CLEAN CODE

BEST TIPS AND TRICKS IN
THE WORLD OF CLEAN CODING

ELIJAH LEWIS

Clean Code

Best Tips and Tricks in the World of Clean Coding

Table of Contents

Introduction

As obvious as it might appear, not everyone still agrees that clean code matters. But if you have ever had to spend hours wading through an ocean of tangled codes with several hidden pitfalls, you will most likely agree that clean code indeed matters. In fact, I dare say that any programmer with significant experience would have been impeded by messy code at one time or the other.

Of course, the degree of the slowdown may vary from one case to another. If you are lucky, you may find some hint or clue that will help you make sense of the otherwise senseless code. But there are cases where the degree of slowdown is quite significant. It may take teams of programmers years trying to sift through the mess. An otherwise fast project may slow down to snails pace. Every change you make to the code will break another part of the code. Every modification may cause the code to become even more twisted and tangled.

Over time, messy code grows even messier and more difficult to clean up. Work grinds to a halt as productivity diminishes. There are countless examples of companies that have been brought down by bad code. An otherwise good product might meet an unfortunate end due to bad codes. A badly transcribed formula, with a line of code missing, led to the Mariner 1 space rocket being wrongly launched, ultimately costing a whopping $80 million. There are several other examples of rockets mis-launch like this due to bad code. Another famous example is the race condition bug in the Thorac-25 radiation therapy machine, which caused the machine to deliver lethal doses of radiation, leading to the death of three people.

The quality of code also determines the maintainability of a product. As features are added, or changes are made to any product, time will pass, and the original developer may move on from the project or forget some details of the code. If the quality of the code was bad, to begin with, such changes could become a lot more complex and risky to make.

As a programmer, you should consider yourself as an author. While it may seem that your target audience is your computer, this is not always the case. Proponents of clean code say you should write code as if your target audience is other programmers and yourself. You (or some other programmer) will

have to read old code to write new ones). If your code is not clean, more time will be spent on reading code than on writing.

Writing clean code ultimately makes the code easier to grasp going forward. Clean code is essential for the creation of maintainable products. But writing clean code requires learning how to make disciplined use of several little techniques that are applied through an acquired sense of cleanliness. This unique code-sense is central to the art of writing clean code. It is a natural endowment for some programmers. But most people have to go through the learning process of acquiring code sense painstakingly. And this is what we have set out to learn in this book-the best tips and tricks in the world of clean coding. These tips will not only help you to recognize bad code but to write clean codes yourself.

Being able to recognize messy code is not enough. After all, being able to recognize bad art doesnt automatically make you a great artist yourself. Only a programmer with good code-sense will be able to look at messy code and see ways changes and variations can be made to clean it up. That is what this book is all about, learning the sequence of transformative behavior that will help you write clean and elegant code.

Chapter 1: What is Clean Code?

Even if you are one of those that agree that messy code is a significant problem in programming, this is just one step on the long road to writing elegantly clean code. There is still the question of how to write clean code. This all-important question can only be answered if you know what clean code is in the first place.

Writing clean code is quite similar to painting a picture. Most of us can tell when a picture has been painted badly and can easily appreciate great artistry when we see it. But being able to appreciate the beauty of good art does not make you a great painter. Still, this sense of appreciation of good art is an essential skill every great artist must-have. This same applies to clean coding too. Knowing what clean code is, is the very first step in writing it. So, Lets begin with the question. What is clean code?

Clean Code-Definitions

Like almost everything in life, the concept of clean code is subjective. Put any 2 or 3 human beings in a room, and you will most likely have varying opinions on the same subject matter. Even if they can reach a sort of consensus after long periods of argument, most times, the consensus is merely nothing more than agreeing to disagree with each person still holding on to their own believes about the subject matter.

For a field as diverse as software development, this is a problem you can expect with any concept. Developers cant even seem to agree on which programming language is the best in their field. Hence, you can expect multiple (and sometimes widely differing) opinions about what clean code is. Thankfully, clean code does not rely on any language-specific rules. The tricks and tips you will find in this book are language-agnostic principles that are somewhat generally accepted by the entire developer community no matter their language of choice.

Still, that does not mean clean code means the same thing to everyone. Here is probably the simplest definition of clean code you will find around:

> Clean code is code that is very easy to understand and also easy to change.

Of course, this definition is simple and seems to be pretty much straightforward. But a close look will reveal a problem that is all too

common. Different people have varying opinions about what easy to understand or easy to change is. Essentially, this definition has stated something without actually stating anything at all at the same time. To put things in a better perspective, we will examine some definitions of clean code put forward by various renowned programmers whose knowledge and experience we can trust.

Bjarne Stroustrup, the author of The C++ Programming Language, believes that clean code should do one thing well. Here is how he defines it.

I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.

For Grady Booch, author of Object-Oriented Analysis and Design with Applications, its all about the simplicity and straightforwardness of the code with the intent of the code clearly stated.

Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designers intent but rather is full of crisp abstractions and straightforward lines of control.

Big Dave Thomas, the founder of OTI and godfather of the Eclipse strategy, states a truth that seems all too obvious about clean code. In essence, your code can only be considered clean if other developers find it easy to read and enhance in our absence. Big Daves definition further states those features that will make any code easy to read, including meaningful names, unit tests, minimal dependencies, and of course, clarity of purpose.

Clean code can be read and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways of doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language; not all necessary information can be expressed clearly in code alone.

From these three definitions, it is possible to identify some major characteristics of clean code. These include:

Elegance: most people agree that clean code should be pleasing to read. It should make the reader smile rather than frown and grown endlessly as they try to make sense of the tangled and knotted mess you have made. Martin Golding put it this way Always code as if the guy who ends up maintaining your code is a violent psychopath who knows where you live. There is an almost generally accepted definition of what is pleasing and what isnt. So, writing code that meets this criterion shouldnt be a problem.

The consequence of inelegance is expensive. According to Stroustrup, an inelegant code will make it too easy for bugs to hide. The result is that people who try to maintain or make changes will only make a bigger mess. A stray drop of paper will be easy to spot in an elegantly arranged room. So anyone entering the room will be forced to maintain decorum and keep things in order. And even when an unruly person decides to intentionally or inadvertently make a mess in a clean room, the fact that there is no prior mess will make it easier to clean up. The case is entirely different in an already dirty room. The new user will be tempted to add litter to the room. Such a new mess will only add to the heap of rubbish already in the room, ultimately making cleanup a lot more difficult.

Attention to detail is one of the hall-marks of anyone with good code sense. Just as the overall integrity of any structure depends on the strength of individual bricks, seemingly minor details such as naming, error handling, race, conditions, and memory leaks all need to be carefully thought out and executed elegantly in clean coding.

Readability: Readability is arguably the commonest talking point when discussing clean coding. Everyone agrees that code that is hard to read (no matter the language it is written) is bad. Grady Booch takes readability a step further when he stated that clean code should be as easy to read like well-written prose. From this description, it is safe to conclude that readability isnt just about how easy it is to make sense of syntax or understand each line of code (even though this is important). But more about how it all comes together.

Brooch is essentially saying that reading clean code feels more or less like reading a good book. Seasoned authors put in all the effort to ensure that the reader no longer sees letters, words, and sentences. Instead, what we read is quickly replaced by images. A good book may feel just as real and tangible as

watching a movie. Of course, reading clean code may not feel as surreal as this, but you get the idea. Anyone reading your code should be able to see what your program is meant to do. After all, codes are only abstractions meant to solve real-world problems. It should not be speculative. If, at the end of reading your code, the reader is still wondering what it does, then you have only succeeded in writing a pile of jumbled crap that works.

Simple: another common recurring point in every of the definition examined is simplicity. Everyone agrees that clean code should be simple. Readability is hard to attain when the code is complex. Even when your software is meant to solve a complex problem, the code itself should be as simple as it gets. Complexity creates a myriad of problems. According to Murphys law, If anything can go wrong, then it will go wrong. The more moving parts and turning wheels you have, the higher the chance of something getting broken.

Complex code is even harder to fix. This is why Big Dave emphasis the need to make codes easy for other people to enhance. This is a point that cannot be overemphasized. You cannot assume that you will be the only one to read your code. This is the recipe for bad coding. The idea that smaller is better is not a new concept in product development. One of the main drives since the inception of technology of any form is to make it smaller while being faster and more efficient. This holds through for codes too.

Testable: a few years ago, tying clean coding to tests would have raised some eyebrows. But today, the growing significance of Test-driven development has made testability one of the most significant features of clean code. No matter how elegant or simple your code is, it is still unclean if it has no units and acceptance tests.

Fundamental Principles of Clean Coding

The principles of clean coding can be applied to pretty much any programming language. This is a good thing since having multiple rules for various languages will only further complicate an already difficult process of keeping your code clean as you write. In this book, we will go over some of the specific techniques, tips, and tricks that you must learn to become a master at writing clean codes. However, all of these techniques can be summarized under the following basic principles,

Keep It Simple Stupid (KISS): This is a popular design principle that has

been in use since the 1960s. The principles originated from the US navy, although it is now commonly applied in all forms of design process. This principle emphasizes the need to avoid unnecessary complexity in any design process. When writing code, one of the most important questions you should ask yourself as a developer is if there is no simpler way to write the same code while achieving the same result.

Dont Repeat Yourself (DRY): this Principle is very closely related to the keeping it simple principle earlier stated. Avoiding repetitions is an integral part of the minimalist design philosophy. This theory states that every piece of knowledge within a system must have a single and unambiguous representation. In this case, every piece of code you write must be written only once within the code base authoritatively and unambiguously.

You Aren't Gonna Need It (YAGNI): this principle is part of a school of thought in programming known as Extreme Programming Methodology (XP). The XP method is aimed at improving software quality while increasing its responsiveness to the requirements of the customer. The YAGNI principle states that only functionality that has been deemed to be absolutely necessary should be added to the system. In other words, dont add any line of code to your program, except it is doing something.

Composition Over Inheritance: one of the modern principles of design is to favor composition over inheritance. This principle simply emphasizes the need to design your types based on what they do rather than on what they are. Building based on inheritance will force you to build a taxonomy of objects from the start of your project. This will inevitably make your code difficult to modify later on as you build on it.

Favor Readability: when writing codes, it is important that you always keep in mind that your codes will also be read by humans rather than just machines. This is particularly important when you will be collaborating with multiple people on a project. And even when you are not, writing codes that are hard to read may even create development or maintenance issues for you in the future.

Various recommended techniques can help you write more readable codes. One common example is by placing a common number into a constant that has been well named. Another technique that fosters readability is using long names that say a lot about what you are naming over short, ambiguous

names. These techniques and others will be further discussed later in this book.

Practice Consistency: Consistency is by far one of the most important principles of keeping code clean. Sometimes, writing clean code is as simple as maintaining the same rules throughout the entire project. If, in any case, you need to deviate from what you have been doing at all, then explaining your new choice with comments is highly recommended.

Chapter 2: Managing Complexity in Coding

To a layman, the main work of a software developer is to write the codes for creating an application. While this has some truth in it, this is not always the case. In fact, a developers primary work isnt building applications but managing complexities. In writing codes, any developer with good code sense knows that they have to try their best to eliminate complexities the barest minimum.

Complexity is one of the biggest hindrances in clean coding. Writing codes that are too complex to read has an ultimate disadvantage: the end product is not as scalable, reliable, and robust as it is meant to be. If you want to build software systems that are easy to understand and modify, then you must understand complexities and learn the proper clean coding techniques that help to avoid this problem. This is one of the fundamental keys to writing clean code.

Before we dive into how to manage complexities while coding, you should get familiar with some of the symptoms of a complex system. This will help you to avoid them. You know your code is more complex than it should be if it exhibits any of these symptoms:

Change Amplification: while building a product, changes in design decisions will require code modifications for implementation. Sometimes, a seemingly small change might require you to modify multiple parts of your code. When writing clean code, the goal is to ensure that only a few codes will be affected by any change in design decisions. This will reduce the impact of such changes on the entire code.

High Cognitive Load: Cognitive load is the amount of information or knowledge a developer must have to complete any given task involving your code. In simple terms, cognitive load refers to how much anyone reading your code has to know about it before any changes can be made. If the cognitive load of your code is high, then there is a greater chance of bugs arising in the code. The result is that any developer working on your code will need more time to complete a specific task.

Unknown Unknowns: another symptom of overly complex codes is that the pieces of codes that must be modified to complete a given task successfully are not obvious. The problem of unknown unknowns is arguably the most

significant of all the symptoms of complexity in coding. This is because of the very nature of the problem, i.e., something a software developer needs to be aware of, but there is no way for them to find out. Hence, there is an inherent unavoidable problem that is further complicated by the fact that it is not obvious. The consequence of unknown unknowns is that there is a possible bug that will probably remain hidden until a change is made in the code.

Having a high cognitive load and the problem of change amplification in your code can lead to annoying issues that waste time and increase the cost of maintenance or changes. But it is unknown unknowns that cause the biggest issues. Developers looking at such a code will be at a loss for what to do since they have no idea at all of the potential impact of any changes they make or the potential effects of the solutions they propose to any existing problems.

What Causes Complexities?

Even if a lot of us do very little to avoid it, the truth is that no one wants to write bad code. At least not intentionally. Also, writing messy code does not mean a developer is poorly skilled. Even the best of developers can churn out badly written code for several reasons. If you are saddled with the unfortunate responsibility of untangling such complex webs of codes on a later date, it may seem the programmer was on a personal vendetta to make your life miserable. But this is far from the truth.

Several factors could lead to complex codes or make good codes turn bad in a flash. It could be some design requirement changes that affect the original design of the product, or you were simply in a hurry, trying to meet a deadline. Those are external factors that could lead to bad codes. But on the code level, what could make an otherwise great programmer write bad codes?

Most people attribute complexities in coding to two major things: Dependencies and obscurities. Although external factors such as time and product management decisions may indirectly force a good developer to make a bad coding decision, these bad decisions are commonly manifested in these two major ways. Thus, if obscurities and dependencies have too much impact on the development process, you must understand the principles behind them.

Dependencies: a dependency is said to exist when a given piece of code cannot be understood or changed in isolation. Given the nature of dependencies, it is safe to say that it is normal to have a dependency in every code. Programs are built from complex algorithms that require the interaction of various codes. No code can exist in isolation. So in every program, any given code will relate to other codes in one way or the other. Hence, these other codes must be considered if a code is to be modified for any reason.

Thus, we see that dependency is a normal part of any software design process. It cannot be completely eliminated. Every time you create a new class or add a new function to your code, you create dependencies around the API for that function. However, the goal of clean coding is to reduce the number of dependencies to the barest minimum. You cant do without them, but you can make them as few and far between as possible. Also, for the few dependencies you do create, clean coding protocols will require you to make them as simple and obvious as you can. Usually, it is the obscurity of dependencies that makes codes really complex.

Obscurity: As the name implies, obscurity occurs when vital information is not as obvious as it should be. Whenever it is not obvious that a dependency exists, an informed reader of your code might make changes that will break or complicate other parts of your code. Inconsistency is one of the major causes of obscurities in coding. It could also be as a result of inadequate documentation during the design process.

This is a common problem that developers or organizations that choose to implement Agile methodologies in design often encounter. In many cases, such organizations do away with documentation because:

1. They take individual interactions over tools and process and
2. They believe in building working software over creating comprehensive documentation that explains dependencies and makes them less obscure.

