal code quality visible only to the programmer. However, we will now discuss some symptoms of bad code from the point of view of the application. These are external symptoms that a customer would actually be able to observe. Some characteristics of external software quality include obvious positive qualities such as correctness and usability, as well as characteristics such as adaptability and robustness (McConnell, 2004).

It is easy to see how bad code that is difficult to maintain and understand might lead to an increased number of software defects – programmers may take longer to find and correct defects in bad code (McConnell, 2004). This in turn impairs the correctness and reliability of the software from a customer's perspective. Similarly, code that is rigid and not easily changed from the programmer's perspective can result in an application that is less adaptable to different environments and less adaptable to a customer's requests (McConnell, 2004).

WRITING CODE THAT IS EASILY UNDERSTOOD

Everything in software contains names of variables, functions, arguments, and classes. Since everything is named, we need to name things so that it is easy to understand. To help understand what a function, variable or class does, the name should have a meaning or be descriptive. If a name is not descriptive, for example variable int d, then then it is not considered a good name. Proper naming will make it easier to understand, debug, maintain and change code. An example of naming variables properly is:

1.) incomePerMonth over ipm
2.) circleRadius over cr

A simple guideline to follow when naming variables is to begin with a lowercase and use camel case. Constants should use all capital letters and underscores. One of the reasons for naming constants with all capitals is so that it can easily be located in a body of text. Classes should have a noun that represents the class, and should not consist of a verb. A method should consist of a verb that starts with a lowercase letter and use camel case. You should make sure that names do not vary in small ways to avoid confusion. For example handlingParsingOfString vs handlingParsingOfStringToInt, or getTreatment(), getTreatments() and getTreatmentInfo. Names should also not consist of a single letter, because this will make it difficult to search for in your code body. One exception to this guideline is using single letter local variables inside short methods. Another part of clean code is to pick one word for a concept. An example of what not to have is to use stop, stay, halt, pause, and freeze for a single concept. Functions should be small, so that it can be easily understood and should only do one thing. The purpose of small functions is to shorten larger chunks of code that can complicate and confuse someone reading the code.

When reading code, comments can help us understand what is happening. They should be used to make sense of code that is hard to describe with naming alone. However, comments can also be useless and misleading. For example, they can be used as an attempt to explain poorly written code. If a method or function is confusing and badly written, it is better to re-write the code rather than only add a comment. Clean code with few comments is better than complex code that is not easily understood. Code should explain what is being done, and comments should explain why. Comments should not restate code, as this is redundant. An example of this is "*int count = 1 //set count to 1.*" Each method should be documented with its inputs, outputs and side effects. This is a made up example :

```
/** Description of aRandomMethod(int aNumber, int bNumber)
*
* @param aNumber                Description of aNumber
* @param bNumber                Description of bNumber
* @return                       Description of a return value
*/
```

Another part of good commenting practice is to not have commented out code. It will raise questions as to why it is commented out and if it is important or not. It is better to delete code, as most software development uses source control that will remember the code. Changes can be reverted back too, and the code will not be lost should you require it in the future. The reason for having clean code is because it is the only part of software that describes accurate information.

When formatting code, you need to watch your whitespace and indenting. The purpose of whitespace is to make it easier to read. Logic can be separated by blank lines. An example of spacing is

"for(int i=0;i<N;i++) vs. for (int i = 0; i < N; i++) "
(Princeton University, 2013).

In the example above, it is easier to read the right hand side than the left hand side. The left hand side seems more cluttered. A rule to follow is to space after a comma or after a statement. As this is up to the programmer, it should be made legible. Some rules for indenting are:

1.) Avoid lines longer than 80 characters
2.) Wrap your lines

(Princeton University, 2013).

