CSE 4513

Lec – 10

Clean Coding

CODING

Developers …..by definition **they code**.

For some of them it's just **a job**, for others it is **a hobby**, **a craft** or even **an art** !

You need **concentration** and **focus** to achieve the 4 main goals of coding :

1.Your code must work !

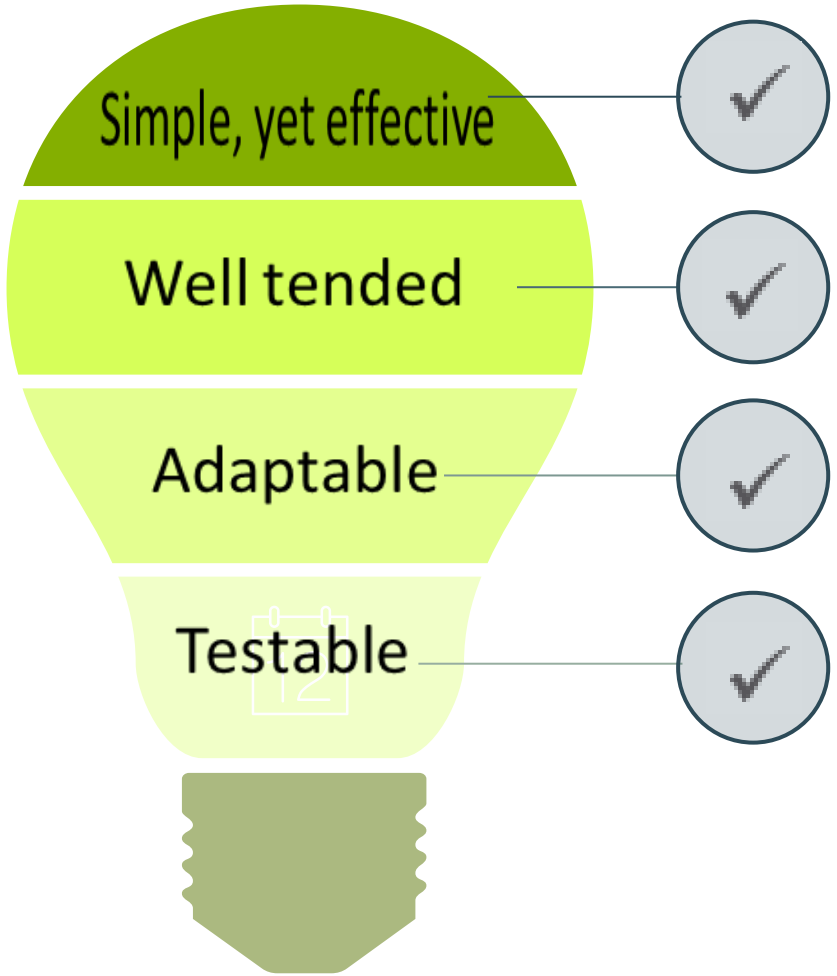
2.Your code must solve the problem !

3.Your code must fit well into the existing system !

4.Your code must be readable by other programmers!(Clean Code)

What exactly is Clean Code?

”software code that is formatted correctly and in an organized manner so that another coder can easily read or modify it”.



solves the problem at hand without introducing unnecessary complexity.

like a well-kept garden – time and effort is spent continuously tending it and improving it.

easy for colleagues to understand, update and modify. In addition, it can be adapted when requirements change.

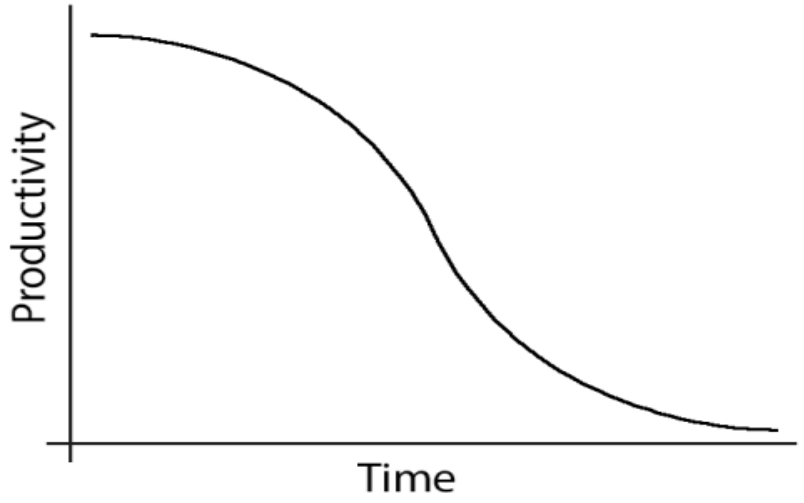
is designed to be testable. need to be covered by tests that can be run automatically. Using tests that can be run as we make changes, we immediately get feedback on whether or not we have broken something.

