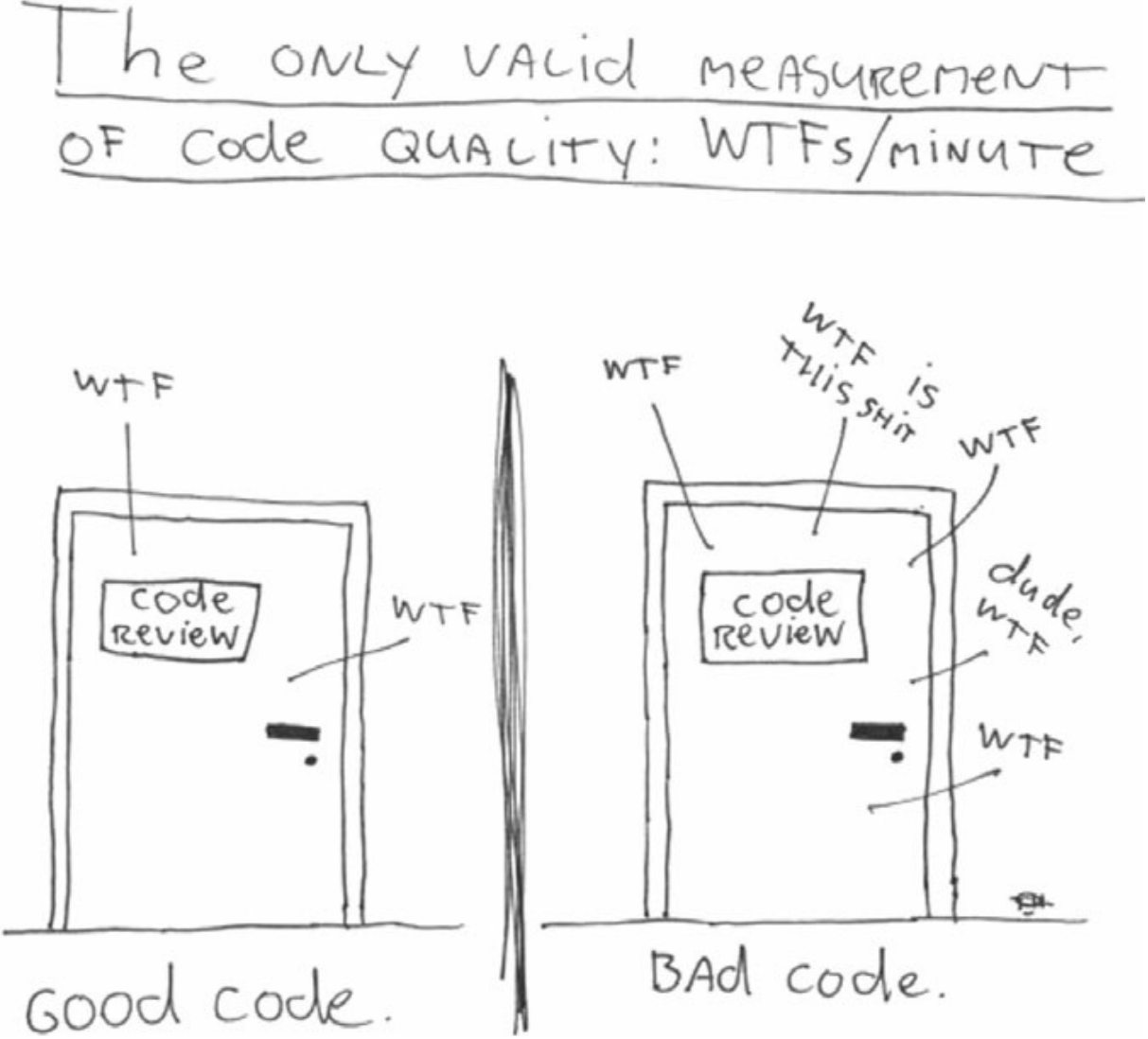
