**Clean Code**

**CHAPTER 1**

**There Will Be Code**

* Code will be never got rid because it represents the details of the requirements
* Programming – specifying requirements that a machine can execute
* Code – specification
* All specifications written in higher level and domain-specific language is code
* Creating a machine that will create what we want is impossible
* Well-specified requirements are as formal as code
* Code is the language where we express the requirements

**Bad Code**

* Good code matters because we are struggling with its lack
* Wading- felt impediment because of the bad code
* Programmers will experience impediment by a bad code.
* Rushing a code will be everyone’s experience
* Programmers fixing it later will always equal to never

**The Total Cost of Owning a Mess**

* A team that moving very fast at the beginning of a project might find themselves moving slower pace in the future
* As the mess builds, productivity of the team continues to decrease
* Pressure to increase productivity might result into more messes

**The Grand Redesign in the sky**

* Redesigning the whole project
* Keeping code clean is a matter of professional survival

**Attitude**

* We are at fault from making our good code into a bad one
* We share a great deal of the responsibility for any failures
* Programmers’ job is to defend the code with equal passion
* It is unprofessional for programmers to bend to the will of managers who don’t understand the risks of making messes.

**The Primal Conundrum**

* The only way to go fast is to keep the code as clean as possible at all times.

**The Art of Clean Code?**

* Its no use to write clean code if we don’t know what it means to have a clean code
* Writing clean code requires the disciplined use of a myriad little techniques through acquiring the sense of “cleanliness”.
* “code-sense” is the key, its to see whether the code is good or bad and applying our disciplined to transform bad code into clean code
* “code-sense” helps programmer choose the best variation and guide him

**What is Clean Code?**

* According to Bjarne Stroustrup, inventor of C++
  + Elegant and efficient
  + Logic should be straightforward
  + Clean code does one thing well.
  + Error handling should be complete
* According to Bjarne, a clean code is pleasing to read
* Bad code tempts the mess to grow
* Pragmatic Dave Thomas and Andy Hunt: “A building with broken windows looks like nobody cares about it”
* Bad code tries to do too much
* According to Grady Booch, author of Object Oriented Analysis and Design with Applications:
  + Simple and direct
  + Reads like well-written prose
  + Never obscures the designer’s intent
  + Full of crisp abstractions and straightforward lines
* Clean code should clearly expose the tensions in the problem to be solved
* Code should be matter-of-fact as opposed to speculative
* According to “Bid” Dave Thomas, founder of OTI:
  + Can be read and enhanced by a developer other than its original author.
  + Has unit and acceptance tests.
  + Has meaningful names
  + Minimal dependencies
  + Clear and minimal API
  + Should be literate
* Smaller is better
* Literate – code should be composed in sch a form as to make it readable by humans.
* According to Michael Feathers, author of Working Effectively with Legacy Code:
  + Looks like it was written by someone who cares.
  + There is nothing obvious that you can do to make it better.
* According to Ron Jeffries, author of Extreme Programming Installed and Extreme Programming Adventures in C#
  + Runs all the tests;
  + Contains no duplication
  + Expresses all the design ideas that are in the system
  + Minimizes the number of entities such as classes, methods and the like
  + Focus mostly on duplication
  + Meaningful names
  + expressiveness
  + Extract Method refactoring on it, one method that says more clearly what it does and submethods saying how it is done
  + Wanting a particular item from that collection , wrap the particular implementation in a more abstract method or class
* According to Ward Cunningham, inventor of Wiki:
  + Each routine you read turns out to what much you expected.
  + The code also makes it look like the language was made for the problem
  + It should be obvious, simple and compelling.

**Schools of Thought**

We are Authors

* Rightness within a school does not invalidate the teachings of a different school
* Next time you write a line of code, remember you are an author writing for readers who will judge your effort.
* The ratio of time spent reading vs writing is well over 10:1
* Making it easy to read actually makes it easier to write

**The Boy Scout Rule**

* Writing code well is not enough, it has to be kept clean over time
* “Leave the campground cleaner than you found it”
* Cleanup doesn’t have to be something big.

**Prequel and Principles**

* Sporadic references to various principles of design

**Chapter 2**

**Use Intention-Revealing Names**

* Choosing good names takes time but saves more than it takes.
* Take care with names and change it with better one if possible.
* Names should answer all the big questions: why it exists, what it does, how it is used.

**Avoid Disinformation**

* By giving concepts names, the code can improve considerably
* Avoid leaving false clues that obscure the meaning of code
* Avoid words whose entrenched meanings vary from our intended meaning
* Beware of using names which vary in small ways

**Make Meaningful Distinctions**

* Programmers create problems for themselves when they write code to satisfy the compiler or interpreter
* Number-series naming are noninformative
* Noise words are redundant

**Use Pronounceable Names**

* If you cant pronounce it, you cant discuss it without sounding like an idiot

**Use Searchable Names**

* Single-letter names and numeric constants have particular problem, they are not easy to locate across a body of text
* Single-letter names can only be used as local variables on short methods
* The length of a name should correspond to the size of its scope

**Avoid Encodings**

* If variable or constant might be seen or used in multiple places , give it a search-friendly name
* Encoding type or scope information into names adds an extra burden of deciphering
* Encoded names are seldom pronounceable and easy to mis-type

**Hungarian Notation**

* Considered to be pretty important back in the Windows C API.
* Compiler did not check types before so programmers needed a way to remember the types
* Nowadays HN and other forms of type encoding are simply impediments.
* Create a possibility that the encoding system will mislead the reader

**Member Prefixes**

* Prefix member is not necessary nowadays especially when there are environment that highlights or colorizes members to make them distinct.

**Interfaces and Implementations**

* Abstract Factory- creation of shapes
* This will become interface and will be implemented by a concrete class

**Avoid Mental Mapping**

* Readers shouldn’t have to mentally translate your names into other names they already know
* In other contexts ,a single-letter name is a poor choice
* Difference between smart programmer and professional programmer is that the professional understands that clarity is king

**Class Names**

* Classes and objects should have noun or noun phrase names like Customer, WikiPage
* A class name should not be a verb

**Method Names**

* It should have verb or verb phrase names like postPayment, deletePage or save
* Accessors, mutators and predicates should be named for their value and prefixed with get , set and is
* When constructors are overloaded, use static factory methods

**Don’t Be Cute**

* If names are too clever, they will be memorable only to people who share the same humor of the author.
* Choose clarity over entertainment value
* Say what you mean. Mean what you say

**Pick One Word per Concept**

* pick one word for one abstract concept and stick with it
* The function names have to stand alone and consistent
* A consistent lexicon is a great boon to the programmers who must use your code

**Don’t Pun**

* Avoid using the same word for two purposes
* Using same term for multiple ideas is a pun

**Use Solution Domain Names**

* Remember that the people who read your code will be programmers
* Choosing technical names for those things is usually the most appropriate course

**Use Problem Domain Names**

* The programmer who maintains your code can ask a domain expert what it means.
* Separating solution and problem domain concepts is part of the job of a good programmer and designer

**Add Meaningful Context**

* Place names in context for your reader by enclosing them in well-named classes, functions, or namespaces.
* Add context by using prefixes

**Don’t Add Gratuitous Context**

* Shorter names are generally better than longer ones
* Don’t add more context to a name than is necessary