# **Chapter 1: Clean Code**

## **There Will Be Code**

* The most important details in this text are that code is not going away, and that the level of abstraction of our languages and the number of domain-specific languages will continue to grow. This will not eliminate code, as all specifications written in higher level and domain-specific language will still need to be rigorous, accurate, and so formal and detailed that a machine can understand and execute them. Additionally, code is the language in which we ultimately express the requirements, and we may create languages that are closer to the requirements, but we will never eliminate necessary precision. Finally, it is important to remember that well-specified requirements are as formal as code and can act as executable tests of that code.

## **Bad Code**

* The important details of this context are:
  + Program shouldn’t be push if you are in rush
  + Programmers also experienced this problem that is slamming a module together so that to proceed to the next.
* “Wading”
  + Through bad code

## **The Total Cost of Owning a Mess**

* If you have been a programmer for more than two or three years, you have likely been slowed down by someone else's messy code. This can be significant over the span of a year or two, as teams that were moving very fast at the beginning of a project can find themselves moving at a snail's pace. As the mess builds, the productivity of the team continues to decrease, asymptotically approaching zero. Management adds more staff to the project in hopes of increasing productivity, but they are not versed in the design of the system and are under horrific pressure to increase productivity. This leads to more and more messes, driving the productivity ever further toward zero.

### **The Grand Redesign in the Sky**

* + Eventually the team rebels and inform the management that they cannot continue to develop in this odious code base.
  + A new tiger team is selected and everyone wants to be on this team because it’s a greenfield project.
  + Now the two teams are in a race and the team must build a new system that does everything that the old system does.
  + This race can go on for a very time, and by the time it’s done the original members of the tiger team are longer gone. If you have experienced even one small part of the story it’s a matter of professional survival.

### **Attitude**

* + The most important details in this text are that the fault is not in the stars, but in ourselves. We are unprofessional and should not be shy about telling our users what we think, even when they don't look to us for the information, they need to make promises and commitments. The most important idea is that it is unprofessional for programmers to bend to the will of managers who don't understand the risks of making messes. This is because the patient is the boss, and the doctor knows more than the manager about the risks of disease and infection.

### **The Primal Conundrum**

* + Developers should avoid making messes in order to meet deadlines, as it will slow them down and force them to miss deadlines.

### **The Art of Clean Code?**

* + The most crucial points in this work are that creating clean code is similar to painting a picture and that it necessitates a methodical application of the concept of "cleanliness." This "code-sense" is essential for producing clean code because it enables us to distinguish between good and bad programming and to map out a series of transformations that preserve behavior as we move from point A to point B. A programmer who produces clean code is an artist who can transform a blank screen into an eloquently programmed system through a sequence of transformations.

## **What Is Clean Code?**

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

* + Bjarne's code should be elegant and efficient, with logic straightforward, dependencies minimal, error handling complete, and performance close to optimal. He uses the word "elegant" to describe it, and the word "tempt" to describe the consequence of bad code. When others change bad code, they tend to make it worse. Dave Thomas and Andy Hunt used the metaphor of broken windows to illustrate the importance of paying attention to details in software design. Bjarne also mentions that error handing should be complete and that clean code does one thing well. This admonition is no accident that there are so many principles of software design that can be boiled down to this simple admonition.

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

* + Clean code should read like well-written prose, with crisp abstractions and straightforward lines of control. Grady's use of the phrase "crisp abstraction" is a fascinating oxymoron, as it is nearly a synonym for "concrete." Despite this juxtaposition of meaning, the words carry a powerful message that clean code should be matter-of-fact and contain only what is necessary.

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

* + Big Dave shares Grady's desire for readability, but with an important twist. He asserts that clean code makes it easy for other people to enhance it, and ties cleanliness to tests. He also values code that is small, rather than large, and says that code should be literate. This is a reference to Knuth's literate programming, which states that the code should be composed in such a form as to make it readable by humans.

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

* + Clean code is code that has been taken care of by someone who cares deeply about the craft, and there is nothing that can be done to improve it.

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

* + Ron began his career programming in Fortran at the Strategic Air Command. He has written code in almost every language and on almost every machine. Simple code runs all the tests, contains no duplication and minimizes number of entities, he says. With modern coding tools such as Eclipse, renaming is quite inexpensive, so it doesn't trouble me to change. Here, in a few short paragraphs, Ron has summarized the contents of this book.

### 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.

* + When you read clean code, you won't be surprised at all. You will read it, and it will be pretty much what you expected. Each module will set the stage for the next. Ward says that beautiful code makes the language look like it was made for the problem. It will be obvious, simple, and compelling.