An example of proper indenting:

"//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();
}"
```
(Oracle, 2006).

WRITING CODE THAT IS TESTABLE

Apart from writing that code that is easy to understand, code should also be testable. Test driven development and unit testing are important parts of agile software development. Tests are important because it helps you understand your program and reduces the code complexity. It can also help you realize what effect code changes have on specific parts of your software.

Unit tests examine parts of your code. Code that is easy to test should be loosely coupled, meaning its components have little or no knowledge of other separate components. Code should also separate logic from factories. Logic consists of declarative statements, while factories are objects that create other objects. The factories are full of new operators that build objects, while the logic does not contain any new operators. Tests focus on the application logic, and if this part is separate, it is easier to test. Code should be asking for things instead of searching:

> "If you are a House class then in your constructor you will ask for the Kitchen, LivingRoom, and BedRoom, you will not call the "new" operator on those classes (see 1). Only ask for things you directly need," (Google blog, 2008).

And also "If you are a CarEngine, don't ask for FuelTank, only ask for Fuel" (Google Blog, 2008).
Code in the constructor should not have the constructor doing many different things. It should be for instantiating an object and any of its dependencies. This makes testing easier because, in the constructor, you do not want to navigate through every instance you create for a certain property because it can be complex. If you have multiple instances of an object and that object has a variable, you want to make sure that variable is not global. This is because the point of testing is to test individual parts and global variables would persist through all tests.

Maintainability of code is an important issue every application faces. It defines the speed and ease of which developers can read, understand, and modify code. Code maintenance can be divided into four categories. These categories are corrective maintenance, preventive maintenance, predictive maintenance, and maintenance prevention (Blanchard, Benjamin and Verma, 1995).

Corrective maintenance includes cases of unscheduled maintenance after a system failure. This is a category that developers try to avoid by developing a well-structured design. However, in most cases this does not happen, meaning that the majority of maintenance problems fall under this category.

Preventive maintenance is when scheduled maintenance is performed to prevent future failures. Larger systems spend large amounts of time and money in this category to ensure their users do not experience failures (Blanchard, Benjamin and Verma, 1995).

Predictive maintenance is the process of monitoring equipment to evaluate their status, so that any degradation noticed can be quickly maintained (Blanchard, Benjamin and Verma, 1995).

Maintenance prevention is the practice of minimizing maintenance downtime and requirements for support resources. These four categories emphasize the four types of maintenance that a system can undergo (Blanchard, Benjamin and Verma, 1995).

There is a close connection between clean code and maintainability. Developers constantly use the acronym KISS (Keep It Simple Stupid), because having code that is too complex to read or even understand hinders the ability of other developers to correct the failure. Corrective maintenance is always a problem for companies as "A system that is expensive or risky to change is an opportunity to cost your business" (Miller, 2006). Although it is not always possible to develop simple and well-structured code, good naming conventions can make a world of difference. Developers that are knowledgeable about a system should be able to quickly understand the purpose of certain functionality. Companies realize this issue and are proactive to solve the problem by performing preventative maintenance.

Scheduling maintenance on systems allows companies to constantly extend their systems. Doing so requires a well-structured system that contains the following aspects; documentation, descriptive naming conventions, reducing coupling, and code testing. Scheduled maintenance allows companies to constantly adapt to the ever-changing market, and allowing relatively fast extensions on already existing systems gives companies the edge. However, constantly adding extensions into poorly maintained code can result into major failures and loss of money. Larger systems are usually developed by large groups of developers, making it difficult to maintain consistent naming conventions. In order to avoid these problems, development standards are put should be put into place to help prevent inconsistency.

The most important aspect that may be overlooked in a system is code testing. Code testing can quickly identify if changes to the system can cause errors. To safeguard changes into the system, companies implement testing frameworks that can quickly identify failures. In addition to writing maintainable code, it is also important to write maintainable test cases. Ever-changing code means ever-changing test cases. Following the same standards as code conventions, test cases should also be easily understood and adaptable.

Clean code and maintainable code go hand in hand, as developing code makes it easier for the developer to perform maintenance. The two categories described above define the major instances of maintenance systems can have, and can simply defines how well a system is maintained based on what type of maintenance a system undertakes.

WRITING CODE THAT IS REUSABLE

Code reusability is an essential component in software development. It allows for teams of developers to work together on highly complex applications while keeping their code organized. Code encapsulation is a process that allows developers to bundle functionality into related classes, while at the same time hiding other functionality that should not be visible to outside tasks. This is similar to reusable code because code should be stored in a location where it is accessible anywhere in the application. However, if the application is too complex, the code should be further sub-divided to ensure code is not repeated in multiple classes. Code that is repeated in multiple places in the application defeats the purpose of reusability. In addition to adding unnecessary code, it also makes tasks such as maintaining code more time consuming and expensive. Reusable code also makes the application extensible, as the code is well organized and easy to maintain. The reusability of code can simplify the structure of code, and is just one more step used by developers to create clean code.

Separating business logic from generic utility classes, which only contain tasks that are required by the system, is a fundamental step for creating reusable code. Although separating the business logic from the utility classes is important, it is also vital to understand the code. If a number of tasks are dependent on a section of reusable code (Ausnit-Hood et al. 1995), the code must be easily understood and changed accordingly. In addition, the code must also be independent so that unnecessary major refactoring is avoided. To avoid major refactoring developers must construct code to only adopt necessary information from other parts of the application that use it. Otherwise, the complexity of the code will make it difficult to be reusable as well as maintainable. Complexity in reusable code can result in fatal errors that can have far reaching consequences, therefore, code that will be reused must be of the highest possible quality.

CODE REFACTORING

Refactoring is prevalent in many aspects of software development. It occurs during application development, extension and maintenance. However, time constraints and code complexity often present barriers to refactoring.

There has to be a comprehensive understanding of the design, such that introducing new errors would be minimized. However, every software project has time constraints to which developers must abide by, so refactoring may not be possible. A project is typically paid for adding new features, not fixing something that is not broken. Consequences of not refactoring include the degradation of the design of the software. Code becomes more brittle and changes to the system are more expensive and frequent. In this excerpt, we will go over strategies and tools involved in code refactoring with examples to help visualise the concepts.

CODE REFACTORING STRATEGIES

Code refactoring is meant to preserve the behaviour of the system. Refactoring strategies following an arithmetic learning style and expect unit tests to occur after major/minor refactoring. This section will go over bad coding and refactoring strategies that can be used to alleviate the bad code.

With duplicate code, any changes made to one set must be made on all duplicates and the developer maintaining this duplicate code must be aware of all occurrences of it. To fix duplicate code, you can either push identical methods or the more general method up to a common superclass, or put the method into a common component that can be accessed by all who require the code. This also reduces the size of larger methods/classes as the code in larger methods and classes can be parsed into smaller methods and sub-components. Thus, these components and methods can be used more generally than their larger affiliates.

There are issues with instance variables where some classes would never use them and on the opposite end, co-occurring parameters are commonly seen. A set of co-occurring parameters would be something like (x-coordinate, y-coordinate). To handle instance variables, subclasses can be created that entail the instance variables that cover a high percentage of methods that would need them. To fix co-occurring parameters, it is basically the same approach where the creation of the class is specifically meant to entail those parameters (Point). Thus, the classes can handle any behaviour for the set.

APPROACHES TO REFACTORING

EXTEND - REFACTOR

Extend – Refactor is finding a similar class/method and making it work for the extension. Once the class/method is working, eliminate any redundancies that occurred during the implementation of this new method (Wake, 2013).

REFACTOR - EXTEND

Refactor-Extend considers the design of the system with the new extension. If it feels that this extension is not suited to fit well in the design, then there must be a refactor process to make it so the extension would be implemented easily (Wake, 2013).

DEBUG - REFACTOR

Debug - Refactor is meant to handle a bug, thus the behaviour of the system would change. Following debugging, refactoring would then proceed to make future bugs more obvious by extracting large methods, assigning names that describe the variable and getting rid of magic numbers and expressions (Wake, 2013).

REFACTOR - DEBUG

Refactoring first should preserve the behaviour of the system, so in this case it would preserve bad behaviour. It would also simplify complicated methods if done right, which makes it easier to debug the program (Wake, 2013).

REFACTORING TO UNDERSTAND

Refactoring in this case is to make the code readable to many developers rather than just the author. Performance and extensibility is also disregarded somewhat with this strategy. A big issue with refactoring to understand is that naming conventions or logical operations may only be obvious to the author. Common tasks involved with refactoring to understand are breaking apart large methods, removing magic numbers/expressions and descriptive variable/method/class names (Wake, 2013).

As there are no extensions being implemented or bug fixed using this refactoring strategy, it is not usually obvious to people outside the development team why there is time allocated to this process (Wake, 2013).

CODE REFACTORING EXAMPLE

A sample code refactoring will now be done on the following piece of code. We will be excluding unit tests from this example to keep it short, but it is imperative that unit tests are done and successful after each refactoring (Yoder, 2002):