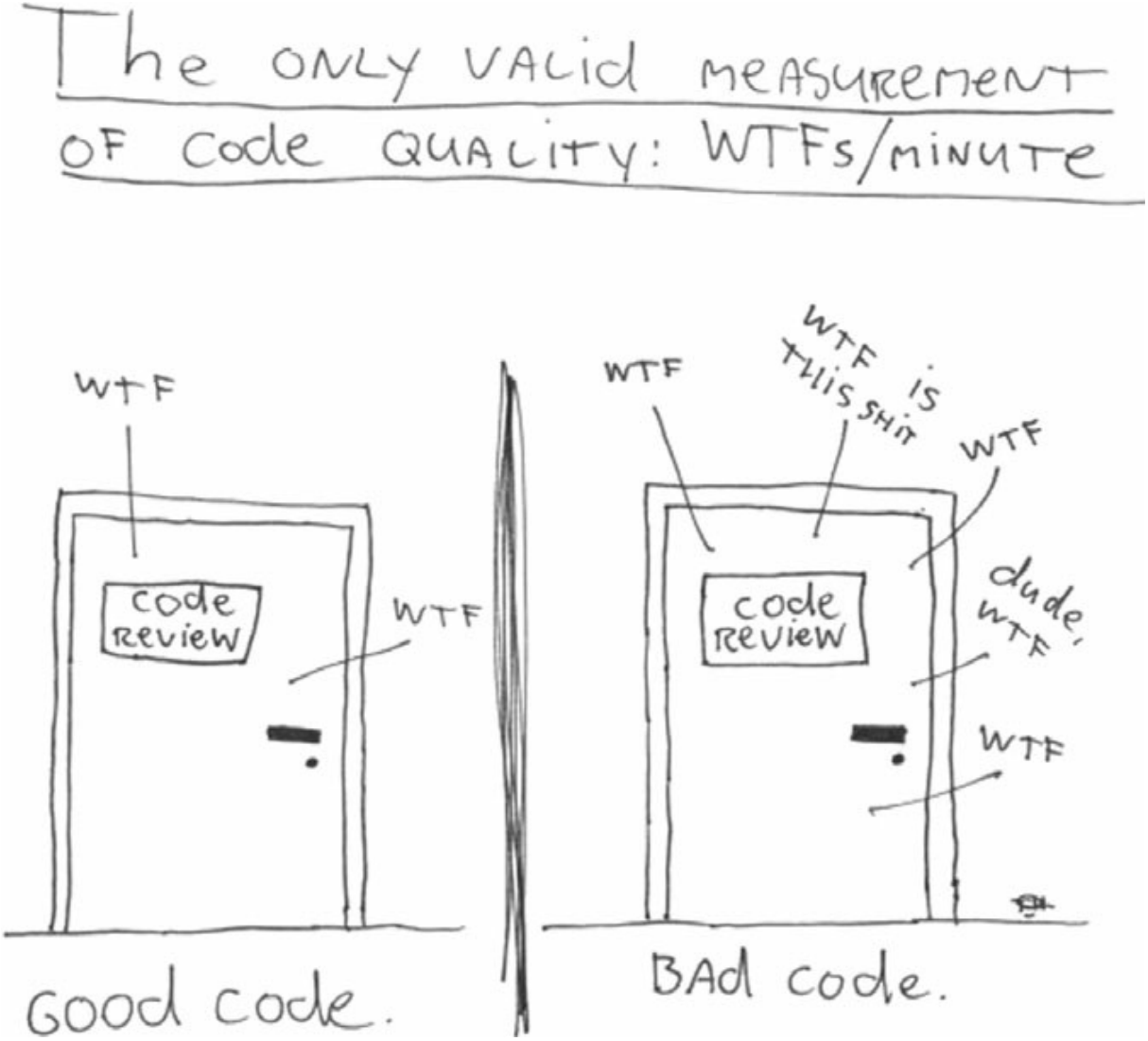