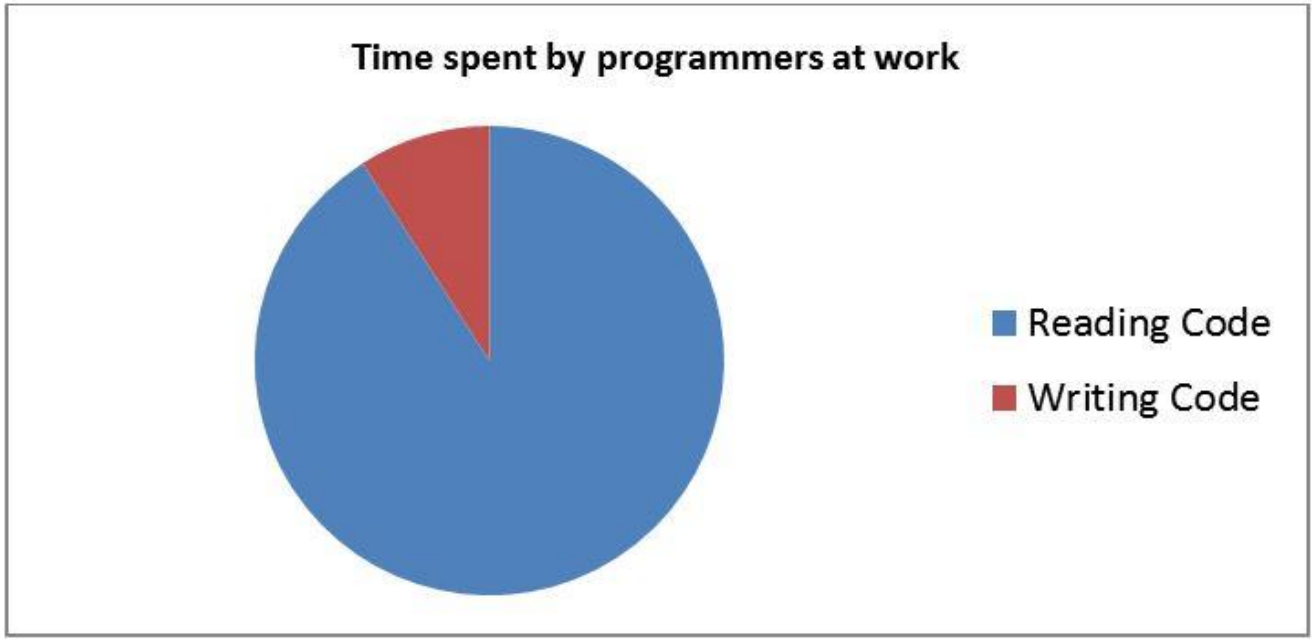
Bad Code – What's that

- Inappropriate Information
- Obsolete Comments/ Redundant Comments / Poorly Written Comments
- Commented out code
- Too many arguments
- Dead functions
- Duplication
- Incorrect Behavior and Boundaries
- Code at wrong level of abstraction
- Base classes depending on their derivatives
- Too much information
- ….

Why is Clean Code Important

- Clean Code saves time
- Maintains fast productivity
- It is costly to own a mess
- Productivity decreases
- To make next developer Happier





What Prevents Clean Code

- Ignorance
- Stubbornness
- Short-Timer Syndrome
- Arrogance
- Job Security

But How do We Do it???

- Naming
- Functions
- Comments rules
- Source code structure
- Objects and data structures
- Error Handling
- Tests
- Code smells

Any fool **can write code** that a **computer can** understand. Good **programmers write code** that humans **can** understand.

