Faculty Of Engineering
Computers and Systems Dep.

QUICK PRESENTATION ON :

Clean Code

PREPARED BY :

AHMED MAGDY EBRAHIM	504015
ABD AL-RAHMAN SAAD	504061
MOHAMED ASHRAF HAMZA	504094
MOHAMED SALAH SAYED	504102

REFRENCES

- A handbook of Agile Software Craftsmanship
  by Robert C. Martin Series (Uncle Bob)

- Summary of 'Clean code'
  https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

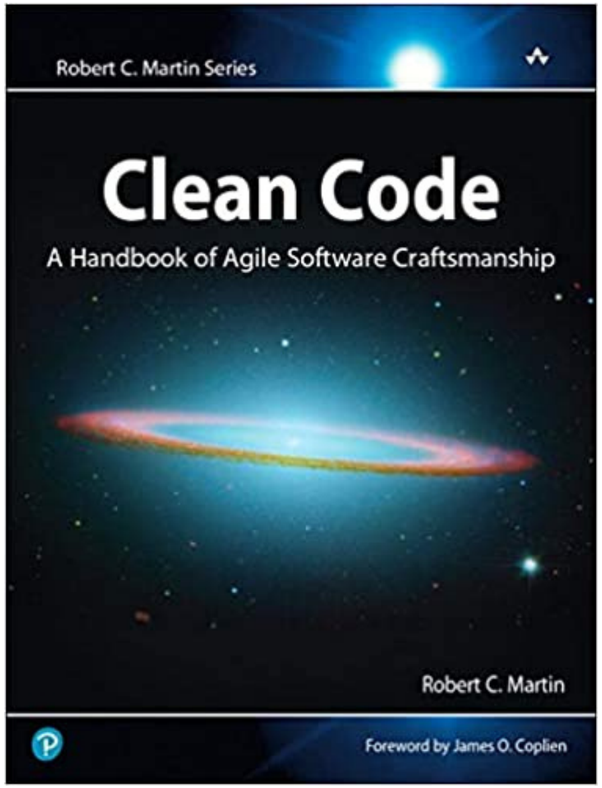
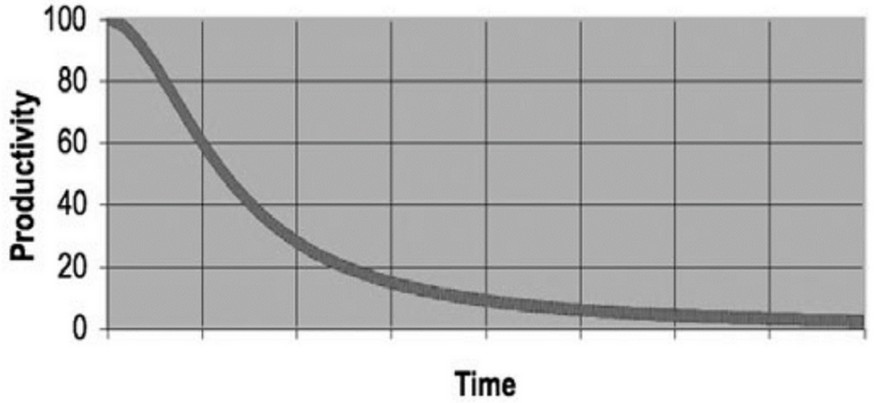


TABLE OF CONTENT

BAD CODE

► Productivity:



As the mess builds, the productivity of the team continues to decrease.As productivity decreases, management does the only thing they can; they add more staff to the project in hopes of increasing productivity.

They don’t know the difference between a change that matches the design intent and a change that thwarts the design intent.Furthermore, they, and everyone else on the team, are under horrific pressure to increase productivity. So, they all make more and more messes, driving the productivity ever further toward zero.