Clean Code – A Handbook of Agile Software Craftsmanship Highlights and Condensed Version

Table of Contents

Chapter 1: Clean Code



Motivation

I (Uncle Bob) know of one company that, in the late 80s, wrote a killer app. It was very popular, and lots of professionals bought and used it. But then the release cycles began to stretch. Bugs were not repaired from one release to the next. Load times grew and crashes increased. I remember the day I shut the product down in frustration and never used it again. The company went out of business a short time after that.

Two decades later I met one of the early employees of that company and asked him what had happened. The answer confirmed my fears. They had rushed the product to market and had made a huge mess in the code. As they added more and more features, the code got worse and worse until they simply could not manage it any longer. It was the bad code that brought the company down.

Have you ever been significantly impeded by bad code? If you are a programmer of any experience then you've felt this impediment many times. Indeed, we have a name for it. We call it wading. We wade through bad code. We slog through a morass of tangled brambles and hidden pitfalls. We struggle to find our way, hoping for some hint, some clue, of what is going on; but all we see is more and more senseless code.

Chapter 2: Meaningful Names

"You should name a variable using the same care with which you name a first-born child."

Names are everywhere in software. We name our variables, our functions, our arguments, classes, and packages. We name our source files and the directories that contain them. We name our jar files and war files and ear files. We name and name and name. Because we do so much of it, we'd better do it well. What follows are some simple rules for creating good names.

The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

```
int d;          // elapsed time in days
```

The name d reveals nothing. It does not evoke a sense of elapsed time, nor of days. We should choose a name that specifies what is being measured and the unit of that measurement:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Why is it hard to tell what this code is doing? The reason is *implicity*.

- What is in "`theList`"?
- Why is the digit "4" in `x[0]` so important? What does it tell us?
- What does "`getThem`" in the function name mean?

Another version of the same code (with some renaming and added constants):

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Can you now figure out the purpose of the code? What new information can we extract?

- theList is now called gameBoard (telling us that it has something to do with a game)
- the iterated array is now called cell instead of x (we could think about some chess game)
- the value 0 for indexing the array is now a constant called STATUS_VALUE
- the value 4 is now encoded as FLAGGED

Since there are no STATUS_VALUE and FLAGGED properties in a chess game, it is something different – a minesweeper game.

Member Prefixes

You also don't need to prefix member variables with m_ anymore. Your classes and functions should be small enough that you don't need them. And you should be using an editing environment that highlights or colorizes members to make them distinct.

Besides, people quickly learn to ignore the prefix (or suffix) to see the meaningful part of the name. The more we read the code, the less we see the prefixes. Eventually the prefixes become unseen clutter and a marker of older code.

Class and Method Names

Classes and objects should have noun or noun phrase names like `Customer, WikiPage, Account, and AddressParser`. Avoid words like `Manager`, `Processor`, `Data`, or `Info` in the name of a class. A class name should not be a verb.

Methods should have verb or verb phrase names like `postPayment`, `deletePage`, or `save`. Accessors, mutators, and predicates should be named for their value and prefixed with get, set, and is according to the javabean standard.

When constructors are overloaded, use static factory methods with names that describe the arguments. For example,

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

is generally better than

```
Complex fulcrumPoint = new Complex(23.0);
```

Consider enforcing their use by making the corresponding constructors private.

The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background. This is a teaching issue rather than a technical, business, or management issue. As a result many people in this field don't learn to do it very well. People are also afraid of renaming things for fear that some other developers will object. We do not share that fear and find that we are actually grateful when names change (for the better). Most of the time we don't really memorize the names of classes and methods. Follow some of these rules and see whether you don't improve the readability of your code. If you are maintaining someone else's code, use refactoring tools to help resolve these problems. It will pay off in the short term and continue to pay in the long run.

Chapter 3: Functions

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.

Do One Thing

Functions should do one thing. They should do it well. They should do it only.

The problem with this statement is that it is hard to know what “one thing” is. The level of abstraction may not be clear. Use one level of abstraction, and call a function to dive more and more into details.

Different abstraction levels may be:

- `getHtml()` // high level function to retrieve the HTML-Code of a webpage
- `PathParser.render(pagePath);` // mid-level of abstraction
- `.append("\n")` // extrem low lvl of abstraction, revealing implementation details

One way to know that a function is doing more than “one thing” is if you can extract another function from it with a name that is not merely a restatement of its implementation.

Function Arguments

Don’t use functions with more than 2 arguments without a good reason. If you need more then 2 Parameters, consider creating a Class for the parameter structure then.

Parameters, especially out Parameters can really make code readability a lot worse.

`writeField(name)` is usually easier to understand than `writeField(output-Stream, name)`

So consider making output-stream a member variable.

```
Point p = new Point(0,0);
```

Is due to its nature perfectly reasonable

```
assertEquals(expected, actual);
```

How often have you mixed up expeceted and actual? Or even worse

```
assertEquals(message, expected, actual);
```

(or rememeber Kafas: `log.StepStatus(status, mess, detail, expec, receiv)` what a mess ^^)

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Have No Side Effects

The following code shows a common example creating a side effect for unknown users calling the `checkPassword` function of the `UserValidator` class.