Because the intent, reasoning, and objectives of code are almost always difficult to decipher by merely reading a code, dependencies and obscurities become major problems that are difficult to tackle, especially in the absence of the original creators of such a project.

The three symptoms of complex coding earlier described (cognitive load,

change amplification, and unknown unknowns) can be said to be direct results of dependencies and obscurity. The fact that codes are dependent on each other leads to change amplification as small changes have bigger effects on multiple codes. The presence of several dependencies also means cognitive load will be higher. A developer will need to learn all about how the various aspect of the codes interacts to make an informed change. Of course, the biggest problem of all is that which is not known. Obscurity is what creates the unknown unknowns that bug codes and give developer sleepless nights. The solution to complex codes, therefore, is to learn techniques and tricks that minimize dependencies and makes their existence obvious where they inevitably exist.

Complexities Build Up

The concept of iterative, incremental complexity is one of the most important concepts to grasps to master clean coding. Codes dont just go from good to bad with the stroke of a keyboard. Bad codes build up over time. A single obscure dependency is rarely a problem, but multiple ones will significantly make it difficult to maintain software systems and make building on them even more dangerous. Such complexities only build up over time. Eventually, they become so much that it is difficult for a third party to understand your code or change anything without breaking something.

It is this incremental nature of complexities that makes bad code terrible and difficult to read and control. Most times, developers may excuse the addition of a new complex layer of code as no big deal (that is if they are conscious enough to recognize the fact that complexity has been introduced in the first place). The downside, of course, is that one is rarely enough. Unless a conscious effort is made to reduce complexity, it will surely accumulate. Worst still, once it does accumulate, it is extremely difficult to eliminate. In many cases, complexities become so overwhelming that a grand redesign seems like the only viable solution.

At its very core, clean coding practices involve eliminating or attempting to reduce complexity while building a software system. Hence for anyone interested enough in keeping their code clean, the art is to learn all the little techniques of clean coding and be disciplined enough to pay attention and apply them as you build. This, however, is never as simple as it sounds. Clean code solutions are neat and efficient, but the challenge is in the

willingness to apply them when they are needed the most.

Chapter 3: Naming Things - The Clean Coding Way

Lets take a bit of a refresher course on the law of thermodynamics. The law states that disorder will increase in any system unless energy and work is expended to stop it from increasing. This law holds true for writing clean code as well. An effort is needed on your part to keep your code clean as you write. It is hard work that takes lots of practice and focus throughout the development process.

Writing clean code is all about the readability of your codes. Hence, making small changes in little things such as variable or class names can make a world of difference. The reason for this is simple; names are used everywhere in coding. We give names to various parts of a program, including variables, functions, classes, arguments, packages, and so on. We also name directories and the source files they contain. Since names are used so much in coding, learning to create good names is arguably one of the most important skills in clean coding.

Names are important for communicating the intent of your code to anyone who will be reading it. This includes even yourself. You may find yourself spending hours untangling messy code that you wrote yourself simply because you failed to follow professional guidelines for correct naming.

Imagine you are a teacher in a class where all the students have the same name. Imagine the levels of confusion you will have to deal with on a daily basis. To solve the problem, you will most likely give each student new nicknames based on their attributes or characteristics.

We do it every time, even without giving it much thought. It is not uncommon for a group of friends to have a tall Joe and a short Joe just to make it easier to distinguish between them in conversations. Much more so, a complex program with several functions, classes, and arguments you need to name. Choosing good names takes time, but if you stick to it, you will end up with cleaner and much better code. Developers and your future self will thank you for it. In this chapter, we will go over some of the dos and donts of naming in clean coding.

The Dos

Use Names That Reveal Intent

You see, this seems like this most obvious rule (and it really is). Still, it is one of the most commonly broken naming rules in coding. Choosing intent-revealing names is just as straightforward as it sounds. The name of anything in our code should tell you why it exists. It should also give you an idea of what it does and reveal its usage within your code. As a general rule of thumb, you dont need comments to explain what a name does. If someone would need to check the comment to get all of this information, then the name you have assigned is doing a bad job at revealing intent.

There are several possible examples of ways people name things that show no connection at all to the intent. For instance, simply writing int s; reveals nothing about the intent of the name. You may go further and add a comment about what s does, but thats not clean coding. The better practice would be to choose a name that clearly specifies what d measures and possibly the unit of measurement. Thus, instead of adding a comment like this //time elapsed in seconds, choosing a name like int Dimensions is a much better nomination.

Choosing an intent-revealing name can make it easy to understand a code and even a lot easier to make changes. Sometimes codes are hard to read, not because the expressions are complex. Messy codes might still have proper spacing and indentation with very few polymorphic methods or fancy classes. Still, the purpose of such a code may be uncertain simply because of how things have been named.

Switch the names for less ambiguous ones, and you will see the code transform into something a lot more readable and less vague. It remains simple with the same numbers of constant, operators, or nesting levels it has always had. Using the intent-revealing names, however, will make the code more explicit. Good names will ultimately make it easier to understand what is going on in your code.

Make Meaningful Distinctions

Most programmers write codes solely for machines rather than humans. Your code will run, but humans will have a hard time reading and making sense of what you have written. A lot of problems will arise when you write programs mainly for compilers and interpreters.

For instance, because compilers will not allow you to use the same name for

two different things within the same module, you may decide to change one of the names slightly in a way that satisfies the compiler. Typically people intentionally misspell a name or change it in any arbitrary way. Your computer might accept this, but you are creating potential problems for anyone reading your code or trying to make changes. Another programmer might assume the misspelling in your code as an unintentional error and make corrections. Of course, the machine will not recognize the new code, and it will fail to compile due to spelling errors.

Another common mistake is to add a number series to names distinguish between them. Again, while this might satisfy the compiler, the human reader may not see the distinction or may assume it to be an error. If different names are used, you should make them as uniquely different as possible. Giving variables names like a1, a2, a3, and so on isnt good enough. You can also use articles like a, an, or the as prefix for names as long as it will make the names meaningfully distinct.

Adding noise words that make two names different without changing their meaning isnt going to work too. The names ProductInfo and ProductData may look different, but Info and Data are still too similar to noise words to make a world of difference in your code. Making distinction with noise words like this is particularly tricky. Some noise words are obvious but even commonly used, which are examples of bad coding. Here are some of them:

- You should never add the word variable as a noise word in a variable name.
- The name of a table should never carry the word table as a noise word
- A name is almost always a string, so giving it a name like NameString is pointless.
- Dont use a noise word to try to distinguish between two class names. E.g., the class names AccountInfo and Account are still too similar despite the addition of Info as a noise word.

There are just a few examples that may make it difficult for programmers in your code to know the right functions to call due to the similarity in convention. Hence, it is better to distinguish names in ways that the differences are clear and easy to detect.

Use Pronounceable Names

Using pronounceable names is one of the key things that makes code readable. While code syntax is usually different from human language (even for the most human-friendly program), humans still read code with their spoken language to a large extent. Hence, making names pronounceable is one of the key ways to ensure that anyone reading your code will find it easier to make sense of it.

The human brain is naturally conditioned to simply gloss over and skip anything that cannot be pronounced. You know it’s there, and youll probably remember it. But it doesnt really register as much as it would have if it can be pronounced. More so, you dont want a group of programmers trying to look over your code pronouncing silly-sounding names. When the names are pronounceable, they can make intelligent conversations about variables in your code without anyone wondering they are speaking in an alien language. With time, the original intent of the name might be lost, or the name confused some for something else entirely due to wrong pronunciation.

Therefore, as much as abbreviations will make your code shorter and look simpler, it will still be more difficult to read compared to when you use clear and complete names.

Use Searchable Names

When working on several lines of codes, one of the quickest ways to find things is by simply searching through the code. This shortcut may end up being a fruitless venture due to bad naming practices. It is easier to search for class names that are specific and elaborate like Max_Student_Per_Class compared to single-letter names or constant. Try searching for a variable with the name e in your program, it will most likely show up in every module of text in your program and you will either have to look for ways of sifting through the long list of names to find the exact one you are looking for or abandon search entirely,

If you are using single-letter names at all, then you should only use them as local variables within short methods. But for constant or variables that will be used or reoccur several times within our code, then a more searchable name is recommended.

Understanding Solution and Problem Domain Names

One of the jobs of a good programmer is being able to identify and distinguish between solution domain concepts from the ones that are in the problem domain. This knowledge will come handy in naming. A code which has more or solution domain concept should have names related to the solution domain and vice versa.

What are Solution Domain Names? Solution domain names technical names that are easy for programmers to recognize. They are more of computer science terms, pattern names, mathematical terms, and so on. They are names that programmers will find it easier to recognize and relate with. There are names that a programmer will easily recognize in any code. For instance, any programmer familiar with VISITORpattern will not have a hard time recognizing a name like AccountVisitor. Names like JobQueuwas will also resonate to a programmer

What are Problem Domain Names? Problem domain names are names that are more commonly related to the concepts of the problem that are being solved. A solution domain number should be your number-one goto for naming (after all, programmers are the ones that will be reviewing your code). However, when there is limited programmer-friendly terms for what you are doing, you can use names from the problem domain as well. A quick check with a domain expert will help any programmer makes sense of the domain name you used if they run into a problem.

Add Meaningful Context to Your Code

You can help readers make sense of the names you use through the use of well-named functions, classes, or namespaces. This is particularly important when you are using names that are not meaningful on their own. You can also try adding distinctive prefixes to variable names to help put them in context.

For instance, adding the prefix addr to variable names like firstName, lastName, city, state, and houseNumber will add some context to them. Now when people see the variables written as addrState or addrCity, it will be easier for any reader to attribute them to a larger of a larger concept (i.e., they are all part of an address) compared to if city or state was merely written along.

Use One Word Per Concept

For every one abstract concept you are trying to name, it is best to choose to

use only one name for it and stick to it throughout your code. Words like get, fetch, and retrieve are similar, but juxtaposing them as equivalent methods for different classes will only leave a reader confused. It will be difficult to determine which method name goes with each class.

When words are properly named, modern editing tools can help you with context-sensitive clues that can help you find objects in your code faster. But when you use the same name for more than one concept, you may end up calling a different object than intended.

The Donts

Avoid Disinformation

Some names tell you nothing at all, but even worse, some tell you something else than they are intended. As a programmer, you should try to avoid using names that give meanings different from your original intention. Here are some common scenarios

Using names and abbreviations similar to common programming terms

For instance, using abbreviations that mirror the name of Unix variants or platforms such as sco, aix or hp are not good for your code. Even when such an abbreviation is close to the intent of your code (e.g., sco for score) as long as its use can be easily confused with something else that is specific to programming, it should be avoided.

There are countless examples of words like this. For example, using the word list as a prefix to the name of any container that is not actually a list is bad convention. This is because the word list would mean something to a programmer. Hence, if you are not referring to a list, go for better substitutes like group, bunch, or any related term that still explains intent without misinforming the reader.

Using similar names with differences too subtle to be easily noticed:

Another way misinformation may occur while coding is when you use similar names with subtle differences that are hardly distinguishable. Look at these two names for examples

ABControllerForCorrectHandlingOfStrings

ABControllerForCorrectStorageOfStrings

While these are two distinct names, their close similarities will inevitably lead to ambiguity and disinformation.

Inconsistent Spelling

This is another likely cause of misinformation in naming. When similar concepts are spelled similarly, or spelling is inconsistent, information other than originally intended may be implied. Modern coding environment may further worsen the problem of bad spelling in naming, especially with the use of auto-completers in coding. Now, when you write a few lines, the IDE may quickly present a list of possible names to complete. If the differences are not as obvious as they should be, a developer may choose a closely similar name for an object. Since your well written descriptive comment is not available alongside the suggestion, an awful error might be made.

Problems with upper and lowercase usage: the lower case of some letters like the letter L or uppercase of the Letter O dont make good variable names, especially when you use them in combination. They look quite similar to constants 1 and 0. This may disinform a reader all too easily. For example, take a look at this code:

int a = l;if (O == l) a = O1;else l = 01;

This a problem that is likely to deepen with some types of fonts or if the original author of the code is not available to provide clarification. Completely changing such names and avoiding them will make it easier to grasp the code without having to guess the intended meaning.

Avoid Encodings

Encoded names are difficult to deal with. They are better avoided since they only add to your burden rather than ease it. Adding scope information or encoding types to names will only introduce something else that must be deciphered to make your code readable. Worst still, encoding is rarely pronounceable. They can also introduce a new range of problems to your code since they are quite easy to mistype.

Adding encoding into names is a mental burden that can (and should) be avoided. There was a time when encoding names was necessary and super important. Back then, Hungarian notation was considered as super important since everything was either an integer handle, void pointer, or long pointer. In todays modern language system, this is not the same. We now have compilers

that can check and enforced data types. The use of shorter functions and small classes that allows you to see the point of declaration of every variable also makes encoding only mere impediments rather than necessitates. They made it harder to read code and make modifications even more difficult.

Avoid Mental Mapping

Remember what we said about problem domain names and solution domain names? When you fail to use names from either of both domains, anyone who reads your code may interpret the name to mean any other name they know. This may change the context of the name and hence lead to disinformation. This is the exact problem with using single-letter names (except in loop counters where using letters like j or k are traditionally acceptable). Otherwise, single letter names will only appear to a reader as a placeholder that the reader will inevitably try to map and match to an actual concept.

The fact that you are writing codes that another programmer will read one-day calls for consciousness. Programmers are generally smart people that are highly prone to mind mapping. As a professional, you should focus on making your code as clear as possible so that others like you can easily understand without making unnecessary deductions.

Dont Be Cute

As far as clean coding is concerned, it is always better to say exactly what you mean instead of trying to be cute. As tempting and harmless as they seem, colloquialisms and slangs dont really have a place in coding.

When writing code, trying to be clever or using entertaining and amusing names will only work if you are the only one ever that will read your own code. Since this is rarely the case, it is safer and more professional to use names that are not culture-dependent or clever. A few people that are in on the joke you are trying to make might get it too, but thats about all that you get. At the end of the day, dont take being clever over the functionality and clarity of your code.

Avoid Unnecessary Context

While meaningful context is great for coding, adding a gratuitous context is a terrible practice. Overkill is terrible in anything, including coding. Adding context when it is really not needed will only complicate your code in very

bad ways. For instance, it is a bad idea to add a prefix to every class in an application you are creating in the name of adding context to our code. Doing this will turn the search tool in your IDE against you at some point. For instance, an attempt to search for a specific class will turn up a long list of classes in the code since they all share the same prefix. It will also make coding more complicated than it is meant to be.

Additional Tips for Naming Classes and Methods

Class Names

- Rules for creating class names
- Class names or object names should be made up of nouns or noun phrases
- You should not use verbs as class names.
- Names like Customer, AddressParser, Accounts and so on will work as class names
- Class names to avoid includes names like Processor, info, Manager, and so on. Do not use them as either class names or even prefix to a class name.

Method Name

- Generally, it is recommended that you use only verb phrases or verbs for methods names.
- When constructors are overloaded, it is recommended that you use a factory static method that has a name that clearly describes the argument.

Naming things correctly while coding is basically a skill. It requires you to be good at describing things. Sadly, this is a skill not many people have learned or mastered. Because it is difficult, if not virtually impossible, to memorize the names we use in our codes, it is best when they follow all the rules in the book as closely as possible.

