Clean Code Craftsmanship

1. Symptoms of Poor Application Code at PayPal

2. Introduction

Even bad code can function. But if code isn't clean, it can bring a development organization to its knees. Every year, countless hours and significant resources are lost because of poorly written code. But it doesn't have to be that way. A professional developer must have the proper skills to not only implement clean code, but also to remedy bad code.

Programming highlights our limitations when it comes to complexity along with out desire to work together to slve large problems in pieces. That's the engineering of programming. People make building blocks that others can understand and reuse easily.

We believe that the current code situation can be significantly improved by focusing on clean code agile coding practices. In particular, the combination of practices for TDD, UT, code reviews, other agile practices and clean code will greatly benefit our code base. PayPal is and will be investing a huge amount of money for the next generation of products. We must insist that the code be clean!

This series of blogs will provide a non-exclusive guideline in this endeavor.

Our goal as evangelical craftsman will mostly use the Code Review Champions as an implementation methodology. This does not, however, exclude individual developers from practicing clean code! On the contrary, we warmly encourage this education!!

Unfortunately we do not expect many PayPal developers to actually read most (or any) of this documentation. We must present this material through code reviews and pair programming. A developer has to experience clean code before becoming a convert.

2.1. Clean Code: A Definition

What exactly does "clean code" mean? And how does it apply to developers?

Some attributes:

- Easy to read and understand
- Easy to modify
- Easy to test
- Works correctly
- Comments are up-to-date and do not obscure the logic
- Code that reads as close as possible to human language
- Focused: Each class, method or other entity implements responsibility for a single aspect.
- Implements a good, clean solution without workarounds which make the code and language look awkward.
- Reading the code should be pleasant
- Easily extended by any other developer.
- Minimal dependencies.
- Smaller is better.
- Proper unit and acceptance tests
- Expressiveness. Names should be meaningful and express their intention.

So what? Clean Code obviously does not have an easy definition.

Chapter 1 of "Clean Code" by Robert C. Martin, "The Total Cost of Owning a Mess", p. 5, provides several definitions of Clean Code by prominent software leaders.

2.2. Craftsmanship

The act of writing clean code results in craftsmanship. Craftsmanship results from years of work and study and attention to details. A craftsman studies his craft because a passion drives the craftsman. No one has to persuade the craftsman that the study must be done - the craftsman knows it must be.

The software craftsman fully understands that the ills of the mainstream software industry arise due to a priority of financial concerns over developer accountability. The calls for an engineering approach that would require licensing, certifications and provable domains of knowledge result from these financial concerns. The Agile manifesto emphasizes, in part, "individual and interactions over processes and tools".

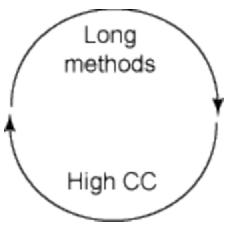
A metaphor may be drawn comparing the software apprenticeship to medieval apprenticeships: apprentice/intern, journeyman, master. Even in modern times doctors, plumbers, carpenters, electrician and many other trades practice this. Would you visit a doctor who was not a master at his trade? Then what is the purpose of placing the financial well-being of your company in someone who has not progressed significantly along the path to a master software craftsman?

Every time I try my hand at car repair or carpentry, I realize that I am not an expert at those things, and I am happy to do business with people who are. I am not an expert cook, but I am happy I live with one. Becoming a software expert is no different from becoming an expert cook or an expert car mechanic. It takes a lot of time and effort. It is easier to become an expert if you work with an expert, but you can usually learn on your own with a lot of practice and by reading books and articles. Even if you work with an expert, you have to practice a lot. You don't really learn how to do something by reading a book or listening to someone talk, you learn by doing. The book and the expert tell you what to practice, let you know what to aim for and when you miss the mark, and give you information that you would otherwise have to learn on your own. But expertise is always expensive to acquire. While I feel I am trying, I also have a *long* way to go.

