Chapter 1: Clean Code

**There Will Be Code:**

* A book about code is arguably said to be behind times and that programmers won’t be needed anymore due to codes being generated automatically instead of being written.
* An argument that code will be lost is nonsense because code itself represents the detail of the requirements, a specification that the machine follows and execute in every detail; also known as programing.
* Languages may improve, grow, advance and continue to increase in its abstraction which in itself is good but it will never replace code, because however advance it may be it will still need to follow an accurate specification that machines follows in order to execute it.
* People who thinks codes will disappear hopes that humans can create a machine that can do what we want rather than what we say, however a machine can never fully understand vaguely specified requirements into an executable program that is precise to what humans needs.
* As long as humans can never understand and fully provide and create successful systems from their client and customer vague instructions, then such machine will never be created.
* Code will always be present since it is the language that we specify and convey our requirements to the machine, and no matter how many languages or tools are created to help those requirements form a structure, it will never get close to a necessary precision that would make code obsolete.

**Bad Code:**

* Kent Beck’s said in his book Implementation Patterns that “this book is based on a rather fragile premise: that good code matters.” but it’s disagreeable because good code does matters and its most supported and overloaded of all the premise in programming.
* Without a good code a company in the late 80s that wrote a popular killer app flop and went out of business due to the bad codes that produced multiple bugs and errors.
* Bad code is the result of rushing the product to market, gradually adding features overtime made the bad code worse until it resulted to the company shutting down.
* Wading is what it’s called when a programmer is being hindered by a bad code that we have no idea of what’s going on.
* Programmer usually writes bad code due to rushing the deadline, no time to clean up the codes, too tired and frustrated with the program or just forgot about the things to that needs to be done or just choose to ignore it so you can move on to the next program.
* We all experience having to leave the mess we made for tomorrow, or having the program work with bad code and not wanting to touch it again in case it gets broken again or not wanting to clean it up while not knowing the law of LeBlanc’s: Later equals never.

**The Total Cost of Owning a Mess**

* Programmers with few years of experience have experience being slowed down by someone else’s code and the degree of slowdown will gradually increase overtime as more changes are made with the code, and every error that gets fix creates more until finally the mess becomes so big it’s impossible to fix it.
* As it the mess keeps piling up the productivity of the programmers continues to decrease and when the productivity decrease the management will add more programmers to the team in an attempt to increase productivity.
* However the newly added programmers are not well versed to the system and does not fully understood the changes they should make if it fits with the design intent or would it prevent it.
* The programmer would be under pressure and stress to increase productivity resulting in additional mess being added and making the productivity even further down to zero.

**The Grand Redesign in the Sky**

* Programmers stuck with this problem will start to rebel and complain to management about the system and its mess.
* The management not able to deny the developers and the mess that resulted in the system will assign a new team that will create a brand new system.
* The creation of the brand new system will be managed by only the brightest and highest developer team while everyone else will continue to maintain the old one.
* It will be a race for the new team to build what the old system is capable of, and keep up with the new changes the old system does.
* As long as the new system is not done the old system will not be replaced by the management and this race can take a really long time until the original members of the new team is gone and current members of the old system are left to demand the old system be replaced.

**Attitude**

* This type of problems of having to fix code that should’ve been just one line turned into multiple modules is due to us being unprofessional.
* We made reasons and complain about requirements blame managers, customers and marketing when the fault is within ourselves.
* We are responsible for the code we produce and the managers and marketers depends on us for the information they need, users look up to us for validation for how the requirement will fit the system. The project manager look at us to work out the schedules.
* Therefore we share a great deal of the responsibility of the failures that has to do with bad code so we must communicate to them and not be shy to tell them what we think.
* Communicate with your project manager truthfully, as they want good code. Their job is to manage the schedule and requirements, so you must focus on the code as your priority.