Most modern programming tools also rely on how well classes, methods, or functions have been named in a code. Thus, improper naming habits can lead to even bigger problems when you turn to these tools to aid readability. Following the dos and donts stated in this chapter will help you improve the readability of your code and improve the overall quality of your program.

Chapter 4: Functions

In the last chapter of this book, we delved deeper into the concept of naming in writing clean code. We could see how a seemingly small thing like using meaningful names or using names that reveal intention could transform code completely. When names are used correctly, the overall quality and readability of code are improved.

In this chapter, we will go deeper into the world of clean coding by discussing Functions. A function is another simple and basic mechanism of coding that can impact the ease of maintenance of code. Functions, when written right, can also make it easy to extend code or make corrections.

A function is the first line of organization within any code. In times, past, codes were made up of various systems of organizations, including programs, subprograms, and functions. Functions are the only ones that have survived to this day.

Imagine reading a book where all different font sizes are used for a different part of it. How awful would that be? Imagine if all the paragraphs of that book were disorganized and mixed with some chapters having as much as 20 pages. Of course, reading such a book will be extremely difficult. That is exactly how it would feel like to read a program with bad code.

The Top to Bottom Rule

The code should be written in a way that makes it possible for anyone to read it like a book. Others should follow every function in your code at the next level of abstraction to it. This way, when the program is read, the reader descends from one level of abstraction to the next in a top-down manner with each different logic grouped in the code grouped separately. This rule is also commonly referred to as the step-down rule.

One could also say that programs should be written like a set of TO paragraphs. Each To paragraph should describe a specific level of abstraction while making reference to subsequent TO paragraphs at the next level below it.

As simple as this rule sounds, it turns out to be quite difficult for programmers to follow. Writing functions that stay at just one level of abstraction is a basic but typically complicated rule to follow. But for those

who have mastered this trick, they will find it easier to keep their functions short and ensure that they only do one thing. This is key to maintaining consistency in abstraction level withing your code.

Keep Functions Small

This is arguably the most important rule for writing functions. You should keep functions as short as you possibly can. The reason for this is quite simple; writing a short function ensures that you write a function that only does one thing. Additionally, a small function is easier to understand and maintain.

Of course, the big question is, how short is short enough? Unfortunately, it difficult to answer that question because it would be wrong to generalize it. How long a method should is dependent on several factors such as code conventions like how often a programmer hits the enter key to add a new line to the code.

However, as a general rule of the thumb, if your method is more than 10 to 15 lines long, then you should take a look at it again and try to figure out why it is so long. Try to see if it is that long because of code conventions, or there is too much logic in the code.

There is no rule or holy grail that says functions must be short to a specific length. But writing short functions will definitely give your work more order and organization as much as it is possible to try to keep your function to a maximum of three to four lines. Each one should be transparently obvious. It should tell a story that led to the next one simply and compellingly. This is how short your functions should be.

Blocks and Indenting

Another thing you should pay attention to keep your functions short is blocks and indenting. As much as possible, try to make your IF, ELSE, WHILE and REPEAT statements just one line. Writing an If statement that is ten lines long will make your code hard to read or understand. To ensure this, you can extract the check to form a separate function then call it with the IF statement.

Applying this rule should help you keep your If or While statement at the shortest length possible. This will improve the readability of your code as well and make documentation clear and precise. Developers who look at the

code will find it easy to tell what it does and what the If and While statements are meant to do.

Additionally to this, you will improve the readability and documentation of your own code. For developers, it will be very easy to understand what the code does and what the check behind that IF or WHERE is supposed to do.

Do One Thing

Long functions do more than one thing, and that is why they are so bad. You might have come across or even written several of those in the past. The same function might be used for opening the DB connection, execute a query, conversion of results to other types, and the handling of special cases all in the same file. This is bad practice and should be avoided if you want to keep your code clean.

Functions should do one thing only, and they should do it well. If there is a need to carry out more than one task, then it is recommended that you split up your code into separate functions.

How to Tell if a Function is Doing One Thing?

But how can you tell if a function is doing only one thing? Thats the tricky part. Generally, the functions stated name could give you an idea. Even if a statement has more than one step, as long as the function is doing only those steps that are one level below its stated name, then it does only one thing. This aligns with the purpose of a function, which is to break down a large concept (captured by the functions name) into a set of steps at one level of abstraction below it.

Another way to tell if a function is doing only one thing is to see if it is possible to extract another function from it. Such a new function must not be a mere restatement of the original function. If the statement is simple enough, extracting another function from it will be quite hard. However, if you can still extract a part of a function to form a separate function with a meaningful name, then it is doing more than one thing.

Another mechanism that tells us if a function is doing more than one thing is the level of abstraction. Every function you write should have only one level of abstraction. For instance, if a function processes an entity, splits, and transfer one of its fields, then it has more than one level of abstraction. In such a case, you should extract some of the statements within the function

into another function. This way, only statements at the same level of abstraction will be within any one function.

Switch Statements

Switch statements are quite difficult to simplify. Yet, it is almost impossible to do without them. It is hard to write a switch statement with just one case and even harder to write one that does only one thing. However, since we cannot do without Switch statements entirely, we can ensure that each one of them is contained within a low-level class. You should also make sure that your switch statement is never repeated. This can be done by making use of polymorphism.

As a general rule, you should only use switch statements if they will appear only once in your code, and you intend to use them to create polymorphic objects. Switch statements should also be hidden as deep as possible behind an abstract factor/inheritance. This way, they are invisible to the rest of the system.

Use Descriptive Names

Naming things correctly cannot be overemphasized in writing clean code. This applies to functions as well. A functions name should describe what it does. While it is true that naming takes time, be willing to take the extra time required to choose the right name. You can try as many names as possible until you find one that accurately describes your function. This will help you in clarifying the design module of your code in your mind.

Consistency is also important in naming functions. You should use the same phrases, verbs, and nouns as that of your modules. By using similar phrases, the name of your function sequence will tell a complete and comprehensive story. Hence, it is best to maintain the same name pattern, especially when the functions you are naming are similar.

Function Arguments

What is the ideal number of arguments that a function should have? Niladic functions (functions with zero arguments) are the best. However, since this is not always possible, functions with one or two arguments are allowed too. Each time you add a new argument to your function, you should consider its role before you make the change.

The value of arguments should never be up to 3. if you have a function like this, you should consider moving the function to a different location or adding another level of abstraction to it.

Arguments are difficult and require lots of conceptual power. Thats why its best to get rid of them entirely or keep them to the barest minimum. Instead of keeping an argument in your function, you can simply make it onto an instance variable. This way, a developer reading through would not have to interpret it. Most times, the interpretation will only provide details that arent particularly important to a reader since the argument is at a level of abstraction different from that of the function name.

Arguments are even more difficult to deal with when you have to write test cases. It is simpler if there are no arguments at all. One or two arguments are also a bit easy to deal with. But with more arguments, writing each of the test cases will be a lot harder. This is another reason why you should keep arguments to a maximum of 2.

An input argument is recommended over output arguments. They are easier to read and understand. This is because when a person reads a function, they usually do so expecting information to be going in rather than out. Hence, it takes longer to make sense of output arguments. If you would be using arguments at all, making them input arguments is recommended.

Functions Should Have No Side Effects

when your function does more than one thing without telling the client, it is said to have side effects. A side effect is a lie and hidden mistruths. For instance, a Read method that reads the contents of a file, then thereafter, deletes it, notifying the user who has a side effect. In this case, the user should have been notified, or the name of the function modified to show that it performs other functions asides reading the content of the file. A method name such as ReadAndDelete would have been a more fitting descriptive name.

Side effects lead to order dependencies and temporary couplings. These are functions that are only safe to be called at certain times. One way of dealing with this is to write the function such that only the methods that are available at a specific time are exposed. For instance, if you have a function, GoLeft, which can only be called if a function StartEngine is called. You can write

the function such that StartEngine returns an object with commands like GoRight, GoLeft, and so on.

Command Query Separation

Any function should do only one thing. It is either it is executing a query or issuing a command. Writing a function that does both will only lead to confusion. For instance, you should never create a function that can change the state of a variable and return some information about the variable. These two actions (commands and queries) should be separated at all times with no exception to the rule.

A common way programmers subtly violate this rule is when they write command functions that return error codes. This promotes the use of commands as expressions in If statements. This creates two additional problems for the developer. He must be aware of the mapping of each error code and also has to check how the error code is returned. For such cases, it is best to throw an exception as this will simplify the clients work

Writing command function that return error codes violate the rule about functions doing just one thing. Error handling is one thing. Thus, when a function that does that should do nothing else. You should separate error handling completely from your logic.

Other Rules for Simplifying Functions

Extract Try/Catch Blocks

Codes that contain try/catch blocks are pretty difficult to read. They are usually ugly and long. They also combine normal processing with normal processing, and this will only disorganize your code structure. Hence, it is best if you extract all such try/catch blocks into a separate function. You can put the Try block into one function and the catch block into a different function on its own.

Dont Repeat Yourself

Duplication is a serious problem in coding. It makes the code longer than it should be without serving any additional function. Sometimes duplicate codes are hard to spot, especially when they have been intermixed with some other code or the duplication is not uniform.

Duplication is such a big deal in programming that several principles and

techniques have been created mainly to eliminate it. For instance, all of the normal forms of Codds database are aimed at eliminating repetitions in coding. Structured programming, object, and aspect-oriented programming, as well as component programming, serve the purpose of eliminating duplication in coding

One simple way you can solve the problem of repetition is to extract all duplicated code into a separate function. This will reduce the length of your code and will ultimately make it easier to make changes or modifications of any kind to your program.

When it comes to simplifying and writing clean functions, you should keep in mind that small things matter. You dont need to be a super crazy programmer to write functions that are short and elegant. By following all of these recommendations, you should be able to write functions that are easy to read and even easier to maintain many years from now.

Chapter 5: Stop Writing Code Comments

In writing clean code, the rule for writing a good comment is simple; stop writing them. While this may seem counter-intuitive, it is a valid truth if you think about it. One of the most obvious signs of messy code is that it comes with lots of comments.

If there is anything every programmer should aim for, it is to write code that will be so clean and expressive that comments are barely needed. You should write your code in such a way that every class, function or variable has an implicit name and structure. If your code needs comments to explain what it does, it simply means the code isnt as expressive as it is meant to be, and that speaks very bad of your ability to write clean code.

People who read your code shouldnt need to read comments to get a clear picture of what your code does. Instead of comments, well-named functions and classes are good enough to guide anyone who reads the code much like a nicely written prose.

When people look under the hood of your code, there should no surprises. Sometimes we try to prevent this surprise by simply explaining away what the code does. But as you will see in this chapter, there are several reasons why this method isnt a good solution. But why do we hate comments so much, and why should you do too? Here are some of the reasons why comments arent always as good as they seem:

Why You Should Avoid Comments

They Cover Up Failures

We started this book with how to name things correctly. This is because naming things accurately is the simplest, most important way to tell anyone what your code does. But what comes to mind when you see a comment above a function or variable explaining what code is meant to do?

On the one hand, it seems like a helpful thing to add. Oh, something to tell me what this code does? But wait, isnt that what the variable and function names supposed to do? When you think about it, any programmer who adds a comment to code was most likely unable to be expressive enough with names. Either that or the function is doing something else asides its intended

purpose.

Comments cover up failures, and thats why they are so bad. Rather than write comments (even good ones), it is better to put all your efforts into accurately naming every part of your code precisely. This way, other programmers can easily decipher what your code does with no need for comments.

Lets consider an example:

```
// find employees by status
List<Employee> find(Status status) {
  ...
}
```

As you can see, the name find does not really give a precise description of the code. Hence, the author had to add a comment to explain what the code does. Problem solved, right? Well, what happens if you see the same find function is being called from another module in the code. Now there is no way to tell what the function does. So unless this developer intends to add a comment wherever the function appears (which would be another messy move), then he has failed at expressing himself and writing clean code.

For a much cleaner code, the author could have written the function name differently, like in the example below.

```
List<Employee> getEmployeesByStatus(Status status) {
  ...
}
```

The function name getEmployeesByStatus is obviously a more accurate and better name to give that function. No comment is needed here to tell any ready what that function does. Adding a comment to this new code will be redundant. That brings us to another reason why I think comments arent so cool.

Comments Lie

Comments dont always tell the truth, and thats another reason why you shouldnt use them.

```
public class User {
  ...

  // this holds the first and last name of the user
  String name;

  ...
}
```

Look at the example above; the comment tells you that the string name holds both the first and last name of a user. This should help anyone reading this code in the future. But while this comment is true today, will it still be telling the same story many years later.

One day a new programmer might come along to make one of the several requirement changes or refractors to the code. After the user makes the changes, the code now looks like this:

```
public class User {
  ...

  // this holds the first and last name of the user
  String firstName;
  String lastName;

  ...
}
```

The new developer has split the variable name into firstName & lastName, but sadly, the comment above says something else entirely. To keep comments true and valid, you will have to change them whenever you make any changes to your code. But how can you tell a new change you want to make isnt going to comment somewhere in your code invalid. And do you even want to start manually editing all the comments in your code? Thats adding more stress and time to an already complicated process. To avoid this problem entirely, it is simply advisable to keep comments to the barest minimum.

So when is comment good and when is it bad? Lets see how that works.

Bad Examples of Comments

Ugly Code

Usually, people add a comment to code when they are trying to fix messy code instead of cleaning up the code itself. When you have an ugly code, comments wont fix it. You cant suddenly make bad code beautiful by explaining why it is so messy. Youll probably do a good job at explaining, but your code is still the same terrible mess that it is. If you have an ugly code, just refactor it and rewrite it in a way that makes it look good to read.

Code Explanation

This cannot be emphasized enough. You should always use code to explain what codes are doing. A comment is not the right way to explain code. Before you add a comment to explain your code, try to see if you can rename the fields, or switch around any other element that will make it easier to understand what is going on. For instance, you can extract a method and give it a meaningful name. This way, anyone reading the code can easily understand what is going on based on just the field names alone.

Mumbling

In many cases, the comments on code are nothing more than the programmer talking to himself. It is a terrible practice to add a comment to your code merely because it seems like the necessary thing to do. With no real background or reason for adding a comment, you are only littering your code. Readers will find your code clumsy and difficult to read or understand.

Redundant Comments

A good name is a perfect substitute for a comment. Once you can find a good name for a method or field, then a comment that describes what the method does is no longer needed and should be removed. For instance, if you have a method with the name SendEmail, it is clear from the name what the method does. No additional comment is needed, and you shouldnt add one just because you feel you want to.

Any comment that is not more informative than an already informative name is not needed. Comments should justify the intent of code or give information about the codes intent. It should not be harder to read than the code itself.

Misleading, Mandated and Noise

Like every piece of text, comments can be misleading, especially if it is not accurately written. Some developers inadvertently write comments that do

not express their true intention. This will only lead to confusion when a new user is interpreting it.

For instance, you may end up with a code whose comment says that the function sends a mail to the customer. But while debugging to figure out why the code isnt sending the email as intended, you may discover that the comment misled you all along. The code was never meant to send an email, as the comment stated. Instead, the method only constructs an email that will be sent to a user. Misleading comments like this may occur because the author used information that is known only to him. Or the sentence is simply not precise enough to be accurate.