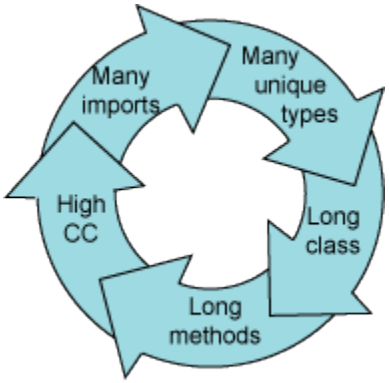
Static Analysis Tools

Static analysis tools such as pmd, javancss (java Non Commenting Source Statements) and jdepend provide reliable metrics as a general guide.

Study of the output of these tools reveals a pattern that emerges suggesting that gluttonous code (long methods, too many public methods, excessive conditionals and imports, etc.) impairs readability, testability, and maintainability. Because this pattern repeats itself in various metrics, they all tend to *correl ate*. For example, long methods generally suffer from high cyclomatic complexity values.



The correlation doesn't stop there, however. Classes with a plethora of imports have many unique types. These classes are typically pretty big. Big classes usually have long methods, and long methods often have high cyclomatic complexity values. These complexity metrics correlate:



Measuring Verbosity with JavaNCSS

As opposed to PMD, which defines specific rules for analyzing source code, JavaNCSS analyzes a code base and reports *everything*relating to the code length, including class sizes, method sizes, and the number of methods found in a class. With JavaNCSS, thresholds don't matter -- it counts every file it finds and reports the values *regardless* of size. While this kind of data may seem pedestrian (and perhaps verbose!) compared to PMD, it couldn't be farther from the truth.

By reporting all file sizes, JavaNCSS makes it possible to understand relative values, which is often difficult with PMD. For example, PMD only reports files that contain violations, which means only understanding the data for a portion of the code base, while JavaNCSS provides code-length data in context.

2.3. Beyond Static Analysis Tools

Clean Code goes beyond static analysis tools. Static analysis tools cannot:

- Evaluate the appropriate use of a method name to its function.
- Determine if refactoring is necessary other than complexity.
- Sense readability issues.
- Detect anti-patterns.
- Decide if polymorphism could replace that switch statement

Clean code has no conflict with these tools. Clean code simply takes the next logical step by allowing the artistry of the developer to emerge and create a truly fine product. These static analysis tools are certainly necessary and must continue to run. These static analysis tools definitely assist developers by clearly detecting overlooked items.

3. The Path to Software Craftsmanship

Software development is not for everyone. A craftsman derives motivation from a burning passion and curiosity. A craftsman continues to investigate and experiment with development not only in the work context, but also as a play-time exercise.

"Play-time"? Yes. By exploring and experimenting with software of all kinds outside of work, a much broader view of the world becomes possible.

3.1. But I'm not a craftsman, yet!

Nobody starts out as a craftsman. A software developer begin a basic software education. Given time, mentoring, role models and study, the craftsmanship grows and grows. That passion that selected software as a career path will guides you towards this goal.

Keep at that goal and, bit by bit, you will get better.

There is a picture of Albert Einstein rocking his child while working on a problem. That's passion. Each bit of learning contributes to your overall craftsmanship. And just like compound interest, the greater your devotion the great your rewards. As the higher your interest level, the greater the payback.

You know what you need to do. Let me help with a list:

1. Start participating in various coding exercise sites. Google "java coding exercises" and similar phrases. Many sites are not java specfiic, but have solutions many languages.
2. Learn a new language. Other languages have fantastic offerings. That new language will provide a new perspective on your java coding. Extra credit: learn a language as different from you corporate language as possible - like lisp, erlang, haskell, ...
3. Read books. This includes blogs on the various topics of your choice related to your current studies.
4. Consider certification. While I am not a great fan of certification, this will solidify you certainty that you have learned something.
5. Attend meetings, conferences, talks, etc. The social aspect can easily create enthusiasm for your quest.
6. Review the basics. What did you take in school that you did not master? Go back and really learn.

While you are on the path, it's important to be *actively* engaged. Don't just copy/paste and example and run it. Actually type that code into a file. Making typos is a great way to learn how to recover from real-life problems. And with each line you enter, you are analyzing what is happening.