**The Primal Conundrum**

* A conundrum where programmers with more than a few years of experience knows that previous messes slow them down and yet all programmers still feel the pressure to make messes due to the deadlines.
* When in truth professional programmers know the second part of the conundrum is wrong, you will not be able to finish the deadline by writing messy codes because it will cause you trouble in the end, it’s much better to code as clean as possible to have less error to fix at the end.

**The Art of Clean Code?**

* Writing code is not like painting a picture, this means that most people recognize a bad or good painting but recognizing good art from bad does not necessarily mean those people can paint, so it’s the same as coding, even if you recognized clean from bad codes does not mean you can write clean code.
* Some programmer are born with code-sense while others have to acquire it, this enables us to recognized good and bad code as well as applying the discipline to turn bad code to good code.
* Programmers that writes clean code is and artist that can transform any blank screen into an elegant coded system.

**What is Clean Code?**

**Bjarne Stroustrup, inventor of C++ and author of The C++ Programming Language**

* According to Bjarne Stroustrup clean code should be pleasing to read, elegant, efficient and straightforward, the dependencies minimal, error handling complete and performance close to optimal.
* The consequence of inelegance is tempt, bad code temps the mess to grow, as Dave Thomas and Andy Hunt said, broken windows allows other people to not care about them and make them actively break and start the process of decay much like bad code.
* Clean code does well, it is focus unlike bad code that tries to do too much making it confusing intent and purpose.

**Grady Booch, author of Object Oriented Analysis and Design with Applications**

* Clean code is simple, direct, and easy to read, like well-written prose.
* Clean code should expose the tensions in the problem and build to a climax, leading the reader to an "Aha!" moment of understanding.
* Grady Booch emphasizes the importance of "crisp abstractions" in clean code, meaning that code should be decisive, without hesitation or unnecessary detail, while still providing all the necessary information.
* Grady's use of the phrase "crisp abstraction" to be a fascinating oxymoron, as "crisp" implies decisiveness and conciseness, while "abstraction" suggests something more abstract and less concrete.
* Clean code should be matter-of-fact, containing only what is necessary, and should be perceived as decisive by the reader.

**“Big” Dave Thomas, founder of OTI, godfather of the Eclipse strategy**

* Clean code should be easy to read and enhance by other developers.
* It has unit and acceptance tests, meaningful names, and one clear way of doing things.
* Clean code has minimal dependencies, with clear and minimal API, and is considered literate as it should be easily readable by humans.
* Big Dave emphasizes the importance of clean code being easy to change and tied to tests.
* Dave values smaller code and the concept of literate programming.

**Michael Feathers, author of Working Effectively with Legacy Code**

* Clean code is characterized by an overarching quality of care, demonstrated by its simplicity and orderliness.
* Clean code cares deeply about the craft and has taken the time to ensure that the code is easy to read and understand.
* The key to clean code is care, and the main topic of the book is how to care for code.
* Michael emphasizes that clean code is code that has been taken care of, and that someone has paid attention to the details to keep it simple and orderly.

**Ron Jeffries, author of Extreme Programming Installed and Extreme Programming Adventures in C#**

* Ron started his career as a programmer writing code in Fortran language at the Strategic Air Command and has experience with various programming languages and systems.
* Ron focuses on Beck's rules of simple code which include running tests, no duplication, expressing design ideas in the system, and minimizing entities in the code.
* Expressiveness to Ron includes meaningful names and avoiding objects/methods that perform multiple tasks. He may change names several times to achieve expressiveness.
* Duplication and expressiveness are two key elements of clean code for Ron, with the third being early building of simple abstractions.
* The process of building simple abstractions helps Ron avoid implementing arbitrary collection behavior and preserve the ability to change implementation later.