~Martin Fowler

NAMING : USE INTENTION-REVEALING NAMES

Bad

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

Good

```
public final static int STATUS_VALUE = 0;
public final static int FLAGGED = 4;

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

NAMING : AVOID DISINFORMATION

- Must avoid leaving false clues that obscure the meaning of code.
- Should avoid words whose entrenched meaning vary from our intended meaning.

Don't refer to a group of accounts as "accountList," whereas it is actually not a List. Instead, name it "accountGroup," "accounts," or "bunchOfAccounts."

Avoid unnecessary encodings of datatypes along with the variable name.

```
String nameString;

Float salaryFloat;
```

NAMING : MAKE MEANINGFUL DISTINCTION

- If there are two different things in the same scope, you might be tempted to change one name in an arbitrary way.

```
String cust;
String customer;
```

What is stored in cust, which is different from customer?

```
class ProductInfo
class ProductData
```

How are these classes different?

```
void getActiveAccount()
void getActiveAccounts()
void getActiveAccountsInfo()
```

How do you differentiate between these methods in the same class?

suggestion here is to make meaningful distinctions.

Naming : Use Pronounceable Names

Ensure that names are pronounceable. Nobody wants a tongue twister.

Bad

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

Good

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

NAMING : USE SEARCHABLE NAMES

- Avoid single letter names and numeric constants, if they are not easy to locate across the body of the text.

```
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 = 0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
}
```

Sum here is not really useful but at least is searchable.
It will be easier to find *WORKS_DAYS_PER_WEEK* than to find all the places where 5 was used.

NAMING : DON'T BE CUTE/DON'T USE OFFENSIVE WORDS

If names are too clever, they will be memorable only to people who share your sense of humor, and only as long as these people remember the joke.

Will they know what the function named `HolyHandGrenade` is supposed to do?

Sure, it's cute, but maybe in this case `DeleteItems` might be a better name.

Don't tell little culture-dependent jokes like `eatMyShorts()` to mean `abort()`.

Say what you mean. Mean what you say.

Naming : Pick One Word per Concept

Use the same concept across the codebase.

FetchValue() vs GetValue() vs RetrieveValue()

If you are using **fetchValue()** to return a value of something, use the same concept; instead of using **fetchValue()** in all the code, using **getValue(), retrieveValue()** will confuse the reader.

dataFetcher() vs dataGetter() vs dataFetcher()

If all three methods do the same thing, don't mix and match across the code base. Instead, stick to one.

NAMING : ADD MEANINGFUL CONTEXT

- Most names are not meaningful.
- you have to place names in context for your reader by enclosing them in well-named classes, functions, or namespaces.
- If not possible, prefixing the names may be necessary as a last resort.

For example, the member variable named "state" inside a address class shows what it contains. But the same "state" may be misleading if used to store "Name of The State" without context. So, in this scenario, name it something like "addressState" or "StateName."

The "state" may make more sense in a context where you are coding about addresses when named "addrState."

More on Naming

- Use Nouns for Variables, Properties, Parameters
  - ✓ indexer, currentUser, PriceFilter
- Use Verbs for Methods and Functions
  - ✓ SaveOrder(), getDiscounts(), RunPayroll()
- Pronounceable and Unambiguous
  - ✓ recdptrl = received patrol? record department role?

NAMING : SUMMARY

1. Meaningful names
2. Avoid Disinformation
3. Avoid small variations in names – s1, s2
4. Meaningful Distinctions
   - if (!fundsAvailable || availableFunds < -0.004m)
5. Pronounceable Names - rptparam
6. Searchable Names – MAX_DAYS instead of 999
7. Avoid Encodings – e.g. Hungarian notation strName, strFName, Compiler
8. Function Names are verbs
9. Data fields are nouns

COMMENTS

- Rule #1: Comments lie
  - ✓ Code is updated or moved, but not the comments.

COMMENTS

- Rule #2: Comments do not make up for bad code
  - ✓ If the code is that unclear, rewrite the code
- Rule #3: Explain yourself in Code

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

Should be

```
if (employee.isEligibleForFullBenefits())
```

GOOD COMMENTS

- Legal Comments
  - ✓ We should write some legal comment like this to help another person know where and when the code come from

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All
rights reserved. // Released under the terms of the GNU
General Public License version 2 or later.
```

GOOD COMMENTS

- Explanation of Intent

```
assertTrue(a.compareTo(a) == 0); // a == a
assertTrue(a.compareTo(b) != 0);  // a != b
assertTrue(ab.compareTo(ab) == 0); // ab == ab
```