If what I have said about does not resonate internally, then perhaps software development is not for you. Perhaps you really need to apply yourself to your passion in life instead of plodding along waiting for that paycheck. Following your passion will definitely improve the quality of your life, your family and those around you. Continuing to insist on a life course not to your liking creates an increasing amount of psychic dissonance significantly contributing to an unhappy, frustrating life.

3.2. Learn Anti-Patterns

The corporate environment, unfortunately, contains countless examples of software anti-patterns.

The point of anti-patterns, and even patterns, is not to describe tricks any of us have invented independently, but solutions and observations *many* of us have come up with, independently, to solve and observe a wide range of problems.

3.3. Master the Basics

While this blog specifically targets Java (with some C++ thrown in), mapping the suggestions to any computer language is strongly encouraged.

Begin by working through "Thinking in Java" by Bruce Eckel and continue with "Effective Java" by Josh Bloch. Any working Java developer should be intimately with almost everything in these volumes.

4. Our Foundations: "Effective Java" and "Clean Code"

We base our efforts on two remarkable books: "Effective Java" by Joshua Block and "Clean Code" by Robert Martin.

For C++ we use "Effective C++ by Scott Meyers as a replacement for "Effective Java". "Clean Code" contains enough portable ideas that can easily be implemented in C++.

From these foundations a wealth of knowledge and wisdom is present for the taking. To properly digest and master the ideas in these volumes could take years. But we don't have years - we have *now*. To begin, start examining your code for the critical potential problems listed below. The suggested solutions are generally too voluminous for this document to easily cover. Beside, other authors have well documented solutions.

Is this worth the effort? Is your passion worth the effort?

With the PayPal switch to Java we have a responsibility to deliver solid, clean, tested and maintainable code. PayPal invests heavily in us, should we not return the favor with a quality product?

4.1. The 7 +/- 2 Principle

Our limit on mental capacity to processing information has been proven to be 7 +/- 2. Stated another way, the number of objects an average human can hold in working memory is 7 +/- 2. This is referred to as Miller's Law.

Miller's Law has implications for writing, reading and maintaining software. A developer must hold multiple concepts in memory when performing any sort of programming. The more a developer must cram into that cranium, the less likely significant code will be developed.

4.1.1. It's Really 3 or 4 for the Long Haul!

The 7 +/- 2 principle does not account for developers working hard all day long! Coding is an intensive activity! Fatigue sets in, interruptions break our flow. Emails and noise and skype distract. Lower expectations and assume that only 3 or 4 concepts may be kept in mind at once. Having only 3 or 4 concepts will certainly result in better quality code because the mind will not likely become overwhelmed.

This 3 or 4 concept limit implies that the admonitions to learn your editor, programming language, libraries significantly enhance your productivity! Distracted thinking on how to perform these basic tasks will cause concepts to drop our of your brain cache.

Many more Clean Code concepts apply the 3 or 4 concept limit. A poorly named method overloads the brain cache or the real intent becomes fuzzy. A complicated conditional forces extra parse cycles that cleaner coding would relieve. Poor comments mislead by poisoning the brain cache. Deeply nested if...then... structures forces one to remember how the logic applies. Poor indentation and lengthy lines force the mind with a bit of an extra effort. Duplicated code requires one to examine each instance to determine if the are truly duplicated or copy/paste/hacked code. How long can this list continue?

4.2. Your Editor

Learn your tools! Do you use eclipse? vim? emacs? linux? windows? Whatever tool you use, learn it well enough that operating it becomes unconscious. You do not have to think about that tool, you just use it. If you struggle with the tool, you will push other concepts from your memory.

4.3. Your Programming Languages

Learn your language! Java, C++, shell, etc. Learn that language well enough that you don't have to think about it.

4.4. Basic Data Structures

Learn the basic data structures. Working with a Map, LinkedList, etc. should be second nature. Your fingers should know the commonly used methods. If it's not, start writing practice code with these structures. Understand the situations of superiority of one data structure over another.