```
public class CodeReplacer {
   public final String TEMPLATE_DIR = "templatedir";
   String sourceTemplate;
   String code;
   String altcode;
    //This method was created in VisualAge.
    // @param reqId java.lang.String
    // @param oStream java.io.OutputStream
    // @exception java.io.IOException The exception description.
   public void substitute(String reqId, PrintWriter out) throws IOException
   {
          // Read in the template file
      String templateDir = System.getProperty(TEMPLATE_DIR, "");
      StringBuffer sb = new StringBuffer("");
      try {
         FileReader fr = new FileReader(templateDir + "template.html");
         BufferedReader br = new BufferedReader(fr);
         String line;
         while(((line=br.readLine())!="")&&line!=null) sb = new StringBuffer(sb + line + "\n");
         br.close();
         fr.close();
      } catch (Exception e) {
      }
      sourceTemplate = new String(sb);
      try {
         String template = new String(sourceTemplate);
            // Substitute for %CODE%
         int templateSplitBegin = template.indexOf("%CODE%");
         int templateSplitEnd = templateSplitBegin + 6;
         String templatePartOne = new String(template.substring(0, templateSplitBegin));
         String templatePartTwo = new String(template.substring(templateSplitEnd, template.length()));
         code = new String(reqId);
         template = new String(templatePartOne + code + templatePartTwo);
            // Substitute for %ALTCODE%
         templateSplitBegin = template.indexOf("%ALTCODE%");
         templateSplitEnd = templateSplitBegin + 9;
         templatePartOne = new String(template.substring(0, templateSplitBegin));
         templatePartTwo = new String(template.substring(templateSplitEnd, template.length()));
         altcode = code.substring(0,5) + "-" + code.substring(5,8);
         out.print(templatePartOne + altcode + templatePartTwo);
      } catch (Exception e) {
         System.out.println("Error in substitute()");
      }
      out.flush();
      out.close();
   }
}
```

The code shown currently is working code that generates a web page by substituting strings in a template read from a file. Right now it shows that there are too many strings created as temporaries.

Now that we have code that is in need of refactoring, we must identify the smells of this piece of code. The most noticeable is the length of the substitute() method, so using Extract Method (Note, capitalization of the method is referencing the refactoring.com catalog) to break the large substitute() into smaller methods. The most obvious breaks would be where the comments are in the middle of the code. Here is the new readTemplate(), substituteForCode and substituteForAltcode() (Yoder, 2002):