GOOD COMMENTS

- Warning of Consequence

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
//SimpleDateFormat is not thread safe,
//so we need to create each instance independently.
SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy
HH:mm:ss z");
df.setTimeZone(TimeZone.getTimeZone("GMT"));
return df;
}
```

GOOD COMMENTS

- TODO comments
  - ✓ We should write TODO comment for note something that we will do in function

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo makeVersion() throws Exception
{
return null;
}
```

GOOD COMMENTS

- Amplification
  - ✓ A comment may be used to amplify the importance of something that may otherwise seem insignificant.

```
String listItemContent = match.group(3).trim();
// the trim is real important. It removes the starting
// spaces that could cause the item to be recognized
// as another list.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

BAD COMMENTS

- Mumbling
  - ✓ Placing in a comment just because you feel you should or because the process requires it, is a hack.
  - ✓ If you decide to write a comment, then spend the time necessary to make sure it is the best comment you can write.Example:

```
public void loadProperties() {

  try {
    String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
    FileInputStream propertiesStream = new FileInputStream(propertiesPath);
    loadedProperties.load(propertiesStream);
  }
  catch(IOException e) {
    // No properties files means all defaults are loaded
  }
}
```

What does that comment in the catch block mean?

BAD COMMENTS

- Redundant Comments

```
// Utility method that returns when this.closed is true. Throws an
exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis) throws
Exception {
  if(!closed) {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be closed");
  }
}
```

What purpose does the comment serve?

It's certainly not more informative than the code.

It does not justify the code, or provide intent or reasoning.

It is not easier to read than the code.

Bad Comments

- Mandated Comments
  - ✓ It is just plain silly to have a rule that says that every function must have a javadoc, or every variable must have a comment. Comments like this just mess up the code, propagate lies, and lend to general confusion and disorganization.

```
function hashIt(data) {
  // The hash
  let hash = 0;

  // Length of string
  const length = data.length;

  // Loop through every character in data
  for (let i = 0; i < length; i++) {
    // Get character code.
    const char = data.charCodeAt(i);
    // Make the hash
    hash = (hash << 5) - hash + char;
    // Convert to 32-bit integer
    hash &= hash;
  }
}
```

Bad Comments

- Journal Comments
  - ✓ Remember, use version control! There's no need for dead code, commented code, and especially journal comments. Use git log to get history!

```
/**
 * 2016-12-20: Removed monads, didn't understand them (RM)
 * 2016-10-01: Improved using special monads (JP)
 * 2016-02-03: Removed type-checking (LI)
 * 2015-03-14: Added combine with type-checking (JR)
 */
function combine(a, b) {
  return a + b;
}
```

BAD COMMENTS

- Noise Comments.

```
/**
* Default constructor.
*/
protected AnnualDateRule() {
}
/** The day of the month. */
private int dayOfMonth;
```

The comments in the above examples doesn't provides new information.

- Position Markers

```
// Actions ////////////////////////////////
```

This type of comments are noising

BAD COMMENTS

- Don’t Use a Comment When You Can Use a Function or a Variable

```
// does the module from the global list <mod> depend on the
// subsystem we are part of?
if
(smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Can be

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

BAD COMMENTS

- Closing Brace Comments

```
try {
      while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
        String words[] = line.split("\\W");
        wordCount += words.length;
        } //while
        System.out.println("wordCount = " + wordCount);
        System.out.println("lineCount = " + lineCount);
        System.out.println("charCount = " + charCount);
      } // try
      catch (IOException e) {
        System.err.println("Error:" + e.getMessage());
      } //catch
```

Bad Comments

- Attributions and Bylines

```
/* Added by Rick */
```

The VCS can manage this information instead.

- Commented-Out Code

```
doStuff();
// doOtherStuff();
// doSomeMoreStuff();
// doSoMuchStuff();
```

VCS exists for a reason. Leave old code in your history.

Comment Summary

- ✓ Always try to explain yourself in code.
- ✓ Don't be redundant.
- ✓ Don't add obvious noise.
- ✓ Don't use closing brace comments.
- ✓ Don't comment out code. Just remove.
- ✓ Use as explanation of intent.
- ✓ Use as clarification of code.
- ✓ Use as warning of consequences.