4.5. Immutable whenever possible.

Use "final".. This allows a developer to scan to the initial setting of a "variable" and know that it has not changed. At least one item has been removed from short-term memory. Think immutable states.

In functional programming a "variable" is *always* final: it gets set once and remains immutable. This allows a developer to scan to the initial setting of a "variable" and know that it has not changed. A least one item has been removed from short-term memory. Think immutable states.

4.6. A Concept Per Day

Take just one concept from our suggested learnings per day and investigate its consequences.

4.7. Small is beautiful for many reasons.

Short methods! A small function is usually easy to understand. And it is certainly less to hold in memory. A properly named short function increases cognition much more than an inline series of statements that perform the same as that short function.

4.8. Functions with Few Arguments.

None is preferable, but definitely three or less. With three arguments your 7 +/- 2 is getting full.

4.9. Software Patterns

Our code is full of software patterns even if they are not recognized. By learning patterns and when to apply them, yet another burden is alleviated from our 7 +/- 2 memory cache. Recognize a pattern, verify the correctness and you have just abstracted a useful chunk of code. A pattern is much easier to recognized and remember than a series of code.

4.10. Don't repeat yourself!

Duplicated code is not only poor programming, but it forces you to review each repeat to ensure that they are indeed performing the same task. And consider the maintenance when an error or upgrade impacts one of the duplicates! Then two almost duplicates exist and you overload your cognitive abilities yet again.

4.11. No Side Effects

Ideally calls have no side effects. A method changing the foundation under you requires even more understanding. Some mutability is unavoidable such as IO or DB access or ...

4.12. The Billion Dollar Mistake: Null References

Nulls by definition are "out of bounds". A null thus requires yet another item in our short term memory. By losing a null from our memory cache, we can easily make serious mistakes and not even know it. The consequences are difficult to predict in the general case. This may range from a NullPointerException to totally ignoring areas that really need attention. Do you *always* handle *all* nulls correctly?

Tony Hoare on his invention of null: "I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years..."

How many instances of null are causing problems in our code? For an approach see the classic text "Thinking in Java", p. 598 by Bruce Eckel.

For a different approach see "Functional Programming in Java", p. 17: "What About Nulls?" for a suite of classes that support much safer null handling.

A criticism of the Null Object Pattern cautions that it can make errors/bugs appear as normal program execution.

For a more detailed discussion with examples see Clean Code: Null Objects as a solution.

5. Implementing Clean Code at PayPal

The processes of Agile will certainly help. We must make a deeper change in the *people* at PayPal. Code Reviews conducted by a trained team should assist in educating developers. "Normal" training classes will not: students will surf the web, ...

> - *"The major problems of our work are not so much technological as sociological in nature."* - Tom De Marco, PeopleWare

The Clean Code initiative is not a goal in and of itself, but a methodology to implement business goals in a timely manner with as few customer disruptions as possible. This specifically means minimizing live bugs, responding to business, help desk, and sales request in a timely manner. In other words, to truly live the stated goals of our yearly goals.

5.1. Engaged Code Reviews

Code reviews can be the easiest way to indoctrinate developers into decent practices. The Code Review Champions was formed expressly for this purpose.

By illustrating clean code changes on code that a dev has familiarity, hopefully some of our ranting and preaching will sink in.

By "engaged" I mean developers actively participating in the code review process instead of tuning out as soon as the light go off. By inviting a Code Review Champion who points out various potential problems, areas of improvements, asks questions, etc., engaged developers can think more concisely about their own code.

The very act of knowing that your code will be reviewed should motivate most developers to write better code. For myself, since I have started this series of documentation, I know my code writing abilities are better.

5.2. Improve English Understanding

A properly named class, method or variable greatly assists in understanding. "Clean Code" devotes an entire chapter to this problem. Developers should not have to translate code names into something else that makes sense to them. A poor name impinges upon the 7 +/- 2 rule. Mentally juggling misnamed code names only increases chances for mistakes. The semantic distance between a method name and body should be minimal. A well named method increases the chances that it may be reused. Additionally, a well named method requires fewer comments because the name itself becomes a comment. When these names get used by other methods, the well named methods read more like comments.