**Ward Cunningham, inventor of Wiki, inventor of Fit, coinventor of eXtreme Programming. Motive force behind Design Patterns. Smalltalk and OO thought leader. The godfather of all those who care about code.**

* It emphasizes the importance of clean and beautiful code in software development.
* clean code is code that is easy to read and understand, with minimal duplication and high expressiveness.
* It mentions that when code is clean, it is not surprising and it should be pretty much what the reader expects.
* Beautiful code is code that makes it look like the language was made for the problem, rather than the other way around.
* The responsibility of making code simple and beautiful lies with the programmer, not the language.
* Programmers should strive for clean and beautiful code to make their programs easier to read and maintain.

Chapter 2: Meaningful Names

**Use Intention Revealing Names**

* It’s important to use good names for variables, functions, and classes, as it makes the code easier to understand and change.
* The name of the variable should answer questions about its existence, purpose, and usage. A name that requires a comment to understand its intent is not a good name.
* An example of a poor names is "d" but its better if its names "elapsedTimeInDays" that reveal the purpose and unit of measurement.
* The code lacks explicit context for its variables and constants, making it hard to answer questions about the contents of theList, the significance of the zerothq subscript, the value 4, and the purpose of the list being returned.
* Renaming variables and concepts in code can greatly improve its readability and understandability.
* By giving meaningful names to variables and using simple classes, code can become more explicit and easier to follow.
* This is demonstrated in a code sample from a mine sweeper game, where the variable "theList" was renamed to "gameBoard" and simple arrays were replaced with a class for cells.
* The code remained simple, but the changes made it much clearer and easier to understand. The power of choosing good names lies in the ability to reveal the intention of the code and eliminate magic numbers.

**Avoid Disinformation**

* Avoid being confusing when naming variables as it can be uninformative to others that has no idea what that variable abbreviation means.
* Its better to be direct to the point, if a variable points to an group of account name the variable “accountgroup” or something similar than can be easily understood.
* Avoid naming variables that has little difference to each other.
* An example of disinformation would be the lower case l and uppercase O which can be confused to the number one and zero.

**Make Meaningful Distinctions**

* Avoid writing code that only satisfies a compiler or interpreter as it can lead to arbitrary changes in naming, such as misspelling, which can result in an inability to compile.
* Ensure names are meaningful and differentiate between items. For example, use "source" and "destination" instead of "a1" and "a2" in a function.
* Avoid meaningless distinctions in names, such as using number-series or noise words, such as "Info" or "Data".
* Avoid using redundant words in names, such as "variable" in "variableName".
* Make a meaningful distinction in names, for example, using prefix conventions like "a" for local variables and "the" for function arguments.
* Ensure names are distinguishable from each other to avoid confusion. For example, "moneyAmount" is distinguishable from "money".
* Avoid using indistinct names, such as "customerInfo" and "customer".
* Avoid using indistinct functions, such as "getActiveAccount", "getActiveAccounts", and "getActiveAccountInfo".

**Use Pronounceable Names**

* It argues that a significant part of the human brain is dedicated to the concept of words, and it would be a shame not to take advantage of that ability.
* For example an acronym "genymdhms" for generation date, year, month, day, hour, minute, and second, will be difficult for people to discuss the variables.
* To illustrate the difference, compare a class named "DtaRcrd102" with a class named "Customer." The latter allows for intelligent conversations about the code and its components.

**Use Searchable Names**

* The use of single-letter names and numeric constants in code can be difficult to locate in a body of text.
* Using common letters or long numeric constants can lead to bugs and hinder searches for the intended information.
* It is recommended to use longer names for variables to improve searchability and limit the use of single-letter names to short, local variables within methods.
* The length of a name should reflect its scope.

**Avoid Encodings**

* Having additional encodings adds to the workload and is not necessary.
* Encoding type or scope information into names creates an extra challenge of deciphering and can be a mental burden for new employees.
* This added burden of learning another "language" of encoding, on top of learning the existing code, is unreasonable.
* Encoded names are often difficult to pronounce and easily mis-typed, making them a problematic solution.