```
String readTemplate() {
   String templateDir = System.getProperty(TEMPLATE_DIR, "");
   StringBuffer sb = new StringBuffer("");
   try {
      FileReader fr = new FileReader(templateDir + "template.html");
      BufferedReader br = new BufferedReader(fr);
      String line;
      while(((line=br.readLine())!="")&&line!=null) sb = new StringBuffer(sb + line + "\n");
      br.close();
      fr.close();
   } catch (Exception e) {
   }
   sourceTemplate = new String(sb);
   return sourceTemplate;
}

String substituteForCode(String template, String reqId) {
      int templateSplitBegin = template.indexOf("%CODE%");
      int templateSplitEnd = templateSplitBegin + 6;
      String templatePartOne = new String(template.substring(0, templateSplitBegin));
      String templatePartTwo = new String(template.substring(templateSplitEnd, template.length()));
      code = new String(reqId);
      template = new String(templatePartOne + code + templatePartTwo);
      return template;
   }

void substituteForAltcode(String template, String code, PrintWriter out) {
      String pattern = "%ALTCODE%";
      int templateSplitBegin = template.indexOf(pattern);
      int templateSplitEnd = templateSplitBegin + pattern.length();
      String templatePartOne = template.substring(0, templateSplitBegin);
      String templatePartTwo = template.substring(templateSplitEnd, template.length());
      altcode = code.substring(0,5) + "-" + code.substring(5,8);
      out.print(templatePartOne + altcode + templatePartTwo);
   }
```

Now that these methods have been added, the substitute() method is much smaller than the initial code. There is still refactoring needed to be done on the smaller methods though. In readTemplate(), Introduce Explaining Variable will be used to replace "templateDir" with:

```
String templateDir = System.getProperty(TEMPLATE_DIR, "") + "template.html";
```

Also, using Inline Temp, we realise that "fr" is not needed:

```
BufferedReader br = null;
    …
        br = new BufferedReader(newFileReader(templateName));
```

We need to surround the function with a try/catch block to properly close the stream in case of any errors:

```
try {
    …
  } catch (Exception ex) {
  } finally {
    if (br != null) try {br.close();} catch (IOException ioe_ignored) {}
  }
```

Inside the while loop, there is a constant initialization of a string that is not needed (Replace String With StringBuffer):

```
    sb.append(line);
    sb.append('\n');
```

Once all these refactorings have been completed, we can move on to substituteForAltcode() and substituteForCode(). These two functions have the same coding logic with two different parameters. It would be simple to add two parameters to a method and name it substituteCode()(Yoder, 2002):

```
        void substituteCode (
            String template, String pattern, String replacement, Writer out)
                    throws IOException {
              int templateSplitBegin = template.indexOf(pattern);
              int templateSplitEnd = templateSplitBegin + pattern.length();
              out.write(template.substring(0, templateSplitBegin));
              out.write(replacement);
              out.write(template.substring(templateSplitEnd, template.length()));
              out.flush();
            }
```

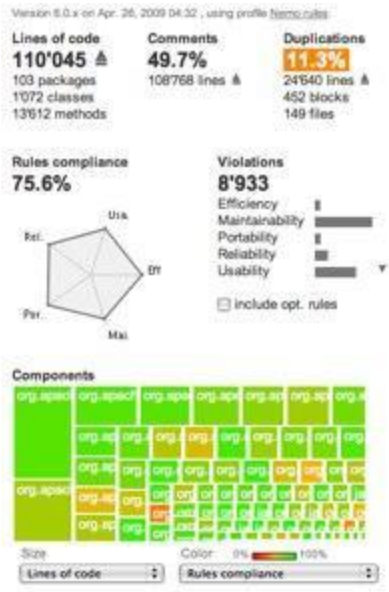
With all these changes, you would then handle errors and refactor tests appropriately to come to the final result (Yoder, 2002):

```
import java.io.*;
import java.util.*;

/* Replace %CODE% with requested id, and %ALTCODE% with "dashed" version of id. */

public class CodeReplacer {
  String sourceTemplate;

  public CodeReplacer(Reader reader) throws IOException {
    sourceTemplate = readTemplate(reader);
  }

  String readTemplate(Reader reader) throws IOException {
    BufferedReader br = new BufferedReader(reader);
    StringBuffer sb = new StringBuffer();
    try {
      String line = br.readLine();
      while (line!=null) {
        sb.append(line);
        sb.append("\n");
        line = br.readLine();
      }
    } finally {
      try {if (br != null) br.close();} catch (IOException ioe_ignored) {}
    }
    return sb.toString();
  }

  void substituteCode (
   String template, String pattern, String replacement, Writer out)
              throws IOException {
    int templateSplitBegin = template.indexOf(pattern);
    int templateSplitEnd = templateSplitBegin + pattern.length();
    out.write(template.substring(0, templateSplitBegin));
    out.write(replacement);
    out.write(template.substring(templateSplitEnd, template.length()));
    out.flush();
  }

  public void substitute(String reqId, PrintWriter out) throws IOException {
    StringWriter templateOut = new StringWriter();
    substituteCode(sourceTemplate, "%CODE%", reqId, templateOut);

    String altId = reqId.substring(0,5) + "-" + reqId.substring(5,8);
    substituteCode(templateOut.toString(), "%ALTCODE%", altId, out);