A mandated comment is another problem developers have to deal with. Typically, some big companies may have rules that mandate developers to add a comment to every method or class created. If it is up to you to decline that rule, then you should. Adding comments where it doesnt add any true value to the code will only clutter things up and pollute your code. For instance, you do not need to add comments to constructors for any reason at all since the scope of any constructor is always clear.

When Are Comments Okay?

Comments arent always bad, though. In fact, there are still a lot of developers that are diehard fans of adding a comment to code. To find common ground, it is safe to say that comment is, most times, a necessary evil. You should always aim to make your code explanatory enough without comments. But if you cant, here are some instances where commenting your code is still okay.

When Writing Complex Expression

Even for the best programmers, complicated regex or SQL codes can be difficult to express cleanly. In such a situation, adding a comment above the expression will help any developer checking out the code later to understand it without stress.

Legal Comments

There are certain that you are mandated to add to your program by corporate coding standards. This can be due to legal reasons such as copyright or authorship of the code. Such comments are necessary and should be added.

Warnings

Sometimes, you need to warn other developers that will be looking at your code about the possible consequences of an action to prevent them from breaking something. In such cases, it is okay to leave a comment close to this code. Such comments add value to your code and make it easy to avoid mysterious behavior in the code in the future. Apart from warnings, you may also use a comment to amplify the significance of a piece of code that may otherwise be overlooked by a user.

When All Else Fails

Although I recommend making your code as expressive as possible without comment when all else fails, and you are unable to write a code that is expressive on its own, feel free to write a comment. Do not leave poorly written code the way it is simply because you dont want to write a comment. You can add a comment to explain or clarify the intent of your code instead of leaving it as it is.

However, if you must write comments at all, make sure you make it local. A local comment is placed close to what it is referring to. A non-local comment, on the other hand, is far away from the code it is explaining and is bound to become a lie at some point.

Place comments that refer to a variable or function directly above it. A warning comment should be right beside or immediately above the code it is referencing. You may also make a warning comment stand out from the rest of your code if your IDE supports highlighting.

Additional Commenting Rules

Refactor Long Code

Typically, codes need clarification when their abstraction level is wrong. Bear in mind that functions should be small and should do only one thing. Any function that breaks this rule will most likely need comments to explain what it does. Rather than rush to add comments, you should rewrite the code and see if each piece of logic can be written as a separate function instead. If you do this right and use meaningful names, your code may not need a description, after all.

Writing each piece of logic as its own function enhances the readability of your code. Any developer looking at the code will most likely be able to tell what it does by simply reading the function name and get additional

information by looking at the implementation. It also improves the testability of your code since each function can be tested on its own more easily compared to when it was part of a larger chunk of code. Future changes to any part of the code will be easier as well

Commented Out Code

One of the most terrible practices when writing comments is commenting out codes. You should avoid this entirely. Not only does this mess up your code, but it will continue to pile up because new developers may not be willing to delete it since they are not unsure of why its there in the first place.

If you ever find commented out code in your work, dont wait and ask where it came from, just delete it. The longer it stays in your code, the messier things will become. Dont even try to uncomment the block of code. Just get rid of it.

TODO Comments

Some programmers argue that it is okay to leave TODO comment since it explains why a comment has a degenerate implementation. A TODO comment also tells what the future of that function might be. But I do not totally agree with the idea of writing TODO comments.

They are basically comments that explain things the developer should do but cannot do right now? But most of the time, most TODOs never get done. They will only become irrelevant after a while. Someone else looking at the code in the future will have a hard time figuring out if the task has been done or not. Hence, instead of writing TODO comments, it is best you just do them immediately if you can.

Chapter 6: Formatting

Youve probably noticed that you have a hard time reading some books compared to others. And it's not always because one has better content. Just like attractive handwriting will make it easier to read a letter, good formatting can make all the difference between whether you will read a book or drop it.

Writing code is very much like a book. No matter how great what you have written is, if the formatting isnt right and your code only hurts the eyes, anyone looking at it will have a hard time making sense of things. This is why your code should be well formatted. The indentation and spacing should be right and used consistently throughout your code and so on.

When people look at your code, how it is formatted will determine whether they will be impressed or not. The neatness of your code, the consistency, and how much attention you have paid to the details will affect the beauty of what they perceive. Using meaningful names and following all the other rules we have stated is something. But all of that wont matter if your code is a scrambled mess that looks tacky and unorganized.

When it comes to formatting code correctly, there are no specific standards to adhere to. The most important thing is to adhere to the same standard consistently throughout your code. Some programmers even come up with their preferred formatting standards. This is fine too, as long as you stick to that same standard everywhere in your code.

The right formatting will guide anyone reading through your code and makes it easier for them to understand. Even when you are working as a team, the team is expected to agree to the same formatting rules that every member must comply to.

What you will find in this chapter are mostly recommendations about how clean the code should look. We may not be able to tell you how many numbers of lines your code should have or the number of characters that should be in a line. These are things you will figure out as you write your code.

Why is Formatting Important?

As a developer, your first and perhaps most important job is to communicate. Most developers only focus on functionality and getting the code to work. If

you are concerned about writing clean code (which you should), then communication will be your priority over everything else. Your code will work, eventually, but you would have done a terrible work if the readability of your code is compromised in the process.

This is why correct formatting is important, especially if you intend to maintain the future extensibility of your code. If you get it right, your formatting is the one part of your code that will most likely stay the same long after most parts of your code has been changed in the future. And what will determine the ease of that change is how neat of a job you have done today.

Vertical Formatting

This refers to how your code is organized from top to bottom. It includes concepts such as vertical size, vertical openness, vertical density, distance, and order, among other concepts.

When it comes to formatting your code vertically, a common analogy programmers use to explain this is how a well-written newspaper article is organized. Any newspaper is composed of several articles. In terms of length, they are mostly small. Although some are still longer than others, only a few of them are ever over a page long. Keeping all the different bits organized this way is what makes a newspaper appear neat and readable. Imagine a newspaper that is just one long mass of story with all the facts and details jumbled together.

In every newspaper, information is organized in a way that makes everything concise and neat. Usually, at the top of the page, you will find the headline which tells you what the article is all about. Most times, the headline of an article is enough to tell you whether you want to read it further or not. This can be likened to the name of functions. It should be simple but still provide sufficient information about the code. After the headline, the first paragraph of any newspaper story serves as a synopsis of the entire story in summarized details. In the same way, the topmost parts of your code should contain all the high-level algorithms of the programs.

As you read down the newspaper article, subsequent paragraphs are expected to provide more details such as important names, dates, quotes, and so on down to the not so significant details. The same applies to your code. The

important functions should be at the top and are supposed to go down this way until you get to the functions at the lowest levels.

Vertical Openness

Codes are made up of multiple lines with each line representing a clause or an expression. As you go from top to bottom, each group of lines is a complete concept or thought. To make it easier to read your code, you should separate each of these concepts from the other with blank lines.

Take the code below; for example, the declaration, imports, and functions have white spaces in between them to separate.

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

This simple action has a tremendous effect on how the code reads and its overall visual appeal. Each blank space automatically tells you the next line is a new concept, different from the one above. The ideas are clearly shown, and you have no hard time reading through the code.

Now imagine all those blank lines gone. Without question, the readability of the code will be affected. It is still the same code (even smaller with the blank spaces gone), but you will need to pay close attention to make out individual concepts.

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

The concepts no longer pop out as they did in the first example. This goes to show the difference vertical openness can make in how your code reads and why this rule is important to keep your code neat and organized.

Vertical Density

While blank spaces should separate individual concepts, lines of code within the same idea should be as close to each other as possible. This is the idea of vertical density. Adding a blank space or even useless comments in between lines of code that are meant to be closely associated may affect the flow of the code and make it harder to read. Rather than space-related concepts out, there are better if they are closer to each other to improve comprehension.

Vertical Distance

It is not enough for your code to look neat, correct formatting can make it easier to figure out what the code does as well. One of the formatting features that affect how easy a reader can make sense of your code is the vertical distance.

If you have ever had to hunt through chains of inheritance and check from top to bottom to find the definition of a function or variable, youll appreciate the importance of having the correct vertical distance between related concepts. As a general rule of thumb, you should keep related content close. This will make it easier to figure out what a system does as you will spend time figuring out where the different pieces are.

Keep closely related concepts as close together as possible. They should be kept in the same file (unless you have a very good reason to keep them in separate files. The vertical distance between the two concepts is a measure of how important they are to each other. This will help the easier to find related

concepts faster and more conveniently.

Conceptual Affinity

Conceptual affinity simply refers to the idea that certain parts of code tend to want to be near each other. Conceptual affinity is one of the basis on which vertical distance can be determined while writing code. In essence, concepts with a stronger affinity for each other should be placed closer to each other. This affinity may be due to a direct dependency (e.g., when a function calls another or when a function calls a variable). conceptual affinity may also exist between a group of functions performing a similar operation. For instance, two functions with a common naming scheme that perform similar tasks or a variation of the same task should be placed proximate to each other even if they dont call each other.

Variable Declarations: As mentioned, it is expected that you declare variables as close to where they will be used as possible. Generally, you should place local variables at the top of every function where it will be used since functions are expected to be very short.

If you are writing control variables for a loop, it should be declared right within the loop statement and not elsewhere in the code. However, in some rare cases (for longer functions), variables may be declared just before a loop or at the top of a block of code.

Instance Variables: There is a long-standing debate about the correct positioning of instance variables. In a well-designed class, instance variables are used by most of the methods in the class. This is why they are best declared at the top of the class. The rules for formatting instance variables may vary from one language to the other. For instance, C++ programmers might place it at the bottom of the class, following what is known as the scissors rule. Java programmers, on the other hand, favor placing them above the class. Either of these conventions, depending on your preference. Just ensure that you declare the variable in a place that will be obvious. This way, anyone reading the code will know just where to look to find the declaration.

Dependent Function: If you have a function that is being called by another, they should be as vertically close to each other as much as possible. The function that is being called is below the one calling it. This helps to preserve the natural flow of your flow, ensuring the readability of your code. If they

more than one, the topmost function should call those below it, while those below should, in turn, call the functions that are below them.

Vertical Ordering

Just like the newspaper analogy explained earlier, it is expected that your code is arranged in a way that makes it easier to find information. The most important concept that explains what the code does should come first. They should be written as concisely as possible. The arrangement of concepts vertically like this is known as ordering

Vertical ordering makes it easy to skim through source files to get the most vital information first, while the low-level details come first. For instance, dependencies should point downward. This implies that the function above should be the caller.

Horizontal Formatting

In the same way that codes are read from top to down, they are read from left to right as well. Hence, formatting your code horizontally is just as important as vertical formatting. There are various aspects of horizontal formatting, which include the length of a line, horizontal openness, density, alignments, and so on.

Generally, as far as the length of individual lines of code is concerned, you should try to maintain your code to be a maximum of about 80 characters long. You may go above sometimes, but a figure about 120 characters is simply out of order. These days, the length of the line can be affected by factors such as the size of the screen or font size. But in all cases, you should maintain your code at 100 to 120 characters per line. Asides length of the line, other factors you should consider includes:

Horizontal Openness and Density

Like vertical openness, horizontal openness can be a great way to show the association between various parts of your code. At the same time, limiting openness is a perfect way to show weak relationships between various things withing your code. Here are some tips about openness that you should keep in mind when writing code:

- White spaces should follow assignment operators: this helps to accentuate the operator and two sides of the code, making the

separation obvious at first glance.

- No space between function name and the opening parenthesis that follows it: the function and the argument in parenthesis are closely associated. Hence, they are better written together. Spacing them will make the code appear disjointed.
- Separate arguments within the parenthesis using blank spaces: this helps to emphasize the comma, which shows that they are separate arguments.
- Showing precedence of the operator: when writing operators, you can use white spaces to show their precedence. For instance, operators with high precedence, such as factors, do not need to have any white space in between them. Similarly, addition and subtraction operators are low precedence. Hence you should have white space between them.

Horizontal Alignment

Horizontal alignment is good, but it does have some limitations and possible disadvantages. If not done correctly, alignment can draw the attention of the reader to an unnecessary part of the code and make the reader focus on the wrong part of the code.

For instance, if you chose to line up and align the variable names in your declarations or have all the values in an assign statement aligned, the reader may end up missing the good stuff; he/she will only read down the list of variable names that are aligned without actually looking at the types right beside it. The same applies to the values too. The reader may only focus on the values without seeing the assignment operators.

Instead of aligning code this way, it is best to leave the assignments and declarations unaligned. Unaligned assignments and declarations are not bad as long as the list is short. Rather than try to organize long code by aligning it, it may be best to split up the class entirely.

Indentation

There is a hierarchy of writing codes that must be followed and respected. In every code, blocks of code form methods, multiple methods form classes, and various classes form the entire file. The file is not a mere outline. It is made

up of various hierarchies, each of which can take name declarations and contains executable statements. These hierarchies tell the story of what your code does. Hence, the flow of your code is better if the ready can easily figure this hierarchy out on time. Indentations will make these hierarchies stand out, which is why they are important.

- One the file level, class declarations, and other statements are to be left unindented
- Methods inside a class are expected to be indented by one level to the right
- Indent the implementations within methods one level to the right of the method.
- For blocks of code withing another block, they should be indented one level to the right of the block that contains them.

The importance of indentations like this cannot be overemphasized. This is practically what makes your code readable at all. It is easy to see the structure of an indented file you can easily skim through and skip parts that are not relevant to what you are looking for. You know the new method declarations, variables, and classes are on the left. It also easier to find variables, accessors, constructors, and methods faster.

Work With Your Team

Every programmer has his/her own way of formatting code. If you are working alone, you are free to set your own rules and follow them. As long as you maintain consistency, your code should still be neat and organized. However, since most jobs require you to work as a team, the formatting style to be used by every member of the team should be set by the team. In a program with multiple authors, coordinating with the team and coming to an agreement on the right formatting style will ensure that your code remains consistent and organized.

Prior to coding, the team should decide on simple formatting rules like the indentation style, where braces would be used, how classes and variables will be named, among other things. To make things easier, these rules can then be encoded into the IDE to ensure that everyone complies with it.

Conclusion

Clean code is one that reads smoothly. To achieve this, the programmer must learn to write with a smooth and consistent style that will make it visually convenient for the reader to follow your code. Since any good software is made up of multiple documents, this formatting style should be maintained across all the documents to keep it the same at all times. This should be maintained whether you are working on your own or with a team of developers.

Chapter 7: Data Structures, Objects and the Law of Demeter

Bad codes dont happen by accidents. Codes dont just get messy with the wrong stroke of a key or any other unfortunate error of that sort. Usually, bad code is the product of the choices you make while writing codes. All the points and techniques mentioned so far in this book are essentially the choices you make while writing your code. Some choices are clear, cut, and open. It is easy to see why you should make them and why you shouldnt. Others arent quite as clear. In this chapter, we will discuss yet another important choice that you will most likely have to make while writing our code. The choice between objects and data structures. It is important to understand how each of these is used and when to use them.

Data Abstraction

Data abstraction is the process of reducing a particular body of data to a simplified version, which still represents the whole. The process itself involves removing certain characteristics of code and stripping it down to the only essential features.

