**Clean Code Summary of Chapter 1 – 2**

**Chapter 1: Clean Code**

**There will be Code**

* All the specification written in these higher level is and domain specific language is code.
* Remember that code is really the language in which we ultimately express the requirements.
* But we will never eliminate necessary precision—so there will always be code.

**Bad Code**

* You’re coding in a rush
* Your boss is pushing you to finish soon
* You don’t have time to do a good job
* You want to get a quick fix out and tell yourself you’ll fix it later. Later equals never.

According to Bjarne Stroustrup, I like my code to be *elegan*t 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. So Bjarne highlighted the word elegant and efficient.

According to Grady Booch, Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designer’s intent but rather is full of crisp abstractions and straightforward lines of control. So Grady emphasis the word simple and well-written.

According to “Big” Dave Thomas, Clean code can 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 for 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. So for “Big” Dave Thomas emphasis the read or readable.

According to Michael Feathers, I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares. There is nothing obvious that you can do to make it better. All of those things were thought about by the code’s author, and if you try to imagine improvements, you’re led back to where you are, sitting in appreciation of the code someone left for you—code left by some- one who cares deeply about the craft. Michael hit it on the head. Clean code is code that has been taken care of. Someone has taken the time to keep it simple and orderly. They have paid appropriate attention to details. They have cared.

**Clean Code should be:**

* Elegant
* Efficient
* Readable
* Simple
* Well-written
* Without duplication

**Authors**

* Are responsible well with readers

**The Boy Scout Rule**

* It’s not enough to write code well
* The code has to be kept clean over time
* Leave the campground cleaner than you found it

**Chapter 2: Meaningful Names**

**Use Intention-Revealing Names**

* The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.
* We should choose a name that specifies what is being measured and the unit of that measurement:

For example:

*int d; // elapsed time in days -* The name d reveals nothing

*int elapsedTimeInDays;*

**Avoid Disinformation**

* Programmers must avoid leaving false clues that obscure the meaning of code.

For example:

hp, aix and sco are all poor variable names and it leads you to the disinformation.

**Use Pronounceable Names**

* Are words, by definition and pronounceable.

For example:

class DtaRcrd102 {

private Date genymdhms;

private Date modymdhms;

private final String pszqint = "102";

/\* ... \*/

};

to this…

class Customer {

private Date generationTimestamp;

private Date modificationTimestamp;;

private final String recordId = "102";

/\* ... \*/

};

**Use Searchable Names**

* If a variable or constant might be seen or used in multiple places in a body of code, it is imperative to give it a search-friendly name. So that it is easy for you to locate.

**For example**

for (int j=0; j<34; j++) {

s += (t[j]\*4)/5;

}

to

int realDaysPerIdealDay = 4;

const int WORK\_DAYS\_PER\_WEEK = 5;

int sum = 0;

for (int j=0; j < NUMBER\_OF\_TASKS; j++) {

int realTaskDays = taskEstimate[j] \* realDaysPerIdealDay;

int realTaskWeeks = (realdays / WORK\_DAYS\_PER\_WEEK);

sum += realTaskWeeks;

} **H**

**Member Prefixes**

**Class Names**

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

**Method Name**

* Methods should have verb or verb phrase names like postPayment, deletePage, or save.
* Accessors, mutators, and predicates should be named for their value and prefixed with get, set.

For example:

string name = employee.getName();

customer.setName("mike");

if (paycheck.isPosted())...

**Don’t Be Cute**

* Cuteness in code often appears in the form of colloquialisms or slang.

For example:

Don’t use the name whack() to kill()

**Pick One Word per Concept**

* Pick one word for one abstract concept and stick with it.

**Don’t Pun**

* Avoid using the same word for two purposes.
* Using the same term for two different ideas is essentially a pun.

**Use Solution Domain Names**

* Remember that the people who read your code will be programmers.
* So go ahead and use computer science (CS) terms, algorithm names, pattern names, math terms, and so forth.

**Use Problem Domain Names**

* Use the name from the problem domain.
* The code that has more to do with problem domain concepts should have names drawn from the problem domain.

**Add Meaningful Context**

* You can add context by using prefixes: addrFirstName, addrLastName and addrState.

**Don’t Add Gratuitous Context**

* an imaginary application called “Gas Station Deluxe,” it is a bad idea to prefix every class with GSD.