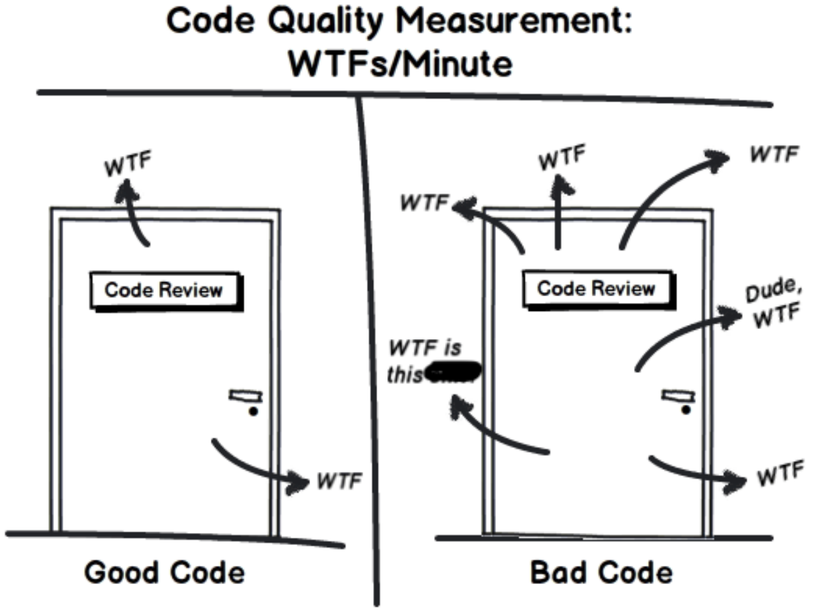
► Redesign:

Eventually the team rebels. They inform management that they cannot continue to develop in this odious code base. They demand a redesign.

CLEAN CODE

Bjarne Stroustrup, inventor of C++

" 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. "



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

" 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 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. "

▶ READABLE

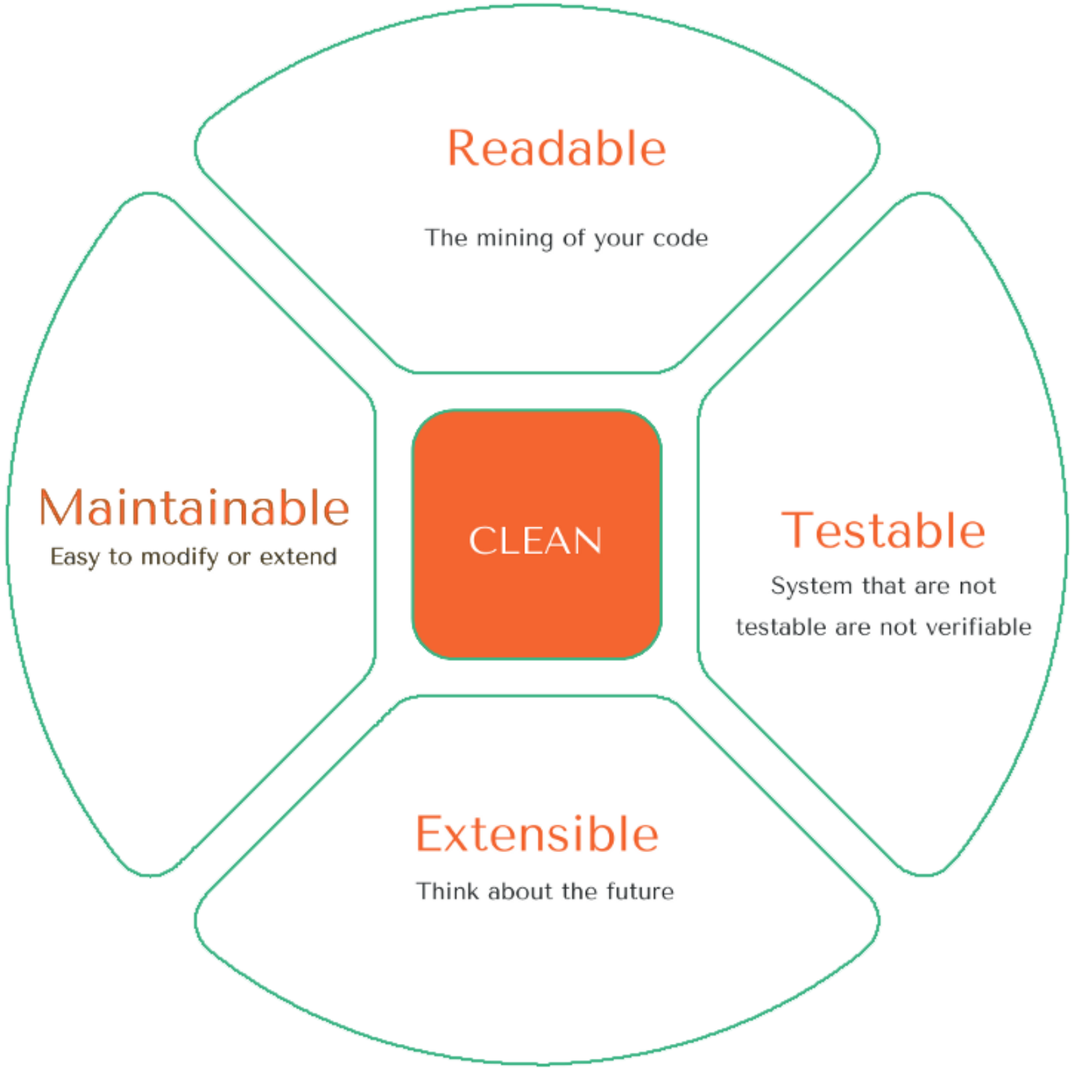
Readable code is code that can be understood by both a programming newbie and a professional developer equally. It is a code that is not time-consuming to understand and use