This is one of the first and most important steps in designing any system or database. It is essentially a simplified specification of any entity. Abstraction is an important vital way to conceal information while coding. This way, only the relevant information is available to any programmer working on the code. This makes it possible to transform a complex data structure into a code that is simple and easier to use.

Any data type is made up of two things; properties and methods. The properties and methods can be either private or public. In abstraction, only public properties and methods are shown. They are expected to cover all the possible functionalities of the data.

Keeping variables private is important. You dont want someone else to depend on them. This privacy also comes with a level of freedom. You are free to change the type of the variable or how it is implemented when you keep them private.

Hiding the implementation of your variable isnt merely about adding a layer of functions between variables. It is all about expressing your data in abstract

terms. Many developers make the mistake of pushing their variables out with getters and setters. This is a common but bad practice. Exposing abstract interfaces that still allows the user to manipulate the essence of the data without revealing the implementation is a much better approach.

Objects Vs. Data structures

There is a significant difference between data structures and data. In objects, data is hidden behind abstractions while the functions that operate on this data is exposed. This makes it easy to add new objects to code without changing the already existing behaviors. Conversely, it is difficult to add new behaviors to an existing object.

Data structures, on the other hand, keep their data exposed, and they do not have any meaningful function. This is why they have no significant existing behavior. Hence, it is easy to add new behavior to data structure but difficult to a new data structure to an existing function.

From these definitions, the asymmetric nature of objects and data is quite obvious. Given the fact that these two are opposite, developers have to decide on which one to use in any given scenario. While many developers seem to be convinced that objects should be the main tool in software development, not everyone is convinced of that. So how do we make the distinction and decide on which one? Its all about understanding object-oriented coding and procedural coding.

Procedural Code and Object-Oriented Coding

While many programmers favor object-oriented coding, if you critically consider the nature of objects, it is easy to see why using data structures will be a better alternative in some scenarios. This is usually the case when you need structures that you can easily manipulate procedurally.

For instance, adding new functions to an object may be a little problematic. To do this, you may have to modify all the objects of the same type within the code. Of course, this isnt a problem you will experience in procedural coding.

In procedural coding, it is easy to add new functions without altering the data structure already in place. For object-oriented coding, you can easily add new classes to the code while leaving the existing functions in place.

Adding new data structures is hard in procedural coding. You will have to change all the functions to achieve this. Similarly, adding new functions is difficult in object-oriented coding because you have to change all the classes.

So how do you decide when to choose between object implementation or procedural coding. Here is a simple rule to follow.

1. If you expect to add more functions to your code with time while retaining an unmodified data structure, it is recommended that you choose procedural coding.
2. If you are expecting to add new types to your code in the future, but you dont intend to add any new functions later, then you should go with the object-oriented approach.

Law of Demeter

As earlier explained, an object is expected to keep data hidden while exposing functions that operate on this data. However, some programmers may make the mistake of exposing the internal structure of their data through the use of accessors. When they do this, they are breaking a well-known law known as the law of Demeter, and this is bad practice.

This law states that a module should not know the innards of an object that it manipulates. According to the law of Demeter:

A function f of class C should only call the methods of C, an object created by f, an object pass as an argument to f, or an object held in an instance variable of C. The function should not invoke methods on objects that are returned by any of the allowed functions Talk to friends, not strangers.

To put this law in simpler terms, imagine you have an object A which requests the service (calls a method) of an object B. This is perfectly fine. However, the object, A, is not allowed to reach through the object, B, to request the services (call a method) of another object, C. This is because reaching through B to get to C this way will require object A to have a greater knowledge of the internal structure of the object B.

Rather than do this, the law says that you can either modify the interface of object B to serve the request of object A directly or have the object A make a direct reference to C to make its request. By following this law, the internal structure of object B will only be known to that object.

Take the example below, for instance:

```
let outputDir: String = context.getOptions().getScratchDir().getAbsolutePath()
```

The code calls getScratchDir()to get the return value of the function getOptions()before calling the function getAbsolutePath().the law of Demeter is being broken already. To make your code cleaner, you may decide to break up the code like this.

```
let options: Options = context.getOptions()

let scratchDir: File = options.getScratchDir()

let outputDir: String = scratchDir.getAbsolutePath()
```

Even in this state, the law is being broken (sort of). This is because the containing module now knows that options are contained within the function context. Inside options, there is a scratch directory, which has an absolute path. The function already knows too much about the object it calls (if context, Options, and ScratchDir are objects). If they are data structure, then this code is absolutely fine since there is no data to expose. But if they are objects, then the internal structure which should be hidden is already exposed.

The code above can be modified this way:

```
let bufferedOutputStream: BufferedOutputStream = context.createScratchFileStream(classFileName).
```

Dont Create Hybrids

As an additional rule, when working with objects or data structures, you should avoid creating hybrids. Hybrids are extremely difficult to work with. They are partly objects and partly data structures. This introduces several unique problems that are difficult to deal with.

Hybrids have functions that have no significant purpose. They also have public variables or public accessors and mutators. This not only makes it difficult to add new functions but adding new data structures is made harder as well.

Chapter 8: Error Handling

Ever heard the saying, if anything can go wrong, it probably will? Thats Murphys law, and its a reality almost every programmer can relate to. Error handling is an inevitable part of programming. Devices fail, and inputs may be abnormal sometimes. Knowing that theres a lot that can go wrong when a program runs, programmers have the responsibility of ensuring that the code they write does what it has been designed to do.

But how is this connected to writing clean code? Since error handling is an inevitable part of programming, it only makes sense that you learn the proper way to handle them. Programmers who have not mastered error handling the clean coding way may end up with programs that have error-handling code scattered all over them so much that it makes it difficult to figure out the logic of the code. This is a way we will be going over the form of the best clean code techniques that help you handle errors neatly.

Use Exceptions Instead of Return Codes

There was a time where most programming languages did not have exceptions. Back then, to handle the error, you would either have to return an error code in which the caller would check or set error flags in your code. While this worked, it was limited in several respects. Handling error this way also led to a lot of clutter in the caller. The calling function would have to check the code for error after each call. This is quite easy to forget.

These days, a better and more cleaner alternative is to use exceptions. By making the caller throw and error whenever an error is encountered, you can write cleaning call functions whose logic is not cluttered by how it handles the error.

De-cluttering functions this way offers several advantages, and they are not just about the beauty of our code. The logic is also easier to follow when the errors are better highlighted, and error handling is separated with exceptions.

Write Try-Catch-Finally Statements First

When writing codes that are likely to throw an exception, it is best, to begin with, a try-catch-final statement. Following this rule helps you to easily define what would happen no matter what error occurs during the execution of the code. Try blocks are similar to transactions and should be treated like

that.

You should write them in such a way that the catch always leaves your code in a consistent state despite what happens in the try statement. But when you execute code within the try part of your try-catch-finally statement, you are indirectly stating that the code execution may stop at any point only to resume later at the catch.

When writing codes, try to write tests that will force exceptions. Subsequently, you can add behavior to the handler that will satisfy the test you have written. This will make it easier to build the try blocks transaction scope first then build up the rest of the logic with TDD. The logic should be added in such a way that it can pretend that nothing goes wrong in the code.

Use Unchecked Exceptions

There was a time when checked exceptions made a lot of sense. In the early days of java, every method would have a list of all of its possible exceptions that could be passed to its caller in its signature. These exceptions were also included in the type of the method such that the code wouldnt compile if the exceptions in the signature do not match those in the code. While this is great and offers some unique benefits, many programmers are starting to see that it is not necessary to build a robust software

In fact, most new programming languages dont have checked exceptions at all. C# and C++ do not have checked exceptions. Similarly, Python and Ruby do not have it either. The fact that one can still build robust software in these languages using unchecked exceptions puts to rest the argument about the relevance of checked exceptions.

The Price of Using Checked Exceptions

Although checked exceptions offer some benefits, many programmers are now coming to terms with the fact that it comes at a high price. Using checked exceptions violate an Open/Closed principle. Check exceptions may not be a problem if the block throwing the code is right below the block catching it. But this is rarely the case. In most instances, you may have the block catching the exception several levels above. In this case, you will have to declare the exception in the signature for each method in between the throw and the catch block for the code to compile.

Checked exceptions break encapsulation. This is a major problem if you

consider the fact that most large codes are made up of complex hierarchical systems. Functions at the top of the code call the ones below, and those ones call even more functions at lower levels. Thus, if a low-level function is modified to throw a checked exception, your code will fail to compile until you have added a throw clause to the signature of the function. You will also have to modify every function above to either catch the exception or add the throw close to their signatures. These changes must be made throughout your code from the lowest level, where the exception was thrown to the highest.

If you are building general applications, you are better off using unchecked exceptions than checked. The cost simply outweighs the benefit. Checked exceptions may only be beneficial if you are building a critical library.

Provide Context with Exceptions

When you throw exceptions, you will make life easier for anyone reading your code if you provide some context about the source of an error. Hence, you should create error messages that are informative and provide context to your exceptions, which you can pass along with them. In Java, a stack trace does this partly by making it possible to trace the source of an error. But it does not tell you the intent of error. This is why you should add error messages that mention the type of failure, where it failed, and why it did. Passing along sufficient information like this will make error logging in the catch a lot easier.

Using the Special Case Pattern

By following all the error handling instructions highlighted so far, you should be able to separate the logic of your code from the error handling. The external APIs will be wrapped to allow you to throw your exception. Then to deal with aborted computation, you define a handler above the code. This will make your code look neat and organized.

However, doing this also pushes error detection to the edge of your program, and this is not always a good thing. There are times where you may not want to abort your code. In such cases, the highlighted approach may not work. A better alternative would be to use what is known as a special case pattern. This involves creating a class or configuring an object which handles special cases for you. When you take this approach, the client code does not need to deal with exceptional behavior anymore. Instead, the behavior will be

encapsulated within the special case object.

Consider the example below:

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

We have a code that is expected to sum up the expenses in a billing app. According to the code, if the meals are expensed, it is added to the total, but if they arent, the code returns a meal per diem. This exception clutters the logic of your code. The code will look a lot cleaner if this special case is avoided entirely. It will make the code much simpler.

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());

m_total += expenses.getTotal();
```

We can modify the code such that the ExpenseReportDAO will always return a MealExpense object. This MealExpense object then returns the per diem total if the meals are not expensed. The code will now look like this:

```
public class PerDiemMealExpenses implements MealExpenses {

 public int getTotal() {

 // return the per diem default

 }

}
```

While discussing error handling, we must mention some of the subtle ways that we may inadvertently invite errors into our code. One of such seemingly normal actions that may invite error is returning null. This is a fairly common practice for some programmers.

Putting in a null reference can lead to several errors, system crashes, and vulnerabilities in your code. This will make error handling a lot more difficult both now and in the next few years when someone else is taking a look at your code.

A lot of programmers use null to indicate that a return value is absent. While this seems like normal standard practice. It isnt as good as it seems. In fact, there are other better alternatives to returning a null that will make your code cleaner but more importantly make it less error-prone

When writing code, you can assign a null value to any object reference to indicate that it points to nothing. When such a method with a null reference is called, a NullPointerException will be thrown, as in the example below.

```
Blog blog = null;
long id = blog.getId(); // NullPointerException
```

In this case, any static that is uninitialized and the instance member of the reference type will return a null pointer exception. The example below illustrates this better.

```
public class Blog {
   private long id; // 0
   private String name; // null
   private List<Article> articles; // null

   public long getId() {
      return id;
   }

   public String getName() {
      return name;
   }

   public List<Article> getArticles() {
      return articles;
   }
}
```

If a constructor is absent, then the methods wgetArticles() and getName() will always return a null reference. But there is a tendency for your code to become littered with null clauses like this when you repeatedly write methods that return null all over the code.

```
if (blog.getName() != null) {
    int nameLength = blog.getName().count();
}

if (blog.getArticles() != null) {
    int articleCount = blog.getArticles().size();
}
```

This is annoying, difficult to read, and susceptible to run-time exceptions if anyone misses a null check anywhere.

Heres another example:

```
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);
}
}
}
}
```

Almost every line on this code has a check for null. Although returning null isnt a problem on its own, you are creating extra work for yourself and even more issues for your call functions when you return null this way. Everything may seem fine until a missing null check sends your application way out of balance. So what do you do instead of calling null, here are some examples:

Return an Empty Collection

If the method is meant to return a collection class, we can substitute the null exemption for an empty collection. Typically, there is a static method for this purpose in the collection class.

return Collections.emptyList();

Or

return List.of();

In both instances, an immutable list will be returned such that the calling code will not attempt to make any modifications.

Return an Optional Object

Another option is to use a class known as java.util.Optional, which was designed to solve some of the problems with null values specifically. An optional object class is a container that can either contain a non-null value or be empty. This serves as a limited mechanism for library method return types in instances where using null to represent no-result was likely to cause errors to occur.

The class Optional.empty() can be used to represent an empty container while Optional.of(value) can be used if there is an actual value to be represented, as seen in the example below.

```
public Optional<Blog> getBlog(long id) {
   // Blog not found
   if (!found) {
      return Optional.empty();
   }

   // Blog found
   return Optional.of(blog);
}
```

Subsequently, the if Present() method can be used to retrieve the value if it exists, as shown below.

```
Optional blog = getBlog(100);
blog.ifPresent(System.out::println);
```

In this case, the null check has become abstracted away and is now enforced by the type system. As with the example above, if the optional object is empty, then nothing will be printed.

Return a Null Object

The Null Object pattern was designed as a way to identify expected behavior when a null is encountered and encapsulate it with a static constant. Expanding the previous Blog class code above, we have

```
public class Blog {
   public static Blog NOT_FOUND = new Blog(0, "Blog Not Found", Collections.emptyList());

   private long id;
   private String name;
   private List<Article> articles;

   public Blog (long id, String name, List<Article> articles) {
      this.id = id;
      this.name = name;
      this.articles = articles;
   }

   public long getId() {
      return id;
   }

   public List<Article> getArticles() {
      return articles;
   }
}
```

In line 2 of this code, a constant has been declared, which represents the null object. In this case, it is a Blog object with assumed sensible value for a blog. Any method which returns Blog type will now use Blog.NOT_FOUND rather than return null.

Throw an Exception

Many programmers are scared of throwing exceptions. It is not unlikely that this fear of throwing exceptions stems from an expectation to validate user input and recovering them elegantly if the data passed was invalid. If not handled right and one of the values being used is in a null state, this kind of code ma pollute your code logic and leave with a host of illogical codes.

However, it is important to note that throwing an exception is normal practice when you have exceptional circumstances. If you always expect to find values, then it makes perfect sense to throw any exceptions. Basically, the decision to throw an exception if there is nothing to return depends on the nature of your application. For instance, you may throw an unchecked exception if the ID that is passed to your method does not exist in our database.

```
// service1 = null;
// service2 = null;

...

var total = 0;

if (service1 != null)
{
    total = total + service1.performCalculation();
}

if (service2 != null)
{
    total = total * service2.performCalculation();
}

return total;
```

You can easily substitute the code above with something like the code below using proper exception handling at higher levels.