Side effect: `Session.initialize()` - the name "checkPassword" does not tell the developer that the session is initialized. Please do not lie to the readers of your code. Tell exactly what you are doing, don’t do more, and don’t do less.

```
public class UserValidator {
      private Cryptographer cryptographer;
      public boolean checkPassword(String userName, String password) {
```

```
            User user = UserGateway.findByName(userName);
            if (user != User.NULL) {
                  String codedPhrase = user.getPhraseEncodedByPassword();
                  String phrase = cryptographer.decrypt(codedPhrase,
            password);
                  if ("Valid Password".equals(phrase)) {
                        Session.initialize();
                        return true;
                  }
            }
            return false;
      }
}
```

Methods should do one thing - error handling is one thing - so each method containing a try-catch block should start directly with the try statement.

Chapter 4: Comments

> “Don’t comment bad code—rewrite it.”
>
> —Brian W. Kernighan and P. J. Plaugher 1

Indeed, comments are, at best, a necessary evil. Write clean code so you can omit the comments. The proper use of comments is to compensate for our failure to express ourself in code. So when you find yourself in a position where you need to write a comment, think it through and see whether there isn’t some way to turn the tables and express yourself in code.

Why am I so down on comments? Because they lie. Not always, and not intentionally, but too often. The older a comment is, and the farther away it is from the code it describes, the more likely it is to be just plain wrong. The reason is simple. Programmers can’t realistically maintain them.

Truth can only be found in one place: the code.

Legal Comments / Explaining comments / TODO comments

Legal comments may be part of the companies policy and cannot be removed by the developer, even if he'd like to. Especially license texts (e.g. BSD, GPL etc.) are most often found at the top of a source file.

// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.

// Released under the terms of the GNU General Public License version 2 or later.

// @author: xyz

The Purpose of Explaining Comments is usually to transport information in an educational environment. Actual production code should not need any Explaining Comments.

`assertTrue(aa.compareTo(ab) == -1);` // aa < ab

```
assertTrue(ba.compareTo(bb) == -1); // ba < bb
```

Possible improvements omitting the comment without reducing the amount of information given by the reader can be seen below:

- Custom Function: `lessThan(aa, ab) … lessThan(ba, bb)`

TODO Comments may be okay for maybe a week or two. If they exist for a longer period, ask yourself why you are not doing it? Whatever else a TODO might be, it is not an excuse to leave bad code in the system.

Journal Comments / Redundant Comments/ Noise Comments:

```
/*
 * 13-Mar-2003 : Implemented Serializable (DG);
 * 29-May-2003 : Fixed bug in addMonths method (DG);
 * 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
 * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
*/
```

We have version control tools now - please avoid journal comments or file history entries directly in the file.

Redundant / Noise Comments add no information at all. Please see below.

```
/**
* The container event listeners for this Container.
*/
protected ArrayList listeners = new ArrayList();
/**
* The Logger implementation with which this Container is associated.
*/
protected Log logger = null;
/**
* Associated logger name.
*/
protected String logName = null;
/**
* Default constructor.
*/
protected AnnualDateRule() {
}
/** The day of the month. */
private int dayOfMonth;
```

And then there's this paragon of redundancy:

```
/**
* Returns the day of the month.
```

```
*
* @return the day of the month.
*/
public int getDayOfMonth() {
      return dayOfMonth;
}
```

Commented-Out Code:

Few practices are as odious as commenting-out code. Don't do this!

Others who see that commented-out code won't have the courage to delete it. They'll think it is there for a reason and is too important to delete.

There was a time, back in the sixties, when commenting-out code might have been useful. But we've had good source code control systems for a very long time now. Those systems will remember the code for us. We don't have to comment it out any more. Just delete the code. We won't lose it. Promise.

This requires frequently pushing to (your) git (branch).

On the following pages, you find two complete example of the Sieve of Eratosthenes. The first is considered as a bad example, as it is violating many of the principles presented above. Afterwards you will find a much cleaner version. I hope you realize the differences and how they improve readability and decrease the time needed to understand that code.