▶ MAINTAINABLE

If the code is well written, it’s easier to test, find bugs, and fix them for everyone working on the code-base.

▶ EXTENSIBLE

If the code-base is stable, clean, and well-written, the process of scaling the code and extending upon it will be more straightforward.



```php
class Set
{
    private $items = [];

    public function isEmpty()
    {
        if ($this->size() === 0) {
            return true;
        } else {
            return false;
        }
    }

    public function add($item)
    {
        if ($item !== null) {
            if (!$this->contains($item)) {
                $this->items[] = $item;
            }
        }
    }

    public function contains($item)
    {
        if (!$this->isEmpty()) {
            if (in_array($item, $this->items, true)) {
                return true;
            }
```

Conditionals which return "raw" booleans can be collapsed

Deeply nested conditionals can bury the codes primary path

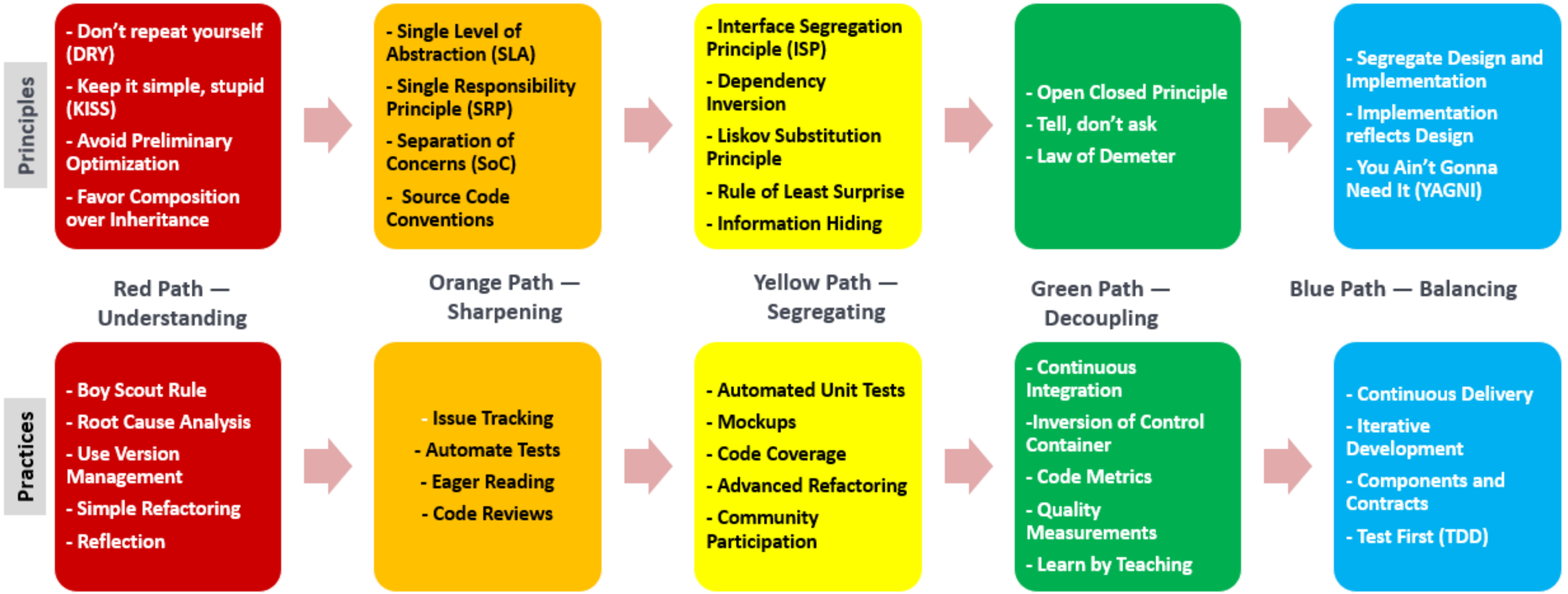
```php
class Set
{
    private $items = [];

    public function isEmpty()
    {
        return $this->size() === 0;
    }

    public function add($item)
    {
        if ($item === null || $this->contains($item)) {
            return;
        }

        $this->items[] = $item;
    }

    public function contains($item)
    {
        return in_array($item, $this->items, true);
    }

    // ...
}
```