```
...

var total = 0;
total = total + service1.performCalculation();
total = total * service2.performCalculation();
return total;
```

To wrap up this chapter, bear in mind that clean code is not only readable; it should also be robust. You may not be able to achieve either of these goals if we combine error handling with the main logic of our code. Instead, it should be considered independently and handled so. How much you can do this will go a long way to determine how maintainable your code will be in the future.

Chapter 9: Understanding Boundaries

Sometimes in programming, we make use of third party codes or open-source software to add more functionality to a program and reduce the time spent on coding. In some cases, it could be components built by other teams within the same company. While maintaining clean coding principles for your own program is important, you must also learn techniques for integrating foreign codes into your system without compromising the organization and integrity of your code. This is why learning about the concept of boundaries is important. In this chapter, we will go over some of the standard clean coding practices that will help keep your software clean at the boundary between external pieces of code and third-party software.

What Are the Boundaries?

Boundaries are points withing your code where it meets code that has been written by others. No matter how well-written your code is and how much you have expended to keep it organized, the fact is that a boundary is an unpredictable area. You have little or no control over how this part of your code will behave especially if you are not entirely sure of what the third party code you are adding does.

Why are Boundaries so Important?

Usually, third party packages are built in a way that makes it easy to plug them into multiple frameworks. While this is great, this multi-applicability introduces a range of new problems to you as a programmer when you consider how your own users will use the framework you intend to build. It isnt uncommon for third party packages to have plenty of hidden capabilities that could be a liability to you in your code if not handled properly. Sure it will serve its intended purpose, but such liabilities can also cause problems in your own code later on.

Generally, it is recommended that you avoid passing a third party object around within your own system. Do not return a function from it or accept it as an argument to your public APIs. If you will be using a boundary interface, it is best wrapped within a class or inside a close family of classes where your system will make use of it.

Handling Change at Boundaries

Another thing a software with a clean code should be able to do well is adapt to changes in the external systems that you have no direct control over. This is best done at the boundaries of your code with such external codes.

Most times, changes in third party programs are done with little consideration about how the integrating systems will be affected based on how the program is being used. The responsibility is on you as the developer to write your code in a way that it adapts to such changes neatly.

At the boundaries of your code, there should be clear separations and unit tests that make it easy to define what is expected of your code. It is recommended that you buffer code boundaries using adapters. An adapter will help concert your own interface to the provided interface of neatly. This way, your dependency on external software is not too widespread. Thus, when changes occur in the external codes, you have very little changes to make within your own code.

Limiting dependency this way is essential since you have little or no control over the external code. This can be achieved by having very few places in your code that make reference to them. Future maintenance of your code will be easier this way, and you will also be able to better adapt to changes in the external subsystems.

Learning Boundaries

Another issue with using external code is that it typically has a steep learning curve. Lets say you want to utilize a third-party package that you are not sure of how to use. Then you have to spend days reading through the documentation to get an idea of how it works and the right way to use it.

But this doesnt even guarantee that your code will work as designed. You may write the code that uses this third party component only to run into bugs and issues when you attempt to integrate both systems. Now you have to spend hours or days trying to figure out if the bug is from your own system, the external system, or how you are implementing the code.

Using this approach, learning and integrating boundaries can be very difficult work. A much better approach is to write tests that explore your understanding of such codes based on its intended usage. Such tests are referred to as learning tests.

What are Learning Tests?

To explore how much you understand a third-party code, you can set up a learning test. This helps to save time that would have otherwise be spent on experimenting on the production code. With a learning test, you can call a third party API the same way you expect it to work in your application. This controlled experiment will help check how much you understand the API without spending so much time on studying the boundary documentation.

For instance, if you have a new third party code to test, after downloading and opening the software, you can write a test case based on the intended use of the API to quickly test it without doing too much reading on the documentation page. When you run your test code, if it produces an error, you can read to the documentation further to figure out the error then create another test to check your knowledge.

Learning tests help you to learn a great deal about how third party code works, and you can easily encode what you have learned into simple unit tests. This is a simple and isolated way to gain knowledge, and it costs nothing. It is a series of precise experiments that boosts your understanding of third party code for free and see if it does what you expect it to do.

Learning tests is also important in handling new releases of third party packages that you have been using before. When you install an update of third-party code, you can run the learning tests to see if there are any behavioral changes in the code before integrating it into your program. Once integrated, it may be difficult to see if a third party code is still compatible with your program based on what you need it to do. Authors of third-party packages typically make changes and fix bugs based on their own needs. These changes introduce a lot of new risks to codes making use of such packages unless you figure out a neater way to tests the compatibility.

How to Use Code That Does Not Exist Yet

While coding, we cannot assume that we know everything that there is to know. A large project typically consists of various systems and subsystems that integrate and interact with each other. In many cases, these systems are built by various teams whose work might not be clearly known to the other. The result is that you get boundaries whose other side is unknown. Such a boundary with the unknown is difficult to handle cleanly.

For instance, consider a team developing software for a communication

system, one of the subsystems that will be needed for the code to work is a Transmitter and the team responsible for building this transmitter had not defined the interface of their work yet. In order not to hold up the project, the main team can proceed with the main system without knowing how the transmitter code would work.

They simply need to be aware of what they want the unknown code to be to their own systems. This will help define a clear boundary. Even if you have no idea how the transmitter would work or what the interface would look like, you can define your interface and decide the things you expect it to do within your code. You may give your interface a catchy name that clearly defines what it does in relation to the expected API, and create a method that serves its purpose.

By designing the interface, you hope to ensure that a large part of the unknown is still under your control. This not only ensures the readability of your code but also helps to keep it focused on what you intend to have it do.

Also, since your transmitter API, which has not been built, is out of your control, you should insulate the classes of your own code from this API. Once the transmitter is defined, you can then proceed to write an Adapter, which will serve as a bridge at the code boundary. The adapter will encapsulate the interaction with the API at the boundary and keeping with what we have discussed earlier; it also provides a single point of change if the API is modified in the future. Finally, you can also set up a boundary learning test code for the API. This ensures that the API is being used correctly in your code.

Chapter 10: Writing Clean Unit Tests

Unit tests have grown in relevance since the early days of Test-Driven Development. Before TDD, unit tests were nothing but short bits of code that were written to confirm if the program worked as intended. It was typically some dispensable ad hoc code that would be written after the classes and methods and other parts of the code had been painstakingly put together.

The unit test is a way to interact with the program after it has been written manually. These days, thanks to the TDD and Agile development approach, writing unit tests have become the norm. Programmers now make it a point of duty to write tests to ensure that every part of their code performs as expected.

Typically, bits of code will be isolated from the operating system and set up tests for them. Once the code passes a suite of tests, it is the responsibility of the programmer to ensure that anyone else who would be working with the code in the future can run those tests conveniently as well. Both the tests and the code will then be checked into the same source package.

However, while writing unit tests have become quite popular, not all programmers are doing it right. In a rush to add tests to code, it isnt uncommon to find tests that are missing the points of writing good tests.

What Makes a Good Unit Test?

Before diving into some of the rules and principles for writing good unit tests, we must identify and discuss some of the properties of good tests.

Easy to Write: as a developer, you will typically have to write a lot of tests that cover all the different parts and cases of your applications behavior. Hence, it should be easy to write such test routines without expending too many efforts.

Readable: clean code should always be readable, and this applies to unit tests as well. The intent of the test must be clear. Like your product code, the unit test is expected to tell the story of how a part of your application behaves. The scenario being tested should be easy to understand. Part of readability is also being able to detect and address the problem if a test fails easily. A good unit test should make it easy to fix a bug without necessarily debugging the entire code.

Reliable: a unit test is expected to fail only if there is a bug in the system that is being tested and not otherwise. While this is a simple and obvious rule, it isnt uncommon to find programmers that right tests that fail even where there is no bug in the system. For instance, some tests may pass when they are run individually but fail when the whole test suit is run. There are also cases where tests will pass on the IDE only to fail on the integration server. Such instances are indicative of a flaw in test design. A good unit test is expected to be reproducible despite changes in external factors like the running order or development environment.

Fast: developers are expected to write tests that allow them to check for bugs repeatedly. If such tests are slow, a developer working on them will likely prefer to skip them to save time. Otherwise, you lose precious time trying to run the tests. However, slow tests are not always a fault with the tests itself. Sometimes, it just means that the system under test interacts with the external system in a way that slows it down. This is not good as well as it implies that the test may be environment-dependent.

It Must be Truly Unit: unit tests are not integration tests. This should be clear from the design of the test. A unit test and the system it is used to test are not expected to have access to the network resources, file systems, databases, and so on.

Why Should You Write a Good Unit Test?

Perhaps the most important rule for writing good unit tests is to treat your test codes just as good as your production code. Unit tests are just as important as production code. Most developers make the mistake of treating test codes as second-class citizens. Most programmers favor the Quick and dirty approach to save time on writing unit tests and get more done. This is, in fact, bad practice. Unit tests require just as much care in thought and design. You must keep it just as clean as you keep your production code.

Of course, this is not as easy as it sounds. Writing a clean unit test code is difficult in practice. But you must uphold the principles of clean tests. This is because writing dirty tests is just as terrible (if not worse) as writing no test at all. As your production code evolves, your test code is expected to change as well. If your test code is dirty, making such changes will become increasingly difficult, and, as the production code gets new modifications, the test code will become more tangled. To keep up, you might have to cram new tests into

the suit to get the tests to pass. This will make the code messier even messier until it becomes a complete liability.

The consequence of not keeping your tests clean is that you will eventually lose them. As the test code becomes more tangled, you may have to let them go entirely. When you do this, you also lose the very thing that helps to keep your code flexible, reusable, and maintainable. With tests, making changes to code is easy, but without them, you cant make changes to your code without expecting something to break somewhere no matter how flexible the design of your code is. Dirty test hamper your ability to improve your code, and this eventually leads to rot.

How to Write Clean Unit Tests

Writing good unit tests is difficult, but as we have pointed out, you must learn how to write them and so do cleanly or risk losing your entire code. Here are some tips that can guide you in writing simple, sharp, and clean unit tests at all times:

1. **Dont Put Multiple Tests Into the Same Function:** Lumping multiple tests into the same function is not good practice. Take the code below, for instance; three different tests are cramped into the same function. This not only affects readability but can potentially affect how the tests run as well.

```
public void testBasicStuff() {
  Bar bar = makeBar();
  Bar expectedBar = makeBar().setBuzzer(3);
  runBuzzProgram(bar);
  assertThat(bar).isEqualTo(expectedBar);
  Bar expectedBar2 = makeBar().setBuzzer(6);
  runBuzzProgram(bar);
  assertThat(bar).isEqualTo(expectedBar2);
  Bar expectedBarFinal = makeBar().setBuzzer(0);
  runBuzzShutdownProgram(bar);
```

```
  assertThat(bar).isEqualTo(expectedBarFinal);
}
```

From the test code above, it is unclear how Test one will affect tests two and three. It is also unclear what the starting states of the second and third tests are since they are no explicit. For example, the starting state in code two depends on whatever the state is after the test one finishes. This makes it harder to grasp what each test does explicitly.

To fix these problems, it is best to set up the second and third tests separately in two different functions, as shown below.

```
public void testOne() {
  Bar bar = makeBar();
  Bar expectedBar = makeBar().setBuzzer(3);
  runBuzzProgram(bar);
  assertThat(bar).isEqualTo(expectedBar);
}public void testTwo() {
  // Here, we set up the same state that test #1 ended in.
  Bar bar = makeBar().setValue(3);
  Bar expectedBar2 = makeBar().setBuzzer(6);
  runBuzzProgram(bar);
  assertThat(bar).isEqualTo(expectedBar2);
}public void testThree() {
  // Here, we set up the same state that test #2 ended in.
  Bar bar = makeBar().setValue(6);
  Bar expectedBarFinal = makeBar().setBuzzer(0);
  runBuzzShutdownProgram(bar);
  assertThat(bar).isEqualTo(expectedBarFinal);
}
```

After separating the tests each one will only take up about 4 lines of code. By

making the starting state of each test clear, readability is enhanced and it is easier to understand the concept of each test.

2. **Use Good Test Names:** the importance of using good names in every code you write cannot be overemphasized. This holds for writing test codes as well. As a general rule of thumb, you can follow the structure below to name tests correctly.

public void testSYSTEM_BEHAVIOR_CONDITION() {

Following this format, you can have your test code to be something like this:

public void testAuthenticate_rejectsUser_whenBadCredentialsProvided() {

Take a look at the test name below.

public void testBasicFunctionality() {

Its a shame that some developers still use names like this. When the name isnt explanatory enough, anyone looking at the code will have to read the test implementation to know what exactly the code is testing. This is bad for the readability of our code. We could have used a more meaningful name like the ones below.

public void testMyFunc_updatesBar_inExpectedGeneralCase() {

...

}public void testMyFunc_updatesBar_whenFooServiceReturnsError() {

...

}

3. **AAA (**Arrange, Act, Assert): this is a simple pattern that has become standard for writing clean unit tests. This pattern suggests that you break down your test methods into three sections.

- **Arrange**: Set up tests. At this stage, you should only write code that you need to set up a specific test. This is the stage where objects and mock setups are created, and potential expectations of the test are setup.
- **Act**: the act stage is where the invocation of the method being

tested takes place. Here you run the function under test.

- **Assert**: here, you check if the expected behavior occurred or not.

Following the pattern above ensures that it is well structured and easier to read.

Take a look at the example below:

```
public void testMyFunc_updatesBar() {
  Bar bar;
  Bar expectedBar = makeBar().setBuzzer(3);
  myFunc(bar);
  assertThat(bar).isEqualTo(expectedBar);
}
```

The code is a bit confusing despite being just four lines because the AAA pattern is not followed and separated. The purpose of expectedBar declared at the top is not known. It is unclear if it will be used in the myFunc() or not. It is also difficult to tell when the setup finishes and when the tests start to run.

We can separate these three steps in the code above to clear up all the confusion. Now we have:

```
public void testMyFunc_updatesBar() {
  Bar bar; myFunc(bar); Bar expectedBar = makeBar().setBuzzer(3);
  assertThat(bar).isEqualTo(expectedBar);
}
```

Following this simple pattern, no major debugging or refactoring is required to implement the code. This structure affects the readability and maintainability of your test suits and makes it easier to format the test units in the present and much later in the future.

Chapter 11: Classes

Writing clean classes is one of the essential clean coding techniques to learn. Paying attention to your classes and how they are organized is just as important as writing single lines and blocks of codes. This is a higher level of code organization that you must pay attention to keep your code orderly and expressive.

Class Organization

There are standard rules and conventions for organizing classes while coding. Following these conventions will help keep your code clean. Here are some simple rules of class organizations to follow.

1. You should begin your class with a list of variables
2. When writing classes, you should write the public static constants first. This should then be followed by the private static variables then private instance variables in that order
3. Write your public functions after your list of variables
4. The private utilities called by a public function should follow the public function that calls it immediately.

Following these conventions ensure that your code flows neatly like a newspaper article in an orderly manner.

Encapsulation

It is recommended that you keep your utility functions and variables encapsulated. However, this is not a rule set in stone. There are cases where you need to make variables protected to allow tests to assess them. But even when a test in the same package needs to access a variable to call a function, you should first look for ways to maintain privacy and only loosen encapsulation when it is the only option available.

Keep Classes Small

Just like with functions, the basic rule when designing classes is to keep them as small as possible. Focus on writing smaller classes, and even when you have a small one, you should still check to see if it can be simplified further or that is the smallest form possible.

It is hard to define how small a class should be clearly. For functions, we can use direct line class to estimate how small it should be. But for classes, the rules arent that straightforward.

Single Responsibility

A class should have only one responsibility. It does not matter the number of public methods it contains. If it has more than one responsibility, then you should figure out ways to make it smaller.

One way to find out if the class has too much responsibility is how it is named. If you find it difficult to give a concise name to a class, then it is most likely too large, and you should do it. An ambiguous class name is one of the obvious signs of a class with too many responsibilities.

In naming classes, try to keep things concise. Beware of ambiguous words like Super, Processor, Manager, and so on. Words like that suggest that you have aggregated too many responsibilities unto your class.

Another way to tell if your class is too large is how you describe it. You should be able to write a description for the class in 25 words or less. Also, when you use terms like and, or but or if to describe your class, there is a good chance that it is doing more than it should already.

The point above aligns with the Single Responsibility Principle, which states that a class or module should have only one reason to change (one responsibility). Trying to see if your code has more than one reason to change is a smart way to recognize if it is possible to create better abstractions from the code.

Single Responsibility rule is one of the simplest concepts in object-oriented programming. Yet, it is one of the most abused rules in class design. Most developers break this rule because they only focus on getting the software to work at the expense of writing clean code. However, it is important to always bear in mind that writing a program that works is only one part of the task. The organization and the cleanliness of your code is important as well. Hence, even after writing code that performs as expected, you should always look it over to see if you can make your code perform the same function without writing overstuffed classes.

You may argue that having too many small classes will give your system too many moving parts, which will make it difficult to figure out. But this isnt a

problem in itself. Having many small parts is not an issue as long as each one is organized. It is a much better alternative to having a few large classes that do too many things. In the long run, this is just as terrible. Complexity is an inevitable part of any large system. Your work as a developer is to look manage such complexities in a way that anyone looking at it can understand what is going on in the shortest time possible.

Cohesion

When writing classes, it advisable that the cohesion between your classes is high. High cohesion implies that methods and variables within that class are co-dependent. This ensures that they all bunch together to form a logical whole.

In every class, you should maintain a very small number of instance variables. This way, each method in the class will manipulate at least one variable. As a general rule of the thumb, the cohesiveness of a method increases in class based on the number of variables that it manipulates. For instance, if you have a class in which each variable it contains is being used by a method, such a class can be said to be maximally cohesive. However, it is virtually impossible to create a maximally cohesive class. Thus, you should ensure that your classes have high cohesion.

In an attempt to keep functions small and shorten the parameter list, one common problem may increase the number of instance variables that are being used by the subset of methods. The implication of this is that at least one other class that there is at least one class that is trying to get out of the class. To solve this problem, you can try to separate methods and variables into two or more classes to form a new, more cohesive class.

Maintaining Cohesion Results in Many Small Classes

When you break large functions into smaller ones to maintain cohesion, you can expect the number of classes to increase as well. For instance, lets assume that you have a function that contains several variables declared within it. You decide to extract a small part of the function to form another function. If there are four variables within the function and the code uses all four of them, you do not need to pass all four variables as arguments into the new function you are creating.

Instead, you can promote the variables to instance variables of the same class,

then extract the code without passing any of the variables. This will make it easier to break the function as intended. But taking this action will cause the classes to lose cohesion since they now contain more instance variables that are being shared by a few functions. This forms a new class entirely. This action has given birth to a new rule: when your classes lose cohesion, it is okay to split them. Your program will be better organized with a much more transparent structure when you split them into smaller classes like this.

Organizing for Change

Change is an inevitable part of any system. But with every change made in a system, there is a risk that something breaks elsewhere in the system. This is, in fact, one of the main goals of writing clean code, i.e., to put together an organized system where the associated risk of changing anything is minimal.

For example, consider an SQL class that was designed to generate properly formed SQL strings when appropriate metadata is provided. Since the code is still a work in progress, it does not support SQL functionality, such as update statements yet. At some point, you will have to open up the class to make modifications that will allow the SQL class to support update statements. A modification like this comes with an associated risk of breaking the code

There is another problem. The SQL class violates the single responsibility principle earlier explained. The class changes when a new type of statement is added or when we change the details of the statement type. Looking at the code from an organizational standpoint, it is easy to see the violation in SRP. There are private methods within the SQL class based on the method outline.

The presence of this private method is an indicator that there are potential areas where the code can be improved. However, a major motivation for taking action is how the system changes itself. If the class is logically complete and no update in functionality will be required in the future, then there is no need to separate the responsibilities into different classes. But if we will be performing any modifications like opening a class, then the design has to be fixed, and the class refactored so that each method now has its derivative.

This simplifies the code to the lowest form possible. By implication, it is easy to understand what the code does. We also dont have to worry about breaking another part of our code any longer. It also easier to write tests for this code

since each class is now isolated from the other.

Finally, when it is time to write update statements for the SQL class, there is no change in the existing classes. A new update statement can be built in a new subclass. This ensures that the change doesnt break anything within the class. This modification is in line with a major object-oriented class design principle known as the Open-closed principle. This principle states that classes should be kept open for extension but closed for modification. This is what is expected in an ideal system. New features are added by extension and not by modification of existing code.

Chapter 12: Concurrency

If you have some coding experience, then you would probably know how difficult writing a clean concurrent program can be. It is easier to write single-threaded codes. But multithreaded codes are difficult to write.

Most multi-threaded codes only look good on the surface. But such codes rarely run as intended under stress. Still, there is a need to write concurrent programs that run seamlessly. This means every programmer concerned about writing clean code must learn ways of dealing with the difficulties associated with concurrent programming. Lets begin by defining concurrency and why it is important.

What is Concurrency? And Why is it Important?

Concurrency is a strategy used for decoupling code. With single-threaded code, the what and when of the application are so strongly coupled that it is possible to determine what the application does by simply looking at the stack back-trace. Debugging such a code is easier as well. All a developer looking at the code needs to do is to set a breakpoint. The state of the system will be determined when any of the breakpoints are hit.

Decoupling a multithreaded code helps to improve the structures and throughput of your programs. When you decouple, your code no longer looks like one giant loop. Instead, it looks more like a network of collaborating computers. This makes it easier to understand the entire system. It also makes it easier to separate concerns.

A common example of decoupling in an application is the use of the Servlet model of web apps. In these systems, concurrency is handled partially by Web or EJB containers. When web requests come into the system, the servlets are executed asynchronously. Each servlets execution is decoupled from that of others so that the programmer does not need to manage all the incoming requests. However, this process of decoupling isnt as easy as it sounds. The type of decoupling provided by web containers still have their limitations, but the structural benefits of this model are too much to be ignored.

In addition to solving structural problems, concurrency solves the problem of response time and throughput constraints in systems as well. For example, imagine a program that aggregates information from different websites and

merge it into a daily summary. If such a program is single-threaded, it has to hit each website on after the other to gather the required information in less than 24 hours. Even if this isnt a lot of work initially, as more websites are added for the aggregator to check, more time will be required to gather data from all the sites since the single thread code will have to do a lot of waiting at the web sockets. It is easy to see that a multi-threaded code that runs concurrently will be better for this task as more websites can be covered within a shorter time.

What Concurrency Cannot Do

While there are several reasons a multi-threaded concurrent code will work better for a complex system, it is important to understand the limitations of concurrency before you opt to adopt it. Not knowing these limitations can lead to some unexpected behavior in your code.

First, it is important to note that concurrency doesnt always improve the performance of a system. While it does this sometimes (if there is a lot of wait time that can be better handled by multiple processors or threads) but this isnt always the case.

Secondly, it Is a huge misconception to say that the design does not change when concurrent programs are being written. In many cases, a concurrent algorithm might have a design different from what you would have gotten with a single-threaded system since the what and when of the code has been decoupled.

It is also wrong to ignore the need to understand concurrency when you are working with containers like Web or EJB. It is recommended that you know what exactly your container is doing, as this will help you deal with issues that may be encountered, such as a deadlock or concurrent update issues.

Correct concurrency is complex, and you should expect to incur some overhead. In many cases, concurrency will require you to make fundamental changes in your design. Even for solving simple problems, you will have to put in extra work in terms of writing extra code or lose out a bit on the performance of your code.

Concurrency Defense Principles

Despite its benefits, writing concurrent code has some problems as well. To defend your system from these problems, you must follow the concurrency

defense principles which are stated below:

Single Responsibility Principle

We have stated this rule earlier in this book. According to the SRP, a given method, class, or component is expected to have a single reason for changing (responsibility). In fact, since concurrency design is complex enough to stand-alone as a responsibility (a reason for change), separating it from the rest of the code makes perfect sense.

However, many developers brazenly break this rule by embedding concurrency code directly into their production code. This isnt acceptable practice since concurrent code has its own life cycle of development, which includes change and tuning. The code also has its challenges, most of which are different from that of non-concurrency related codes.

On its own, badly-written concurrency code can fail in a lot of ways. This makes it challenging to deal with, and even worse when you have it surrounded by other application codes. This is why you should keep all concurrency-related code separately from other code.

Corollary 1: Limit the Scope of Data

It is recommended that you limit the scope of shared data in your code. Modifying two fields with a shared object can interfere with each other. This can lead to unexpected behavior in your code. When you have shared data in multiple places, you may forget to protect one of the shared data points. This will break all the code if the shared data is modified for any reason. Sharing data will also make it difficult to debug the code as it will be difficult to determine the source of failure in your code.

Corollary 2: Use Copies of Data

Instead of sharing data, a better alternative is to make copies of the object and treat it as read-only. You may also copy objects and merge results from multiple threads into a single thread.

Corollary 3: Threads Should Be as Independent as Possible

Threaded code should be written in a way that each thread exists independently, i.e., not sharing data with other threads. This way, each thread will process one client request, and all the required data will come from an unshared source. It is recommended that you try to partition data into

independent subsets. This makes it possible to operate each one as independent threads.

Beware of Dependencies Between Synchronized Methods

When writing concurrent code, you should be careful of dependencies between synchronized method. This can lead to subtle bugs that are difficult to detect. In Java, it is possible to have synchronization between individual methods to protect them. However, having more than one synchronized method may cause systems to be written incorrectly. Hence, in any shared object, it is recommended that you do not use more than one method. If this rule will be broken, and one more than one method will be used on a shared object some of the ways to correct the code include:

- **Client-Based Locking**: you can make the client lock the server before the first methods are called. Also, ensure the lock covers the code calling the last method.
- **Server-Based Locking**: another alternative is to create a method to lock the server then calls the method before unlocking again.
- **Creating An Adapted Sever**: an adapted server can serve as an intermediary to lock the main server. This is done when the original serve cannot be modified.

Keep Synchronized Sections Small

It is recommended that you keep the synchronized sections of your code as small as possible. One way to introduce a lock into your code is to make use of a synchronized keyword. At any given time, all sections of the code having the same synchronized keyword will have a single thread executing through them. Such locks create delays and add some overhead to your code. Hence, you want to limit them as much as possible. Although it is important that you keep critical sections 9sections of your code that must be protected from simultaneous use) of your code guarded, having a code with as few critical sections as possible is important as well. A common mistake programmers make is to make critical sections very large to accommodate the synchronized statement. This will only degrade the performance of your code.

Writing Correct Shut-Down Code

The nature of the shut-down code makes it difficult to write. Designing a

system that is only meant to work then shut down on its own after a while is not easy. It can be difficult to pull off. One of the problems with writing shutdown code is the issues of deadlock. i.e., threads may never receive a continuous signal.

For instance, if you have a parent thread that produces several sub-threads, then waits for them to finish running before finally releasing its resources and shutting down. If one of the spawned threads fails to send a signal, the parent thread will be unable to shut down forever.

A similar problem may occur in a shutdown system with children threads that have a consumer-producer pair as one of the sub-threads. When the parent thread sends a signal for all the children to finish, the producer code will receive the signal and shut down immediately without sending the signal to the consumer thread. The thread is stuck waiting for a shutdown message that never comes. This will prevent the parent thread from shutting down as well

Hence, given how common situations like the ones described above are, it is recommended that you think about shutdown code early on in your project. Be prepared to spend a lot of time getting the code to shutdown correctly and as planned.

Testing Threaded Code

Writing tests for threaded code is difficult. This is because testing only minimizes risks. It does not guarantee the correctness of code. Things are even more complex when you are working with multi-threaded codes with one or more threads sharing the same data. They are a lot more complex to deal with. You should write tests that can potentially expose problems in your code and run them frequently as you code. If such tests fail, you can easily track down the source of the failure and fix it until the code passes the test.

Chapter 13: Clean Design Rules

In this chapter, we will discuss some of the general techniques that you can employ to keep your entire code base as clean and user-friendly as possible by focusing on emergent code design. You can write simple and expressive designs by following Kent Becks rules of simple design. These rules help to create good and functional design. It can also help you gain insights into the structure of code. Following these rules will also make it easier to apply some principles we have discussed earlier in this book, such as SRP and DIP.

Kent Becks Rules of Simple Design

According to Kent Beck, a design can be said to be Simple if it agrees with the following rules:

1. ***It Runs All Tests***

One of the prime purposes of every design is to produce a system that works as intended. This is not merely about having a system that looks good on paper. There must be ways to verify if the system you have built actually works or not. If it doesnt, then all your design effort is a mere waste.

A testable system is one that has been subjected to various comprehensive tests and passes the tests at all times. If your system cannot be tested, there is no way to tell if it performs as intended or not. Hence, such a system should never be deployed in the first place.

Another advantage of a testable system is that it pushes our code towards a design with small singular-purpose classes. Classes that align with the single responsibility principles are easier to tests. Writing more tests will eventually push your code into more simplicity. Hence, it is safe to conclude that making sure your systems is testable is one of the key ways to create a better design.

Testing code also helps to minimize coupling. Codes with tight coupling are difficult to tests. To write more tests on coupled code, you will have to apply principles and tools such as dependency injection, abstraction, interfaces, and so on to reduce coupling in the code. All of these will further enhance the design of your code.

Tests empower you to keep your code and its classes clean by incrementally refactoring the code without fear of breaking anything. When a new line of

code is added, you should pause and reflect on how the change will affect the design of your code. If the new change degraded the code, then you should clean it up.

Tests should be run continually to show that nothing has been broken in the code. While refactoring, you can apply any of the techniques you have learned so far in this book, such as cohesion, separating concerns, shrinking classes and function, decoupling, changing names and choosing better ones, and so on. Tests also make it easier to apply the three other rules of simple design that will be subsequently discussed in this chapter.

2. ***Contains No Duplication***

One of the major hindrances that can affect a system is duplication. Duplication introduces additional risks, more work, and complexity to your system. Duplication can occur in various forms. The commonest and easiest to spot is lines of code that are exactly alike. But duplication can occur in implementation as well. To keep your system clean, you must learn to avoid and eliminate duplication even in a few lines of code.

Extracting commonality even at the lowest levels makes it easier to recognize places where the single responsibility principle is being violated. When this is the case, a newly extracted method can be moved into another class. This helps to increase the visibility and overall readability of the code. This will also make it easier to reuse the new smaller methods in a different context, which can reduce the complexity of the code dramatically.

3. ***Expresses the Intent of the Programmer***

Good code is expected to clearly express the intent of its original author without causing confusion. Writing expressive code takes time and care. But making your code clear manes less time will be spent on trying to understand what the code says.

If you have ever had to deal with convoluted code, it is easy to see why this rule of keeping code expressive is important. At the time the programmer was writing the code that you are currently finding difficult to understand, it was easy for him or her to understand what the code does. This is because, at the time of writing code, you have a deep understanding of the problem you are trying to solve. It may surprise you that you will experience some level of difficulty trying to unravel a convoluted code you wrote yourself some

months or years after.

If the code is not expressive enough, other people trying to maintain your code will have the same issue too. The implications for code maintenance are enormous. Any change is accompanied by the risk of creating potential defects in the code, and more time will have to be spent to understand what the code does. There is also the risk of getting misunderstood. All of these reasons further support the need to write expressive code whose intent is clear to others apart from the original author. The more expressive code is the less time and effort that will be spent on understanding it.

Some of the ways to make code expressive include:

- Choosing good names- the name of a class or function should clearly express its responsibilities
- Keep your classes and functions as small as possible: small functions are easier to name, write, understand.
- Use standard names: one way to improve expressiveness is to use standard nomenclature such as VISITOR or COMMAND in class names that clearly describe what the classes do to any developer looking at the code.
- Write well-written unit tests: a unit test is a form of documentation. When someone reads your unit tests, they can get a quick practical understanding of what the class does. This is why you must ensure that they are clearly and well written as well.

4. Minimizes the Number of Classes and Methods

So far, we have learned several concepts and principles that are aimed at keeping our methods and classes small. The obvious danger here is that you may be left with too many tiny methods and classes in an attempt to keep to these rules. According to this rule of simple design. It is also important that you keep the number of functions in your code low as well.

A mindless dedication to laws and principles may increase the number of methods and classes so much that they are difficult to handle. This eventually compromises the simplicity of the code. For instance, a developer that insists on always creating a separate interface for every class or one that always wants to keep fields and behavior separated into behavior and data classes

will only create several tiny classes. Such a practice does not align with the principles of simple design. At the end of the day, the goal should be to keep your classes and functions small while keeping the overall systems small as well.

Rules for Writing Comments

1. **Inappropriate Information**

Comments are meant to hold technical notes relating specifically to the code design. It should not hold any irrelevant or inappropriate information. Comment should not hold any information that can be better held in another documentation system. For instance, some pieces of information are better held in record-keeping systems such as your issue tracking system, source control system, and so on. Such information will only overcrowd your code if included in your source file. Hence, information such as meta-data (e.g., authors, SPR number, last modified date, etc.) should not be included in your min comments.

2. **Obsolete Comment**

Keep obsolete comments out of your code as they tend to become a floating mess of irrelevant information that will confuse a reader. Obsolete code is a common challenge to writing clean code since comments tend to get old quite quickly and will float away from the code they once explained. Check your code regularly for comments that have become old and irrelevant. When you find such comments, you can either update them with the correct information or simply remove them entirely.

3. **Redundant Comment**

Redundant comments are useless as they describe things that already describe themselves sufficiently. You should avoid adding comments just for the sake of it. If your code already says enough about itself, adding a comment that says nothing more (or even less) is only a waste of space and will only crowd your program.

4. **Poorly-Written Comments**

Comments are not recommended. But if you will be writing them at all, then you should take care to write a good one. Ensure that every comment you add

to your code is written in the best way you can. Your comment should be grammatically correct with the right punctuation. Be careful with the words you use and ensure that the information you are trying to communicate will be clear to anyone reading the code in your absence. Dont ramble and keep the comment as brief as possible.

5. **Commented-Out Code**

There are few things quite as annoying as commented out code. It drives any developer crazy as you are left wondering if the commented out code has any relevance to the code or if there are any plans for it. With each day that passes commented out, code becomes more irrelevant using variables and calling a function whose names no longer exist in the code. The rule for dealing with commented code is simple. Just delete it! Get rid of it immediately. The good thing is that the source code control system will still remember it, so if anyone needs it later, they can always find the previous version here.

Rules for Build Environments

1. **Build Should Have Not More Than One Step**

Building a project should require not more than a single trivial operation. There is no reason to check any little pieces from the source code control or anywhere else, neither do you need a sequence of context-dependent scripts or arcane commands to build each element. When you need to build an element, all that is needed is a simple command that checks out the system and another simple one to build. No need to search for extra JARs, XML files, or any other artifact.

2. **Tests Dont Require More Than A Single Step**

Given that running tests are a basic but important process, it should be easy, quick, and obvious. You should be able to run all your unit tests with a single command and nothing more. In the best-case scenario, running tests should be as easy as clicking a button on the IDE you are using. In the worst case, simple commands in a shell should run all the unit tests.

Rules for Functions

1. **Functions Should Not Have Too Many Arguments**

You should keep the arguments in your functions to the smallest number possible. A maximum of three arguments is ideal. More than three should only be allowed if you are sure that there is no better way to cut down the number.

2. **Dont Use Output Arguments**

Anyone reading your code will expect arguments to be inputs and not outputs. Hence, an output argument is counterintuitive and should be avoided. If your function must change the state of something within your code, it should only change the state of the object that it is called on.

3. **Avoid Flag Arguments**

When you use Boolean arguments, it is obvious that your code is already doing more than one thing. Such arguments should be avoided entirely as they can be confusing.

4. **Remove Dead Functions**

If you find a method that is never called, then you should discard it. Dead code like this is only a waste of space and will make your code smell. Feel free to delete it. You can always recover it from the code control system; there is a need for it later.

Rules for Naming Things

1. **Choose Descriptive Names**

The importance of choosing descriptive names cannot be overemphasized. You should take your time when choosing names for anything in your code. Be sure to use names that are clearly descriptive without room for ambiguity. Names make your code readable. It accounts for up to 90% of the readability of your code. Hence, you should heed this rule every time. Also, since your code tends to evolve, you may have to reevaluate the appropriateness of the names you use with time.

Using well thought out names will make all the difference between a code that makes perfect sense to the reader and one that is just a jumbled mess of symbols and numbers. Good names load a code structure with description, tipping the readers expectations about what the module functions do.

2. Choose Names at the Appropriate Level of Abstraction

This is a rule that is difficult to follow, but one that you must strive to obey, nonetheless. Your names should not communicate implementation. Instead, each name should reflect the appropriate level of abstraction of the function or class that you are working on. It easy to confuse levels of abstractions, and that is why people choose names that mix them up. But you should pay attention to this. When you find a code that is named at a lower level of abstraction than it should, you should change those names to reflect the appropriate level. Such dedication to the continuous improvement of code is important for maintaining the readability of your code.

3. Use Standard Nomenclature Where Possible

Standard conventions or existing name usage make it a lot easier to understand what a name does. For instance, the word Decorator should appear in the names of decorating classes if you are making use of the DECORATOR pattern. E.g., the name AutoHangupModemDecorator is a perfectly fitting name for a class that decorates a modem that hangs up automatically after a session ends. There are numerous examples of standards like this that you can use to make your names follow conventions that readers can relate to.

Asides general standards, some teams also invent their own standard naming systems in a particular project. If this is the case, then it is recommended that you use terms from this set standard in your code as much as possible. This will make it super-easy for other members of your team that may look at your code sometime in the future.

4. Use Unambiguous Names

You should name functions and variables in a way that describes what they do with no ambiguity. Avoid using broad and vague terms that say next to nothing about the workings of the function or name that can easily confuse a reader that the function does something else. No matter how long the name will have to be, a name that explanatory is recommended over a short and vague one.

5. Use Long Names for Long Scopes

Still, on name lengths, the length of a name should be related to its scope. Tiny scopes can take very short variable names. However, if you are dealing with big scopes, then longer names are recommended. For example, simple one-letter variables names like j and I are ideal for code with tiny scopes that are five long as in the example below.