Overly long and complicated logic only increases the meaningful name problem. A function named "do_business_logic" conveys little insight into the actual functional logic.

Given that perhaps 70% of PayPal developers use English as a second language presents challenges. Several non-solutions present themselves:

1. Provide English classes.
2. Stress the importance of meaningful names.
3. Setup email/phone calls to Tech Writers for assistance.
4. Write a utility to extract all class, method and variable names as part of the code review process.

A draconian approach will not work for language acquisition. An individual can easily take 10 years to master English well enough to create meaningful names.

"The Elements of Style" by William Strunt is highly recommended. This describe the style of writing, similar to our style of coding. This book is short. From the introduction “This book aims to give in brief space the principal requirements of plain English style. It aims to lighten the task of instructor and student by concentrating attention (in Chapters II and III) on a few essentials, the rules of usage and principles of composition most commonly violated.”

Comments should be the last item added to a class or method. Comment methods only after doing everything possible to make the method not need comments. Clarify the code instead of adding explanations. The code itself should explain itself if it were done better.

TBD: Is there hope for meaningful names?

5.3. Test Driven Development

Absolutely *everyone* would agree that TDD should reign supreme. With PayPal's new code in Java, there is no excuse for not using TDD. We've got to get smarter in creating unit tests and TDD.

Our mentors in "Clean Code", "Working Effectively with Legacy Code", and man other Agile experts agree that:

> *"The discipline of Test Driven Development has made a profound impact upon our industry and has become one of the most fundamental disciplines."*

TDD is not just a means of ensuring that the code is correct. TDD provides a technique for consistently paced development. Code gets developed incrementally and provides feedback in minutes or even seconds that the current direction is correct. TDD is about confidence.

TDD affects the design of the your system. This is deliberate: TDD guides you into systems that can be easily tested. Systems can be tested in isolation and this leads to a system where objects are not tightly coupled to one another. This decoupling is a primary indicator of a well-designed object oriented system.

TDD provides live and up-to-date documentation of the functionality of the classes. By reviewing the tests, the class can be understood. By reading the names of tests to understand the functionality that a class supports. Each test should be readable as documentation on how to use that functionality. A code test provides a comprehensive specification that others can understand.

Complex code invariably contains holes of defects.The key to success in TDD is to understand its emphasis on early feedback. A defect indicates the unit tests were incomplete. Write that unit test, ensure that it fails, then fix it.

As you write a test, you might consider the test sufficient and choose to move on. In doing so you might not be out of line. However, the less confident you are and the more complex the code, the more tests you should write.

Developers typically resist testing for various reasons:

- They believe they do not need tests, since they write good code.
- They are not motivated by long-term goals since they believe they may leave the project in a short time.
- They believe that the code is so bad they will have a job for a long time in maintaining code. If the code is working and is good, they will lose their job.
- They are more interested in tools and processes that reduce the time they are losing in identifying problems.
- Fixing bugs is not fun. Testing for bugs is even less fun.
- Writing tests is not considered to be a noble task.
- They have poor coding skills, but good political connections. Thus, any tests may expose their skill level.
- Managers and customers are not interested in paying for test but for new features in the system.

5.3.1. A Business Case for TDD

We need to build a business case for TDD. This must demonstrate that tests will not only speed up today's development, but they will speed up future maintenance efforts.

Every time a test gets written, either after a bug or adding a new feature or refactoring, a *reproducible* and *verifiable* piece of information about the system has been added. That test is *verifiable* because it can be run at any time to verify correctness and consistency.

Changing legacy systems is risky. Will the code work after a change? How many unexpected side effects will appear? Can you detect these side effects? Having a running set of tests always in sync with the target system helps in minimizing this risk.

And, perhaps most importantly, really solid testing reveals bugs much earlier in the development process. This is a *big* win in terms of productivity and costs.

TDB: Need a solid business case at PayPal!