**Hungarian Notation**

* Hungarian Notation was used to remember data types in limited language environments.
* Modern languages have advanced type systems, making type encoding unnecessary and hindering code readability and accuracy.
* Type encoding is now considered an obstacle in modern programming.

**Member Prefixes**

* You don't need to use "m\_" as a prefix for member variables.
* Classes and functions should be small enough that they're easily recognizable.
* Use an editing environment that highlights members to distinguish them.
* People tend to ignore prefixes and they become cluttered and outdated markers in the code.

**Interfaces and Implementations**

* Use simple names for interfaces and avoid prefixing them with 'I'.
* Distinguish concrete implementation classes by adding appropriate suffixes such as 'Imp' or 'Concrete'.
* This helps keep the code clean and reduces the amount of information that the users need to know.

**Avoid Mental Mapping**

* Avoid using single-letter variable names unless they are traditional loop counters.
* Use clear and descriptive names that can be easily mapped to the actual concept by the reader. Clarity is important in writing code that others can understand. Professional programmers prioritize clarity over showing off their mental juggling abilities.

**Class Names**

* Class names should be noun or noun phrases, such as "Customer," "WikiPage," "Account," or "AddressParser."
* Avoid using words like "Manager," "Processor," "Data," or "Info." Class names should not be verbs.

**Method Names**

* Methods should have verb or verb phrase names like postPayment, deletePage, and save.
* Accessors, mutators, and predicates should be named for their value and prefixed with get, set, and is.
* Use static factory methods with descriptive names for overloaded constructors and consider making the corresponding constructors private.

**Don’t Be Cute**

* Methods should have clear names that accurately describe their purpose, rather than relying on cleverness or humor.
* Avoid slang, and choose clarity over entertainment value.
* Name functions in a way that accurately conveys their purpose to all users and should say what it mean and mean what it says.

**Pick One Word per Concept**

* Use consistent terms for abstract concepts and avoid using multiple words for the same concept.
* Avoid using inconsistent names for classes, objects, methods, and functions.
* Choose clear and concise names that stand alone and can be easily remembered.
* This helps programmers use your code more easily and efficiently.

**Don’t pun**

* Use distinct words for distinct concepts to prevent confusion.
* Avoid using the same term for different purposes as it creates a pun.
* Follow the "one word per concept" rule, but ensure the different methods with the same name have semantically equivalent parameter lists and return values.
* When creating a new method, choose a name that accurately reflects its purpose, such as insert or append instead of add, if the semantics are different.
* The goal is to make the code easy to understand with a quick skim, not an intense study.

**Use Solution Domain Names**

* Use technical terms in naming methods and classes, as they are familiar to programmers.
* Don't rely on problem domain names as they may not be clear to others and require constant communication with the customer.
* Using CS, algorithm, pattern, and math terms makes it easier for coworkers to understand the code.
* Technical names are more appropriate for complex programming tasks.

**Use Problem Domain Names**

* Use technical terms from computer science when available for programming concepts.
* Use names from the problem domain when technical terms are not available.
* Separate problem domain and solution concepts in code by using appropriate names.

**Add Meaningful Context**

* When naming variables, use names that are meaningful in context and provide clarity.
* If the name is not meaningful in and of itself, add context by enclosing it in well-named classes, functions, or namespaces.
* If necessary, use prefixes to add context.
* Creating a class for related variables can also help provide context and make the variables easier to understand.
* Avoid using variables with opaque meanings.

**Don’t Add Gratuitous Context**

* Using class names like "Gas Station Deluxe" and "GSDAccountAddress" is not recommended as it makes it difficult for the IDE to help you.
* Shorter names are better, but must be clear.
* Add only necessary context to names. Instances of class "Address" can be named "accountAddress" or "customerAddress", but "Address" is a good class name.
* To differentiate between different types of addresses, more precise names like "PostalAddress", "MAC", and "URI" can be used.
* The goal of naming is to make it more precise.