```
private void rollMany(int n, int pins)
{
for (int i=0; i<n; i++)
g.roll(pins);
}
```

On the other hand, if the scope of the code is longer, a short variable name like this will lose its meaning. In such cases, it is recommended that you use a longer but more precise name.

6. Avoid Encodings

Encodings are problematic, and that is why they should be avoided. There is no need to encode names with scope or type of information. m_ or f prefixes have become redundant in todays coding environment. Project and subsystem encodings have become redundant as well. Adding such encoding to your names will only distract your reader, and they only pollute your code.

7. Use Names that Describe Side Effects

When we say names should describe what a function or variable does, we mean that to the very last detail. You should choose names that describe your function, class, or variables, including those simple actions that are hidden.

Rules for Tests

1. Insufficient Tests

What is the ideal number of tests in a test suite? Most developers simply use their discretion and stop when they feel the tests seem to be enough. But this shouldnt be so. Your test suite is expected to test everything that could break in your code. Your tests are still insufficient if there are conditions in your code that are yet to be explored or calculations that are yet to be validated.

2. Use a Coverage Tool

The purpose of a coverage tool is to report gaps within your testing strategy. With a coverage tool, it is easier to find modules, functions, and classes that have not been sufficiently tested. Fortunately, most IDEs come with visual indicators for this already. A green marking line represents covered code, while red ones indicate unchecked code.

3. Dont Skip Trivial Tests

Trivial tests are easy to write, and they have high documentary value. Hence you should never skip them for any reason.

4. An Ignored Test Is a Question about an Ambiguity

When you are uncertain about the behavioral detail of code due to unclear requirements, you should express your uncertainty with a commented out or ignored test (a test annotated with @ignore).

5. Test Boundary Conditions

Boundaries are delicate areas of code. Hence, special care must be taken around them since we are not always sure of what to expect. You should be extra-careful about test boundary conditions, so you dont misjudge them.

6. Exhaustively Test Near Bugs

Bugs tend to congregate. When you find a bug, there is most likely another one lurking close by. Thus, it is recommended that you carry out an extensive test around functions with the bug to see if there are other bugs around it.

7. Patterns of Failure Are Revealing

It is possible to diagnose problems in your code by finding a pattern in the way the test cases you write fail. This is another reason why you should make your test cases complete and order then in a reasonable way, as this will make it easier to identify patterns. Simple patters like observing that all the tests fail when you add input with more than five characters or even the red and green color patters of a test report then give you some insights into what could be wrong with your code.

8. Tests Should Be Fast

Slow tests are likely to be dropped from the test suite when things get tight.

Slow tests are unlikely to get run. Hence you should do all you can to make your tests fast.

Additional Rules

1. **Multiple Languages in One Source File:** Even though most modern IDE allows it, do not put multiple languages into the same source file. You may inevitably have to use more than one, but you should keep the number of extra languages within a source file to the barest minimum.
2. **Obvious Behavior Should be Implemented:** according to the principle of least surprise, any class of function should implement behavior that other programmer looking at the code will reasonably expect. When this rule is not obeyed, a reader of your code can no longer trust their intuition to interpret the function names.
3. **Code Should Behave Correctly at Boundaries:** you should do your due diligence to ensure that your code behaves correctly at the boundary by testing for every boundary condition. Because boundaries are so unpredictable, you cannot rely on your intuition.
4. **Overriding Safeties is Dangerous:** safety tests are there for a reason. And even if they make it difficult to run some tests or you have to turn them off to get a build to succeed, you should understand that you are doing so at major risk to your system.
5. **Base Classes Should Not Depend on Their Derivatives:** concepts in the higher level base classes should not depend on concepts in the lower level classes. This rule only has a few exceptions
6. **Too Much Information:** if your module is well defined, it should have a small interface that does a lot with limited information. A poorly defined module, on the other hand, has a deep and wide interface that uses multiple gestures just to get simple things done.
7. **Dead Code:** dead codes are codes that cannot be executed for one reason or the other. They commonly exist in if statement as checks for conditions that can never occur or in the catch block of a try

code that never throws. Dead code will start to smell after a while, which is why they should be removed from the system whenever you find them.

8. **Be Consistent:** be careful with the conventions you should, and once you do, you should follow the same convention throughout your code. Variable names and methods names should be used consistently throughout your code. By maintaining consistency, you keep your code simple and easy to read or modify later.
9. **Feature Envy:** According to one of Martin Fowlers code smells, the methods of a class should only be interested in the functions and variables of the same class and not in that of other classes. A method should not use mutators and accessors of some object in another class to manipulate it. This is known as envying the scope of another class, and it should be avoided.
10. **Keep Your Code Clutter-Free:** you should keep your code source file clean and neatly organized. Get rid of things that clutter your code, such as unused variables, uncalled functions, or comments that add no information to your code, among other things. They may not affect how your code compiles, but they will unnecessarily clutter up your code.
11. **Structure Over Convention:** When writing code, you should consider the design of your code over rules and conventions. Although conventions are great and needed to keep your code readable, when it comes to choosing one over the other, then you should choose structure over convention.
12. **Express Conditionals as Positives:** Most people find it easier to understand positives compared to negatives. This is why it is recommended that you write conditionals as positives instead of negatives whenever possible.

For instance, you should write

if (buffer.shouldCompact()

Instead of

if (!buffer.shouldNotCompact())

Functions Should Descend Only One Level of Abstraction: A function should only contain statements at the same level of abstraction. Also, all of the statements within a function are expected to be only one level below the operation described by the function name. This is a simple room that commonly gets mixed up by programmers.

13. **Dont Hide Temporal Couplings:** Creating temporal couplings is, most times, unavoidable. However, you should ensure that the coupling is not hidden. Instead, the arguments of your functions should be structured in a way that the order of their call is obvious to anyone reading the code. One way to expose temporal coupling is by creating bucket brigades. When you do this, each function will produce the result needed for the next function. Hence, it will be impossible to call the functions out of order reasonably. While this may increase the syntactic complexity of your function, your code will still be clean.

14. **Be Precise:** Bad code is usually the consequence of bad decisions. Every program your write is an aggregation of the decision you make. This is why you should be precise about your decisions as you code. Whatever you decide to do, be sure of why you are doing it that way and not in any other way. You should also be aware of the consequences of your decision and how you will deal with the exceptions that may arise. For example, if you decide to call a function that is likely to return null, then ensure that you check for null. Before you write a query for what you think might be the only record in a database, be sure to check your code and ensure it is indeed the only one and there arent others. Dont expect the first match of your query to be the only match, do not use floating-point numbers to represent a currency. These are some of the precision rules to keep in mind to keep your code spot on.

Conclusion

Most of the rules have been discussed in greater detail earlier in this book. But they are worth mentioning again for emphasis. This is not a rule book, so we dont expect you to memorize all the conventions. Instead, try to understand the reasons for them and the overall concept of clean coding. This

will make it easier to apply these conventions. With practice and continual dedication to writing clean code, you will eventually master the beautiful art of writing clean codes.

Final Words

As you can see from reading this book, learning to write clean code is hard work. It is easy to go through all the techniques and principles about writing clean code, but that is only half of the job. The real work begins when you start writing or cleaning up your next program after reading this book.

The techniques of clean coding are numerous. They are probably too much for any programmer to completely cram. And we dont want you to. The goal is to get you familiar with as many of these principles as possible.

But thats about all to it. The rest of the work depends on you. Writing clean code will require your dedication and willingness to follow the rules. It begins with an understanding of why writing clean code is important. It will take some time and a lot of care to follow clean coding conventions, but more time will be lost on reading and maintaining messy code.

All that is left from here is practice; you must practice clean coding techniques yourself. You will also have to watch other practices and correct unclean code. You will fail at it sometimes and will also see others fail at it too. But with time, dedication, and complete obsession with writing clean code, you will find yourself gradually mastering the techniques that have been painstakingly compiled in this book.