Bad sieve:

```
/**
* This class Generates prime numbers up to a user specified
* maximum. The algorithm used is the Sieve of Eratosthenes.
* <p>
* Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
* d. c. 194, Alexandria. The first man to calculate the
* circumference of the Earth. Also known for working on
* calendars with leap years and ran the library at Alexandria.
* <p>
* The algorithm is quite simple. Given an array of integers
* starting at 2. Cross out all multiples of 2. Find the next
* uncrossed integer, and cross out all of its multiples.
* Repeat until you have passed the square root of the maximum
* value.
*
* @author Alphonse
* @version 13 Feb 2002 atp
```

```
*/
import java.util.*;
public class GeneratePrimes {
/**
* @param maxValue is the generation limit.
*/
public static int[] generatePrimes(int maxValue) {
  if (maxValue >= 2) { // the only valid case
    // declarations
    int s = maxValue + 1; // size of array
    boolean[] f = new boolean[s];
    int i;
    // initialize array to true.
    for (i = 0; i < s; i++)
      f[i] = true;
    // get rid of known non-primes
    f[0] = f[1] = false;
    // sieve
    int j;
    for (i = 2; i < Math.sqrt(s) + 1; i++) {
      if (f[i]) { // if i is uncrossed, cross its multiples.
        for (j = 2 * i; j < s; j += i)
          f[j] = false; // multiple is not prime
      }
    }
    // how many primes are there?
    int count = 0;
    for (i = 0; i < s; i++) {
      if (f[i])
        count++; // bump count.
    }
    int[] primes = new int[count];
    // move the primes into the result
    for (i = 0, j = 0; i < s; i++) {
      if (f[i])
      // if prime
        primes[j++] = i;
    }
    return primes; // return the primes
```

```
  }
  else // maxValue < 2
    return new int[0]; // return null array if bad input.
  }
}
```

Good sieve:

```
/**
* This class Generates prime numbers up to a user specified
* maximum. The algorithm used is the Sieve of Eratosthenes.
* Given an array of integers starting at 2:
* Find the first uncrossed integer, and cross out all its
* multiples. Repeat until there are no more multiples
* in the array.
*/
public class PrimeGenerator {
  private static boolean[] crossedOut;
  private static int[] result;
  public static int[] generatePrimes(int maxValue) {
    if (maxValue < 2)
      return new int[0];
    else {
      uncrossIntegersUpTo(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }
  private static void uncrossIntegersUpTo(int maxValue) {
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
      crossedOut[i] = false;
  }
  private static void crossOutMultiples() {
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
    }
  private static int determineIterationLimit() {
```

```
    // Every multiple in the array has a prime factor that
    // is less than or equal to the root of the array size,
    // so we don't have to cross out multiples of numbers
    // larger than that root.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
  }
  private static void crossOutMultiplesOf(int i) {
    for (int multiple = 2*i; multiple < crossedOut.length; multiple += i)
      crossedOut[multiple] = true;
  }
  private static boolean notCrossed(int i) {
    return crossedOut[i] == false;
  }
  private static void putUncrossedIntegersIntoResult() {
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        result[j++] = i;
  }

  private static int numberOfUncrossedIntegers() {
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        count++;
    return count;
  }
}
```

Chapter 5: Formatting

Why Clean Code Matters

When people look under the hood, we want them to be impressed with the neatness, consistency, and attention to detail that they perceive. We want them to be struck by the orderliness. We want their eyebrows to rise as they scroll through the modules. We want them to perceive that professionals have been at work. If instead they see a scrambled mass of code that looks like it was written by a bevy of drunken sailors, then they are likely to conclude that the same inattention to detail pervades every other aspect of the project.

You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply. It helps to have an automated tool that can apply those formatting rules for you.

Vertical Formatting

How much LoC for one class / source file? The less, the better! The upper limit of about 500 lines should not be exceeded (if possible). Small classes are usually easier to understand than bigger ones.

Instance variables (members, attributes) should be at the top of each class definition. Local function variables should be declared in the first lines of the respective function.

Dependent Functions

If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. (Forward declare if necessary?)

Horizontal Formatting

I personally set my limit at 120. (Uncle Bob)

Only used for indentations - please keep the loops and blocks nice formatted. (Eclipse Formatting will force you anyway)

Please do not omit the opening and closing braces / parenthesis for single line loops. (You may add a print and totally change the behaviour or in the worst case break the program)

Chapter 6: Objects and Data Structures

There is a reason that we keep our variables private. We don't want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?

A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the essence of the data, without having to know its implementation.

Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.

The complement is also true:

Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.

The Law of Demeter

More precisely, the Law of Demeter says that a method f of a class C should only call the methods of these:

- C
- An object created by f
- An object passed as an argument to f
- An object held in an instance variable of C

The method should not invoke methods on objects that are returned by any of the allowed functions.