FORMATTING RULES

1. Separate concepts vertically.
2. Related code should appear vertically dense.
3. Declare variables close to their usage.
4. Dependent functions should be close.
5. Similar functions should be close.
6. Place functions in the downward direction.
7. Keep lines short.
8. Don't use horizontal alignment.
9. Don't break indentation.

FUNCTIONS

Several best practices on how to write plain, understandable functions

1. Functions should be small !!! Really
2. Block and indenting
3. Do one thing! (Single Responsibility Principle)
4. Function names should say what they do
5. Remove duplicate code
6. Avoid Side Effects
7. Remove dead code
8. One level of abstraction
9. Low number of arguments
10. Command Query Separation

FUNCTIONS

1. Functions should be small !!! Really

- 1st rule – Function should be small.
- 2nd rule – Functions should be smaller than that!
- Functions should barely be 20/30 lines long…
- a shorter function will do less (only one simple thing).
- Additionally to this, it will be more easily to understand and manage it.

FUNCTIONS

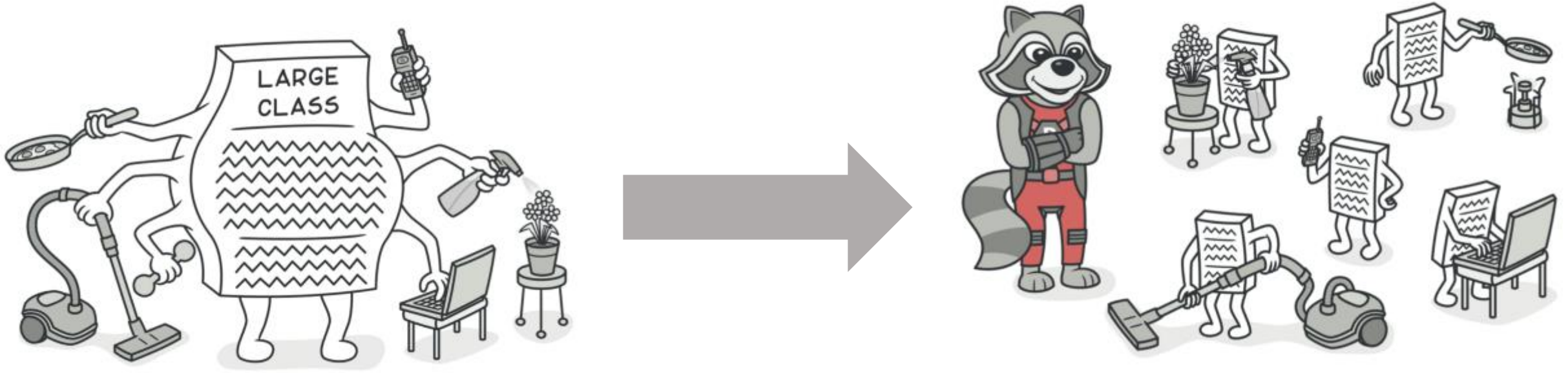
2. Block and indenting

- keep the code blocks inside `if, else` and `while` statements minimal.
- preferably up to one line and that one line should be a function call
- Similarly, we should avoid nesting such structures inside functions by moving the nested logic into another function.
- The indent levels of a function probably shouldn't be more than two tabs!

FUNCTIONS

3. Do one thing! (Single Responsibility Principle)

“Functions should do one thing. They should do it well. They should do it only”.