Directly return the condition

Collapse the conditions to form a "guard clause" which calls out exceptional paths and brings the primary path back to the top level

Collapsing also reveals unnecessary conditions (i.e. checking for emptiness)

PRINCAPLES & PRACTICES



KISS: KEEP IT SIMPLE, STUPID.

This term came from the US navy back in the 60s. Like the name says, this principle insists on keeping things simple. You should not have to complicate things by adding a piece of complex code, when a simple one line code could effectively do the job.

DRY: DON'T REPEAT YOURSELF

This principle clearly states to avoid duplication of data and logic. And provides an alternative for the same and that is finding logic in repetitions.

YAGNI: YOU AREN'T GUNNA NEED IT

YAGNI states that you do not need to add a code for something that isn't needed today. Coding for additional features that would never be used by the project is simply a waste of time, so why do it?

COI: COMPOSITION OVER INHERITANCE

This principle tells you to design your types over what they do instead of over what they are. because inheritance forces you to build a taxonomy of objects early on in a project, making your code inflexible for changes later on.

MEANINGFUL NAMES

```
int d; // elapsed time in days
```

```
int daysSinceCreation;
```

- The hardest thing about choosing good names is that it requires good descriptive skills
- This is a teaching issue rather than a technical, business, or management issue. As a result many people in this field don't learn to do it very well. Most of the time we don't really memorize the names of classes and methods.

▶ USE INTENSION-REVEALING NAMES

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

▶ USE PRONOUNCEABLE NAMES

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

VS

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /* ... */
};
```

▶ USE SEARCHABE NAMES

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

VS

```
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;
}
```

▶ AVOID DISINFORMATION

Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning. For example, hp, aix, and sco would be poor variable names .

▶ AVOID MENTAL MAPPING

Readers shouldn't have to mentally translate your names into other names they already know, One difference between a smart programmer and a 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, Account, and AddressParser.

▶ METHOD NAMES

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, and is according to the javabean standard.

FUNCTION RULES

► In the early days of programming we composed our systems of routines and subroutines.
Then, we composed our systems of programs, subprograms, and functions.
Nowadays only the function survives from those early days. Functions are the first line of organization in any program.

► So what is it that makes a function easy to read and understand?
How can we make a function communicate its intent?
What attributes can we give our functions that will allow a casual reader to intuit the kind of program they live inside?

FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.

SOME FUNCTIONS RULES

- Small
  - Blocks and Indenting
- Do One Thing / Single Responsibility
  - Sections within Functions
- One Level of Abstraction per Function
  - The Stepdown Rule
- Switch Statements
- Have No Side Effects
  - Output Arguments

SOME FUNCTIONS RULES

- Use Descriptive Names
- Function Arguments [testing]
  - Flag Arguments
  - Argument Objects [x, y -> center]
  - Argument Lists
  - Verbs and Keywords
- Command Query Separation [do something or answer something]
- Don’t Repeat Yourself
- Structured Programming / one entry one exit [break, continue, goto]
- Prefer Exceptions to Returning Error Codes

ERROR HANDILING

► It might seem odd to have a section about error handling in a book about clean code. Error handling is just one of those things that we all have to do when we program.

► Input can be abnormal and devices can fail. In short, things can go wrong, and when they do, we as programmers are responsible for making sure that our code does what it needs to do. The connection to clean code, however, should be clear. Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, but if it obscures logic, it's wrong.

► In this chapter I'll outline a number of techniques and considerations that you can use to write code that is both clean and robust—code that handles errors with grace and style.

ERROR HANDILING

1. Use Exceptions Rather Than Return Codes
2. Write Your Try-Catch-Finally Statement First
3. Use Unchecked Exceptions
4. Provide Context with Exceptions
5. Define Exception Classes in Terms of a Caller’s Needs
6. Define the Normal Flow
7. Don’t Return Null
8. Don’t Pass Null

USEFUL TOOLS

1.Sonar Lint

2.CheckStyle

3.PMD

4.Synk

5.SonarQube

6.OWASP Dependency-Check

7.JaCoCo

8.nohttp