5.3.2. When NOT to use TDD

Sometimes TDD is not the appropriate tool. These include trivial getters and setters,. More trivial routines that only emit a few logs, call a service and log return values. These routines could be part of TDD by creating a fake that returns fixed values and ignoring the trivial logic.

Developing UIs can often lead to a number of trivial routines that perform almost no logic. Some database routines have similar routines that may be ignored and replaced by a fake.

5.4. Pair Programming

A Clean Code Team consisting of passionate developers assists others with pair programming. Pair programming provides a remarkably good way to increase quality and spread knowledge around developers. It's remarkably easy to create a problem and not even know it and that second set of eyes helps. Bad code is surgery, and doctors never work alone.

The major down side of pair programming is that is absorbs a developer who is familiar with Clean Code with someone less familiar. Unfamiliarity with the domain will likely hinder development. And, most importantly, the Clean Code Mentor will be engaged on another domain rather than performing assigned work on their own domain.

Pair programming *must* assume an actively engaged pair. Each person must remain alert instead of blindly following whatever the other person decides.

5.5. Best Practices Sessions

Some recurring best practices sessions may be helpful. Similar style best practices are emailed weekly. We could do the same. (Do people read these emails?)

5.6. TODO

1. Need a **cartoonist**. A developer is in the center.All around are symbols of the various impediments assaulting and hindering work: time pressures, QA,unit tests, RQA, FQA, pushes, tickets, meetings, managers, scrum, pressure, estimates, documentation, overflowing email, deadlines, demos, assisting others, group activities. The central idea should be that a developer must overcome great obstacles to actually perform work. Managers should definitely read Peopleware to ease this difficulties of development.

6. Example Problems and Solutions

6.1. Java equals() - An Alarming Problem!

In Item 8 of *Effective Java*, Josh Bloch describes the difficulty of preserving the `equals` contract when subclassing as a “fundamental problem of equivalence relations in object-oriented languages.”

The authors of " How to Write an Equality Method in Java" base their work upon a 2007 paper that concludes "...*almost all implementations of `equals` methods are faulty"* (emphasis added).

The difficulty arises from preserving the equals() contract when sub-classing as a fundamental problem in object-oriented languages.

In our Java implementations how may equals() methods have we written? How many have been reviews for any sense of "correctness"? How many uncaught problems has this caused?

Since hashCode() is the right hand of equals(), do not ignore this supporting method.

6.2. Globals in Java: Statics & the Singleton Pattern

Many different kinds of dependencies can make testing difficult to create and test, but one of the hardest is the global dependency. And this frequently manifests itself with the Singleton pattern.

See the reference "Working Effectively with Legacy Code", p. 118 for a detailed example.

6.3. Cannot Test Method

Sometimes getting tests in place can be tricky. Instantiating the test is only part of the battle. Then write tests for the code that has changed. In most cases we have problems similar to:

1. The method might not be accessible to test. It could be private or have other accessibility problems.
2. It might be hard to call because it is hard to construct parameters we need to call it.
3. It might have bad side effects such as modifying a database.
4. It might use other objects hard to construct.

6.4. Nested ifs: The Arrow Anti-Pattern

Have you ever encountered code that looked like:

```
if
   if
     if
       if
         do something
       endif
     endif
   endif
 endif
```

It often develops when a programmer applies the "OneReturnPerFunction rule" blindly and in poor taste, or when mixing both conditions and loops.

See: Nested Ifs: The Arrow Anti Pattern for discussion and approaches to solutions.

6.5. Refactoring

Please refer to Clean Code: Refactoring.

Refactoring is the workhorse of clean code. Refactoring is a process of changing a software system in such a way that it does not alter the external behavior, yet it improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. Each step is simple. Refactoring properly requires many simple steps. Collect a few statements into a new method. Move a field from on class to another. Rename methods and variables. Test and return early if a parameter fails a test. Yet the cumulative impact of the small changes can radically improve design and enhance maintainability. This is the exact reversal of software decay.

Magic Numbers

"Magic Numbers" are hardcoded numbers or character strings that should belong in a common constants module. These have unique values with unexplained meanings. Use of magic numbers violates one of the oldest rule sof programming as it obscures the developer's intent in choosing that number. It increases the opportunity for subtle errors, decreases readability and make if more difficult to adapt and extend in the future.