FUNCTIONS

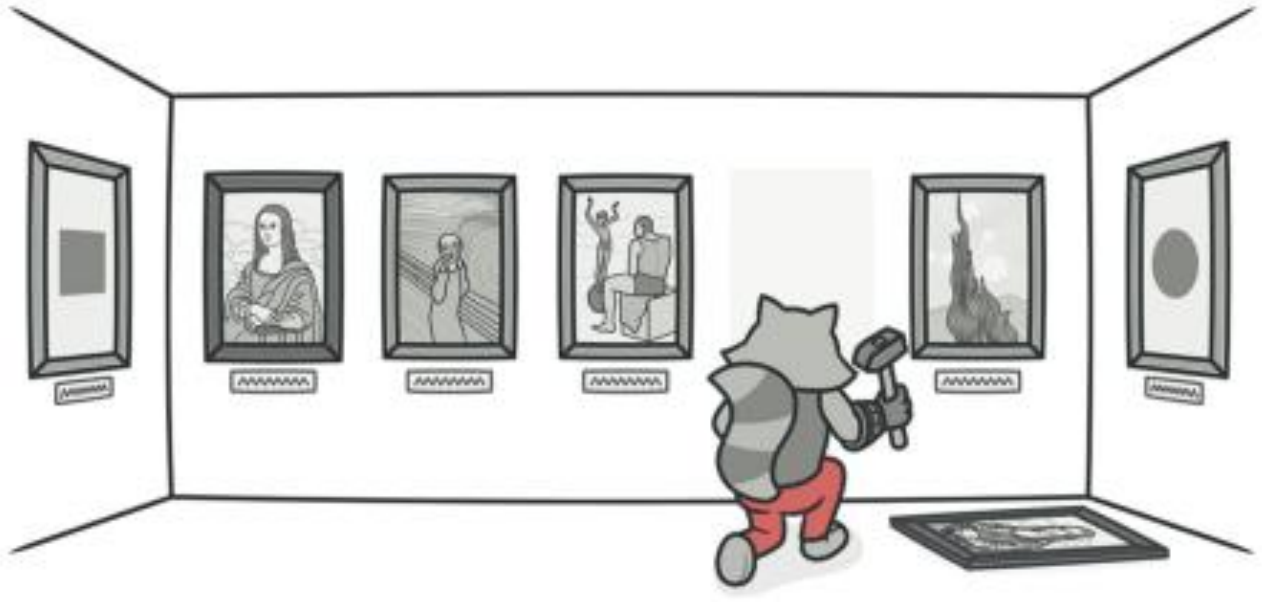
4. Function names should say what they do

- A function‘s name should explain what exactly it is doing. Nothing less, no more.
- The name of a variable, function, or class, should answer the following:
  - ✓ Why it exists?
  - ✓ What does it do?
  - ✓ How it is used?

FUNCTIONS

5. Remove duplicate code

- attempt to prevent duplication of code.
- If the is duplication of code, then you may need to change many places for a single change.



FUNCTIONS

6. Avoid Side Effects

- This is the case when your function is doing more than one thing, but without telling to the client.
- In particular our functions should not create temporal coupling with other function

FUNCTIONS

7. Remove dead code

- Dead code is as poor as double code.
- There’s nothing in the coding base to hold it.
- If you ever need it, it will remain secure in your version history.

FUNCTIONS

8. One level of Abstraction

- a mechanism that can tell us that the function is doing too many things.
- For example a function that processes an entity and also internally starts to split and transfer one of the entity fields has more than one level of abstraction.
- function must contains instruction that are at the same level of abstraction.

FUNCTIONS

9. Low number of arguments

- How much arguments should a perfect clean code function have?
- Zero , monadic (1 argument), dyadic (2 arguments) or triads (3 arguments) functions.
- Functions with more than 3 arguments should be avoided whenever possible.
- When the number of arguments start to growth to more than 3 arguments it is possible wrap them in specific class that describe better their meaning.

```
/// Simple version
Circle makeCircle(double x, double y, double radius);

/// With arguments objects
Circle makeCircle(Point center, double radius);
```

FUNCTIONS

10. Command Query Separation

- Functions should do something or answer something, not both.
- You should never have methods that execute a query and in the same time a command.
- Doing both often leads to confusion.

Exception & Error Handling

- Returning an error code creates two additional things that need to be done by the developer/client.
- Needs to know the mapping of each error code and in the same time he/she has to check the returning error code.
- For these cases, **throwing an exception** is better and will simplify the work of clients.
- Additional to this, the error handing will be 100% separated from your logic.

Error Handling Rules

- Use Exceptions Rather Than Return Codes!

```
int withdraw(int amount) {
  if (amount > _balance) {
    return -1;
  }
  else {
    balance -= amount;
    return 0;
  }
}
```

this method returns a special value that indicates an error

Throw an exception instead.

```
void withdraw(int amount) throws BalanceException {
  if (amount > _balance) {
    throw new BalanceException();
  }
  balance -= amount;
}
```

Error Handling Rules

- Write Your Try-Catch-Finally Statement First
  - ✓ try blocks are like transactions which specify the block of code that might give rise to the exception in a special block with a “Try” keyword.
  - ✓ Your catch has to leave your program in a consistent state, no matter what happens in the try.
  - ✓ For this reason it is good practice to start with a try-catch-finally statement when you are writing code that could throw exceptions.
- Provide Context with Exceptions
  - ✓ Each exception that you throw should provide enough context to determine the source and location of an error.
  - ✓ Create informative error messages and pass them along with your exceptions.
  - ✓ Mention the operation that failed and the type of failure.
  - ✓ If you are logging in your application, pass along enough information to be able to log the error in your catch..