Talk to friends, not to strangers.

prefer `lock()` over `getMutex().lock()` // one quick example .. good ? TODO

Consider why you want a specific object and what you want to do with it. Wouldn't it be better to provide a method that does the job rather than a getter to the object capable of doing that job?

Uncle Bob finds it really important to separate Data Structures from Objects, the first providing a set of (public) variables, the latter providing a collection of functions.

Objects expose behavior and hide data. This makes it easy to add new kinds of objects without changing existing behaviors. It also makes it hard to add new behaviors to existing objects. Data structures expose data and have no significant behavior. This makes it easy to add new behaviors to existing data structures but makes it hard to add new data structures to existing functions.

Chapter 7: Error Handling

Error handling is important, but if it obscures logic, it’s wrong. Use different classes only if there are times when you want to catch one exception and allow the other one to pass through.

// consider a tutorial here ... throwing maybe a division-by-zero-excep ?

// out of my head: to the topic exceptions: "catch a common, throw a custom" ...

// though "common" can be API specific ...

SPECIAL CASE PATTERN

Consider some algorithms processing List or vector data types. They can be much safer, if the programmer guarantees that - in case of an empty list - the actual empty list is returned, and not NULL for instance.

Don't return null:

I can’t begin to count the number of applications I’ve seen in which nearly every other line was a check for null. Here is some example code:

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

When we return null , we are essentially creating work for ourselves and foisting problems upon our callers. All it takes is one missing null check to send an application spinning out of control. To avoid NPE's (NullPointerExceptions) do not return null.

Don’t Pass Null

Returning null from methods is bad, but passing null into methods is worse. Unless you are working with an API which expects you to pass null, you should avoid passing null in your code whenever possible.

Let’s look at an example to see why. Here is a simple method which calculates a metric for two points:

```
public class MetricsCalculator {
  public double xProjection(Point p1, Point p2) {
    return (p2.x – p1.x) * 1.5;
  }
  …
}
```

What happens when someone passes null as an argument?

```
calculator.xProjection(null, new Point(12, 13));
```

We’ll get a `NullPointerException`, of course.

In most programming languages there is no good way to deal with a null that is passed by a caller accidentally. Because this is the case, the rational approach is to forbid passing null by default. When you do, you can code with the knowledge that a null in an argument list is an indication of a problem, and end up with far fewer careless mistakes.

Chapter 8: Boundaries

This chapter is about using Third-Party Frameworks and Packages.

Consider a data structure (List, Map) passed around the system and no of the recipients should be able to remove data from the list.

When using a data structure from a given library (no matter what: STL, java.util, any Apache or Boost library) and passing this data structure around, everyone is provided with the complete set of functions, manipulating and modifying the given data structure, such as `clear(), delete(), remove()`.

A solution to this problem might be creating a custom class with the desired data structure as a private member, providing enough public functions for all other software components to do the desired work.

Chapter 9: Unit Tests

The three laws of Test Driven Development (TDD):

- **First Law:** You may not write production code until you have written a failing unit test.
- **Second Law:** You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- **Third Law:** You may not write more production code than is sufficient to pass the currently failing test.

Clean Code also matters for Unit tests. If the software evolves and changes, also the unit tests may have to change. Test code is just as important as production code. It is not a second-class citizen. It requires thought, design, and care. It must be kept as clean as production code.
Don't care about reduced memory or reduced computing power in embedded systems. The tests should usually run on an external machine providing enough resources.

Single Concept per Test

Think about what you want to test, and if these are multiple use cases and can be split up in multiple tests.

F.I.R.S.T. Fast – Independent – Repeatable - Self-Validating (Pass or Fail) - Timely

For more information the interested reader is referred to Uncle Bobs book.

Chapter 10: Classes

Classes Should Be Small! The first rule of classes is that they should be small. The second rule of classes is that they should be smaller than that. No, we’re not going to repeat the exact same text

from the Functions chapter. But as with functions, smaller is the primary rule when it comes to designing classes. As with functions, our immediate question is always “How small?” With functions we measured size by counting physical lines. With classes we use a different measure. We count responsibilities.

The Single Responsibility Principle

The Single Responsibility Principle (SRP) 2 states that a class or module should have one, and only one, reason to change. This principle gives us both a definition of responsibility, and a guidelines for class size. Classes should have one responsibility—one reason to change. It is one of the most important, and yet most violated design principles in OOP. Many developers focus more on getting the code to run, skipping the refactoring afterwards and heading to the next problem/program feature.