Use of a constants module offers multiple advantages:

1. Readability. A programmer reading a char value like 'I' might wonder, "What does "I" represent? And how does it impact logic here?" While inferring the meaning from code is certainly possible, it may not be obvious to all but the initiated. Magic numbers become particularity confusing when the same numbers get used for different purposes in one section of the code.
2. Alter the value as it is not duplicated. Changing the meaning of a number number will be very error-prone since a developer must locate all instances of its use. When the magic number is in multiple modules and repos, the situation can be grave. By contrast, changing a static final in a single file will guarantee all instances are updated.
3. The declaration of magic numbers are declared adjacent to each other. This facilitates understanding as similar groupings of values may convey similar usage. For example, the single letter character used to denote the different brands of credit cards.
4. It helps to detect typos. Using a misspelled static final constant name will produce a compiler error while a typos in the literal can cause a perplexing problem.

Finding Single Character Magic Numbers

A simple method to find all single magic characters in a repo:

```
find . -name '*.java' | xargs egrep -n "'[A-Z0-9a-z]'"  | grep -v final
```

The last fragment removes definitions embedded in the repo. A more common repo may be more appropriate for these constants.

Logging and Exception Handling Obscures Logic

TDB: Need a separate page.

7. Basic Principles

1. Don't Repeat Yourself. (DRY). Any duplication of code or even intermediate steps triggers mistakes and inconsistencies. Perhaps that initial duplication seemed necessary, but what happens when a bug surfaces in one and the other dup gets ignored?
2. Keep It Simple, Stupid (KISS). If you do more than the simplest, you make the solution unnecessary and make your customers wait.
3. Beware of Premature Optimizations! Get it *right*, then optimize. While our PayPal systems are too slow, let's ensure that they are correct.
4. Favor Composition Over Inheritance. Composition promotes loose coupling and testability of a system and is usually more flexible.
5. Follow the boy Scout Rule. Raise the bar just a little whenever a piece of work is touched. Get there completely free of bureaucratic planning and make this a grass-roots conspiracy!
6. Root Cause Analysis. Treating symptoms is only a fast relief. In the long run this is only more costly. Always look beneath the surface of problems.
7. Apply Simple Refactoring Patterns. Code is easier to read and improve when a developer can understand and test it.
8. Reflect Daily. No improvement, no progress, no learning without reflection. Do at least one Clean Code principle per day. Make this a habit.
9. Names must reveal intent. A name should reveal all the big questions: why it exists, what it does and how it is used. If a name requires a comment, the name failed.
10. Single Responsibility Principle. (SRP) A class or module should have only one reason to change.Identifying responsibility helps to recognizer and create better abstractions in code. SRP is an important concept in OO design and also one of the most abused design principles. Many classes do far too many things.
11. Test Driven Development. (TDD). Absolutely.
12. Dependency Inversion Principle. (DIP). High level modules should not depend on low-level modules. Both should use abstractions. Abstractions should not depend depend upon details. Details should depend upon abstractions.

8. Glossary

Clean Code: Software code that is formatted correctly and in an organized manner so that another coder can easily read or modify it.

Code refactoring is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior"

DRY: "Don't Repeat Yourself". The idea of refactoring and eliminating duplicated code is one of the most important principles in Clean Code.

Duplicate Code: See DRY.

JavaNCSS: Java Non Comment Source Statements. A static analysis tool for complexity.

Readable Code: Code that is easy to understand by simply reading. Many techniques provide this capability: methods grouped at the same level, conditionals easily understood, guard clauses indicate initial parameter checking and routine exits, proper method and variable naming, etc.

Refactor: See Code refactoring.

Single Responsibility Principle (SRP): States that every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

Test Driven Development: A software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

9. Links and References

9.1. Safari eBooks

START HERE => **"The Art of Readable Code"** by Dustin Boswell. This book is about how to write code that's highly readable. The key idea in this book is that **code should be *easy* to understand**. Specifically, your goal should be to minimize the time it takes someone else to understand your code. This book is intended to be a fun, casual read. We hope most readers will read the whole book in a week or two.