Error Handling Rules

- Don't Return Null
  - ✓ If you are tempted to return null from a method, consider throwing an exception or returning a SPECIAL CASE object instead.
  - ✓ If you are calling a null-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.

- Don't Pass Null
  - ✓ Returning null from methods is bad, but passing null into methods is worse.
  - ✓ Unless you are working with an API which expects you to pass null, you should avoid passing null in your code whenever possible.

FAQ: Exceptions and Error Handling

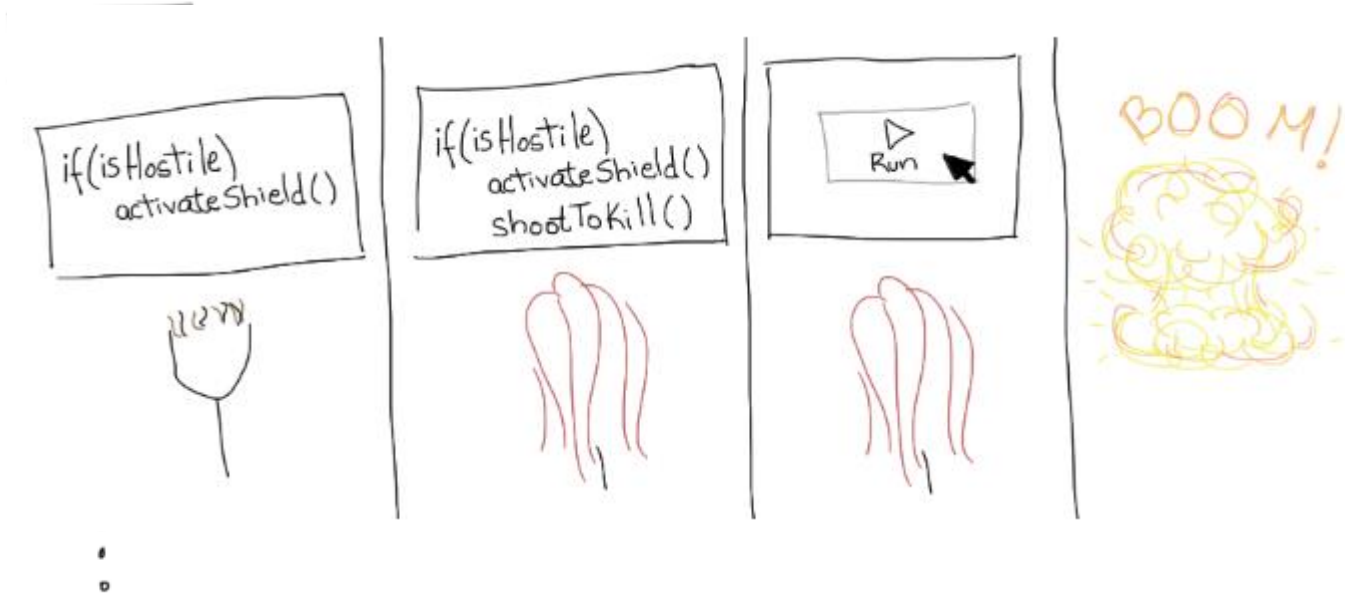
https://isocpp.org/wiki/faq/exceptions

Important contents with good explanation.

You must read it as you may get question form here in your coming exams….

Techniques That Every Coder Should Know

- Keep It Simple
- Understand your Code
- Don't Repeat yourself
- Indent your code
- Naming convention
- Explore
- Use your brain
- Avoid Hard-Coded Strings and Magic Numbers
- Always Write the Braces Around Single Line Blocks
- Good Comments are your best friends
- Test run
- Write Everything Code-Related in English
- Practice your art



Make People Happy with your Code

Through your code, you have the power to affect the life of your fellow programmers:

- ✓ They will **rejoice/panic** when they know they have to work with the code you wrote.
- ✓ They will spend **10 minutes/1 hour** understanding the code you wrote.
- ✓ They will introduce **0 bugs/2 bugs** when they edit the code you wrote.