This class definition, however small it is, violates the SRP:

```
public class SuperDashboard extends JFrame implements MetaDataUser
      public Component getLastFocusedComponent()
      public void setLastFocused(Component lastFocused)
      public int getMajorVersionNumber()
      public int getMinorVersionNumber()
      public int getBuildNumber()
}
```

It does more than one thing, right? It handles the software version (major, minor and build version numbers) and also cares about `FocusedComponents` (GUI).

Creating an additional class called Version would help. This class then is only responsible for the versioning of the software.

Cohesion

Classes should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables. In general the more variables a method manipulates the more cohesive that method is to its class. A class in which each variable is used by each method is maximally cohesive.

The strategy of keeping functions small and keeping parameter lists short can sometimes lead to a proliferation of instance variables that are used by a subset of methods. When this happens, it almost always means that there is at least one other class trying to get out of the larger class. You should try to separate the variables and methods into two or more classes such that the new classes are more cohesive.

Splitting large functions into smaller ones often leads to the possibility of extracting complete classes, splitting a given class into multiple class with higher cohesion, having only a few functions operating on a small amount of instance variables.

Organizing for Change

Imagine the following class, capable of generating SQL Statements to communicate with a data base.

```
public class Sql {
      public Sql(String table, Column[] columns)
      public String create()
      public String insert(Object[] fields)
      public String selectAll()
      public String findByKey(String keyColumn, String keyValue)
      public String select(Column column, String pattern)
      public String select(Criteria criteria)
      public String preparedInsert()
      private String columnList(Column[] columns)
      private String valuesList(Object[] fields, final Column[] columns)
      private String selectWithCriteria(String criteria)
      private String placeholderList(Column[] columns)
}
```

First of all, let's think about the S**ingle Responsible Principle** presented above. What are the responsibilities of the class? Creating SQL strings. Unfortunately it does not yet support update statements. So it has to change when we add new functionality (the update statement) or change existing one (e.g. the select, supporting sub-selects).

Consequently, we came up with two different reasons to change, which is a violation of SRP.

Breaking down the big SQL class into smaller sub-classes solves the SRP violation. Moreover, it also serves the **Open-Closed-Principle**, which is also quite important in OOP.

Classes should be open for extension but closed for modification.

The new design, using inheritance, has the advantage that new features can be added (update statement) by introducing a new subclass (UpdateSQL) without opening any existing class. On the other hand, when changing existing features (like sub-select), only that specific subclass has to be opened and modified. Each class has only one reason to change (when its behaviour or implementation must change).

We want to structure our systems so that we muck with as little as possible when we update them with new or changed features. In an ideal system, we incorporate new features by extending the system, not by making modifications to existing code.

Consider a PortFolio class, that calculates the values of different stocks in the Tokyo stock exchange. Testing this system with real data is hard, as they constantly change. Thus, it is a good idea to create an interface StockExchange. The parameters of the functions in the PortFolio class then accept the Interface StockExchange, and the TokyoExchange must implement that interface. This will help a lot when testing.

“Program to an Interface, not an Implementation.”

—Gang of Four (GoF)

By minimizing coupling in this way, our classes adhere to another class design principle known as the Dependency Inversion Principle (DIP). 5 In essence, the DIP says that our classes should depend upon abstractions, not on concrete details.

Instead of being dependent upon the implementation details of the TokyoStockExchange class, our Portfolio class is now dependent upon the StockExchange interface. The StockExchange interface represents the abstract concept of asking for the current price of a symbol. This abstraction isolates all of the specific details of obtaining such a price, including from where that price is obtained.

Chapter 11: Systems

"Complexity kills. It sucks the life out of developers,
it makes products difficult to plan, build, and test."
—Ray Ozzie, CTO, Microsoft Corporation

In this chapter let us consider how to stay clean at higher levels of abstraction, the system level. Software systems should separate the startup process, when the application objects are constructed and the dependencies are "wired" together, from the runtime logic that takes over after startup.

```
public Service getService() {
      if (service == null)
            service = new MyServiceImpl(...); // Good enough default for
                                              most cases?
            return service;
}
```

This is the **lazy initialization / evaluation** idiom, and it has several merits. We don't incur the overhead of construction unless we actually use the object, and our startup times can be faster as a result. We also ensure that null is never returned. If we are diligent about building well-formed and robust systems, we should never let little, convenient idioms lead to modularity breakdown. The startup process of object construction and wiring is no exception.