# **What Is a Clean Code?**

* Bob Bower: What do I think clean code is? This book will tell you, in hideous detail, what I and my compatriots think about clean code. own schools of thought and gather students to learn from them. We see Jeet Kune Do, founded and taught by Bruce Lee in the United States

# **Chapter 2: Meaningful Names**

## **Use Intention-Revealing Names**

* The most important details in this text are the importance of choosing good names and changing them when you find better ones. Choosing good names should answer all the big questions, such as why it exists, what it does, and how it is used. Choosing names that reveal intent can make it easier to understand and change code.

## **Avoid Disinformation**

* Programmers must avoid leaving false clues that obscure the meaning of code. They should avoid words whose meanings vary from their intended meaning, such as hp, aix, and sco. Additionally, they should not refer to a grouping of accounts as an accountList unless it's actually a List. Finally, they should use names which vary in small ways. The reader may think this a contrivance, but the author of the code suggested using a different font so that the differences were more obvious, a solution that would have to be passed down to all future developers as oral tradition or in a written document.

## **Make Meaningful Distinctions**

* When programmers only write code to please a compiler or interpreter, they run the risk of creating issues. In contrast to purposeful naming, number-series naming (a1, a2,.. aN) conceals the author's intention. Indistinct noise words include the letters a, an, and the. The names of the arguments should be source and destination. The most crucial point is that it is recommended to distinguish meaningfully between local variables and function parameters by using prefix conventions like a and the.
* Variable names should never contain noise words, and distinctions between names should be made clear enough for the reader to understand what the differences give. This is demonstrated with a program in which the variables moneyAmount, customerInfo, accountData, and message are all interchangeable with respect to money, customers, accounts, and messages, respectively.

## **Use Pronounceable Names**

* Words are pronounceable by definition, and people are skilled at using them. This is crucial because discussing programming in public makes you appear stupid. Programming is a social activity. My acquaintances at one workplace go about saying "gen why emm dee aich emm ess," which stands for generation date, year, month, day, hour, minute, and second. I began saying "gen-yah-mudda-hims" since I have the unpleasant tendency of speaking everything exactly as it is written. It was amusing since we understood the humor when it was later referred to as this by a large number of designers and analysts. The variables had to be taught to new developers, and when they did, they used ridiculous, invented jargon rather than correct English terminology.

## **Use Searchable Names**

* The difficulty of finding single-letter names and numeric constants inside a body of text is a special issue. Since e is the most frequent letter in the English language and is likely to appear in any program's text, it is a terrible option for any variable that a programmer would need to look for. Any searchable name triumphs over a constant in code, and longer names triumph over shorter ones. A variable or constant should have a name that is appropriate for its scope, and if it may appear or be used more than once across a body of code, it must have a name that is easy to search for.

## **Avoid Encodings**

* When attempting to solve a problem, encoding type or scope information into names adds unnecessary cognitive load. Rarely pronounceable and simple to type incorrectly, encoded names. In addition to learning the body of code they will be working in; it is not reasonable to require new hires to learn yet another encoding "language."

### Hungarian Notation

* + - BASIC only permitted a letter plus one number, but Fortran imposed encodings by making the first letter a code for the type. When everything was an integer handle, long pointer, or void pointer in the Windows C API, Hungarian Notation (HN) was seen as crucial. Modern languages, on the other hand, tend to have fewer classes and shorter functions as well as compilers that remember and enforce the types. Since Java objects are tightly typed and editing environments have progressed, they can now identify type errors long before you can execute a compilation, Java programmers don't require type encoding. HN and other types of type encoding are essentially obstacles that make it more difficult to alter the name or type of a variable, function, or class and increase the risk that the reader may be misled by the encoding scheme.

### Member Prefixes

* + - The most significant information is that you should no longer precede member variables with m\_ and that you should utilize an editing environment to highlight or colorize members to make them stand out. In addition, users rapidly pick up on ignoring the prefix (or suffix) in order to view the important portion of the name, thus prefixes become unnoticed clutter and a sign of outdated code.

### Interfaces and Implementations

* + - The fact that encodings are a particular instance of encodings is the most crucial information in this essay. To create shapes, for instance, you should name your ABSTRACT FACTORY IShapeFactory and ShapeFactory rather than ShapeFactoryImp or CShapeFactory. This is so that users won't be aware that the implementation is a concrete class and the interface is only an interface. Encoding the interface is better than calling it ShapeFactoryImp or even the horrible CShapeFactory.

## **Avoid Mental Mapping**

* The text's most important aspect is that readers shouldn't have to mentally convert their own names into those of other people they are familiar with. Use of neither problem domain words nor solution domain terms results in the issue at hand. Since they are only placeholders that the reader must associate with the underlying notion, single-letter variable names are often a bad option. Smart people occasionally like exercising their mental dexterity by performing mental gymnastics, but professionals put their skills to use by writing code that is easy to comprehend.

## **Class Names**

* Avoid using terms like manager, processor, data, or info in the names of classes and objects.