    out.close();
  }
}
```

With several code refactoring practices and strategies to follow, one can be overwhelmed with the process until they reach a certain level of experience. Is there anything available to help the average programmer refactor and also prevent the production of bad code? The answer is yes. There are several source code analyzers out there if you know how to look, including Sonar - one of the most widely used and most supported open source platforms for managing code quality.

Sonar has the ability to expel multiple statistics on source code projects, and focuses on covering their "7 axes of code quality":

- Duplicated code
- Coding standards
- Unit tests
- Complex code
- Potential bugs
- Comments
- Design and architecture



(WIKIPEDIA, 2009)

Sonar has over 35 plugins, and supports over 20 different programming languages. It will scan code for the elements listed above, including the cyclic code complexity (a measurement number related to the possible paths a program can take) and store this information into a database to relay back to the user via charts and fields. It should be understood that the information given should be taken with a grain of salt. A user may wish to create a profile to tell Sonar what it should actually be looking out for, instead of pointing out things that the user intended to be clean.

Another useful tool that is worth mentioning is JustCode by Telerik. This is actually a Visual Studio extension that works seamlessly while you code to help prevent problems from ever occurring. It includes several tools such as code generation, navigation, structure and layout schemes, templates, and refactoring tools. While this tool is not free, it could prove to be one of the best tools for coding and learning how to code properly in Visual Studio.

CONCLUSION

The negative effects of bad code should be apparent at this point, it can have major consequences down the road when another developer, or even yourself, has to fix a bug or implement something new.

"A well-written program is a program where the cost of implementing a feature is constant throughout the program's lifetime." (Maman, 2008)

Learning how to code and refactor "for the future" won't happen overnight, but it is important that developers are aware of the fundamentals so that they can gain experience and form good habits throughout their careers. Of course, there will always be criticism towards different people's code, what may seem like easily extensible code to one person, may not to another. However, with a general foundation for code quality, one can have a huge impact on the volatility of a software project, as the iterations pass by and new features and bugs arise, regardless of what other opinions a neighboring developer may have. Co-workers will also appreciate and respect you if they can easily read and grasp the purpose of your code, and less stress is a good thing.

REFERENCES

1. Atwood, Jeff. "Code Smells" . *http://www.codinghorror.com/blog/2006/05/code-smells.html*, n.p., 18 May 2006. Web. 28 Feb. 2013.
2. Ausnit-Hood, Christine, Mr. Kent A. Johnson, Mr. Robert G. Pettit, and Mr. Steven B. Opdahl, "Chapter 8 - Reusability - TOC". http://www.adaic.org/resources/add_content/docs/95style/html/sec_8/, n.p. October, 1995. Web. 27 Feb. 2013.
3. Blanchard, Benjamin and Verma, Dinesh and Peterson, Elmer. "*Maintainability, A Key Effective Servicability and Maintenance Management*", John Wiley & Sons. (1995). pg 15-20.
4. *Google Blog* "Writing Testable Code" . http://googletesting.blogspot.ca/2008/08/by-miko-hevery-so-you-decided-to.html Google 6 August 2008 Web. 27 Feb. 2013
5. Mallet. Freddy and Gaudin, Olivier. "SonarSource". http://www.methodsandtools.com/tools/tools.php?sonar n.p. Web, 27 Feb. 2013.
6. Maman, Itay. "Finally Definition for Good Program". Javadots. http://javadots.blogspot.ca/2008/07/finally-definition-for-good-program.html. n.p., 8 July, 2008. Web, 26 Feb. 2013.
7. Martin, Robert C. *Clean Code A Handbook of Agile Software Craftsmanship*. Upper Saddle River, New Jersey: Prentice Hall, 2008.
8. McConnell, Steve. *Code Complete.* Redmond, Washington: Microsoft Press, 2004. Print.
9. Miller, Jeremy. "On Writing Maintainable Code" . http://codebetter.com/jeremymiller/2006/12/06/on-writing-maintainable-code, n.p., 6 Dec 2006. Web. 28 Feb. 2013.
10. No Author, "File:Sonar-snapshot.jpg" http://en.wikipedia.org/wiki/File:Sonar-snapshot.jpg April 2009. Web, 11 April, 2013.
11. No Author, "Telerik Product Page". http://www.telerik.com/products/justcode.aspx. Web, 28 Feb. 2013.
12. Oracle "Indentation " . http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-136091.html#248: Sun, n.d. 16 July 2010 Web. 27 Feb 2013
13. Princeton University. "Writing Clear Code". http://introcs.cs.princeton.edu/java/11style/ Princeton n.d. Web. 27 Feb. 2013
14. Wake, William C. "Refactoring: an Example". http://www.xp123.com/xplor/xp0002b/index.shtml, n.p., February 2000. Web. 28 Feb. 2013.
15. Yoder, Joseph and Roberts, Don. "Refactoring Tactics and Strategies", The Refactory, Inc. (2002).