"**Effective Java**" by Josh Block. This book is *the* reference for students mastering java. After mastering the syntax of the java language, "Effective Java" provides guidance on how to effectively use java in the real world. It's too easy to write java that appears to work, but has fundamental flaws that create havoc. Available through our Safari docs.

"**Clean Code - A Handbook of Agile Software Carftsmanship**" by Robert Martin. Looking at code from every direction. Be able to tell good code from bad code. And, more importantly, how to transform bad code into good code. Available through our Safari docs.

"Working Effectively with Legacy Code" by Michael Feathers. In this book, Michael Feathers offers start-to-finish strategies for working more effectively with large, untested or poorly legacy code bases.

"Refactoring: Improving the Design of Existing Code" by Martin Fowler, et. al. This book offers a thorough discussion of the principles of refactoring, including where to spot opportunities for refactoring, and how to set up the required tests. There is also a catalog of more than 40 proven refactorings with details as to when and why to use the refactoring, step by step instructions for implementing it, and an example illustrating how it works.

Elemental Design Patterns (link has pdf of entire book) provides a fantastic description of an automatic system to detect and display patterns from code. From the introduction by Gary Booch: "Elemental Design Patterns will help you think about patterns in a new way, a way that will help you apply patterns to improve the software worlds that you create and evolve. If you are new to patterns, this is a great book to start your journey; if you are an old hand with patterns, then I expect you'll learn some new things. I certainly did."

9.2. External Links

Anti-Patterns (eBook) : An Anti Pattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.

Design Patterns Course - This tutorial includes two pdfs in the links and a **total of 97 video lessons** with an overall duration more than **7 hours**. It's impossible not to become an expert in design patterns after you watch them all. The video lessons are on the hypers at /x/home/cmcgregor/dp_all_in_one. zip . The original source is Source Making which contains an incredible repo of clean code ideas.

People Ware - If you think clean code is just about software, you're wrong! The ideas in People Ware applies to work groups of _people_ and the interface with software. People Ware contains interesting ideas about managers as well.

JASON MCCOLM SMITH's PhD thesis for Elemental Design Patterns: SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns from Source Code (pdf)

9.3. Null References

Maybe for Java - "The aim of the Maybe type is to avoid using 'null' references and, therefore, unchecked NullPointerExceptions. In a large project, null references are extremely problematic. Stray null references cause unchecked exceptions in code far away from the source of the reference. If programmers are never sure whether references are null or not, they bloat the code with null checks. And, if a reference is null, what should they do? They may throw an unchecked exception early, which is only slightly better than dereferencing a null pointer a bit later in the method. Or they may fall back to some default value, which is, in my experience, much worse because the actual runtime behaviour of the code becomes very difficult to understand."

Functional Java - "...an open source library that seeks to improve the experience of using the Java programming language in a production environment." The Option class provides a replacement for the use of null with better type checks.

More Links

- *In pursuit of code quality*: Read the complete series by Andrew Glover.
- "*In pursuit of code quality*: Refactoring with code metrics" (Andrew Glover, developerWorks, May 2006): Use the Extract Method pattern to refactor and simplify complex code.
- "*In pursuit of code quality*: Code quality for software architects" (Andrew Glover, developerWorks, April 2006): How afferent coupling and efferent coupling metrics reveal the stability of your software architecture.
- "*In pursuit of code quality*: Monitoring cyclomatic complexity" (Andrew Glover, developerWorks, March 2006): Learn the simple code metrics that let you spot and correct risky code.
- "Testing legacy code" (Elliotte Rusty Harold, developerWorks, April 2006): A step-by-step processing for grappling with the ancient beast.
- "Zap bugs with PMD" (Elliotte Rusty Harold, developerWorks, January 2005): More about PMD's built-in rules and custom rulesets.
- The Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

- PMD: Scans Java code for problems.
- JavaNCSS: A simple utility that measures source code metrics.