## **Method Names**

* The names of the methods should be verbs or verb phrases, such as postPayment, deletePage, or save. According to the javabean standard, accessors, mutators, and predicates should be given names that reflect their values and be prefixed with get, set, and is. Use static factory methods with names that explain the parameters when constructors are overloaded. Consider making the appropriate constructors private to impose restrictions on their usage.

## **Don’t Be Cute**

* Only those who understand the author's sense of humor should find the names memorable for the duration of the joke. It is preferable to prioritize cuteness in code above entertainment value. Avoid using slang or colloquialisms, as well as culturally specific gags like "eatMyShorts()" to denote "abort." Speak your mind.

## **Pick One Word per Concept**

* Choose one term and use it consistently for each abstract idea. Modern editing programs like IntelliJ and Eclipse offer context-sensitive hints, such as a list of the methods you may call on a particular object, but they don't offer comments surrounding function names and argument lists. To recall which phrase was used, it's crucial to keep in mind who wrote the library or class—whether it was a firm, a group, or a person. The benefit of a uniform language is enormous for programmers who have to utilize your code.

## **Don’t Pun**

* Using the same term to describe two distinct concepts should be avoided as it is effectively a pun. For instance, the "one word per notion" guideline may result in several classes having the same add function but various interpretations. Authors should use the well-known paperback paradigm, in which it is the author's responsibility to communicate clearly and not the scholar's, in order to make code simpler to grasp.

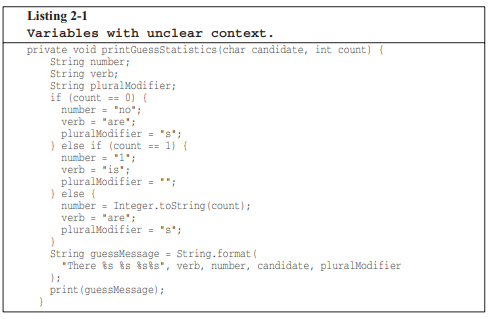
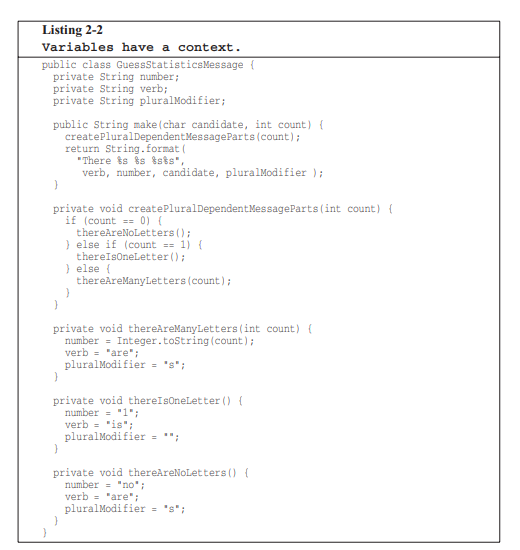
## **Use Solution Domain Names**

* The most important information in this article is that while developing code, programmers should utilize technical words from computer science (CS), such as algorithm names, pattern names, arithmetic terms, and so on. It is crucial to utilize technical names for technical concepts since they are acquainted with the VISITOR pattern and can aid programmers in understanding the idea. Examples of such names are AccountVisitor and JobQueue. The best line of action is to assign such objects technical names.

## **Use Problem Domain Names**

* A smart programmer and designer must be able to distinguish between the notions of the issue and the solution domains. Use the term from the issue domain when there is no "programmer-eese" for what you are doing. Names for code that are more closely related to notions from the issue domain should be taken from it.

## **Add Meaningful Context**

* It's important to put names in the reader's context while creating code. This can be achieved either by prefixing them or by encapsulating them in appropriately named classes, functions, or namespaces. You may add context by using prefixes like addrFirstName, addrLastName, addrState, and so on if your variables have names like firstName, lastname, street, houseNumber, city, state, and zipcode, for instance. Creating a class called Address is a better option so that even the compiler is aware that the variables are a part of a larger idea. Think about the approach in Listing 2-1, where the context must be inferred despite the variables being a part of the "guess statistics" message. 
* Make the three variable fields of the class uessStatisticsMessage a part of the GuessStatisticsMessage. This gives the three variables a clear context and enables the method to be cleaned up by decomposing it into several smaller functions. Refer to Listing 2-2. 

**Don’t Add Gratuitous Text**

* Prefixing every class with GSD in a fictitious program named "Gas Station Deluxe" is a horrible idea. This is so that the IDE can't help as easily. Do you utilize GSDAccountAddress if you created a MailingAddress class in the GSD accounting module and you want a postal address for your customer compliance application? Ten of the 17 characters are unnecessary or unimportant. In general, shorter names are preferable than longer ones as long as they are understandable and don't provide any more content. If you need to distinguish between MAC addresses, port addresses, and Web addresses, PostalAddress, MAC, and URI are better options than Address for class names.