# Introduction to Programming

Writing Clean Code

It compiles, it runs, it produces the expected output for the business.

Why is it Bad Code?

# Clean Code

- “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.” Grady Booch
- “Clean code can be read, and enhanced by a developer other than its original author.” Dave Thomas
- “Clean code always looks like it was written by someone who cares.” Michael Feathers
- “Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.” Martin Golding

# Why writing clean code is important?

# Why writing clean code is important?

- Developers usually work in teams.
- Maintenance is not always performed by the developers that wrote the initial code.
- Adding features and fixing bugs might be requested after enough time has passed for the developers to forget some parts of the product.
- Maintenance is the longest phase a product passes through.

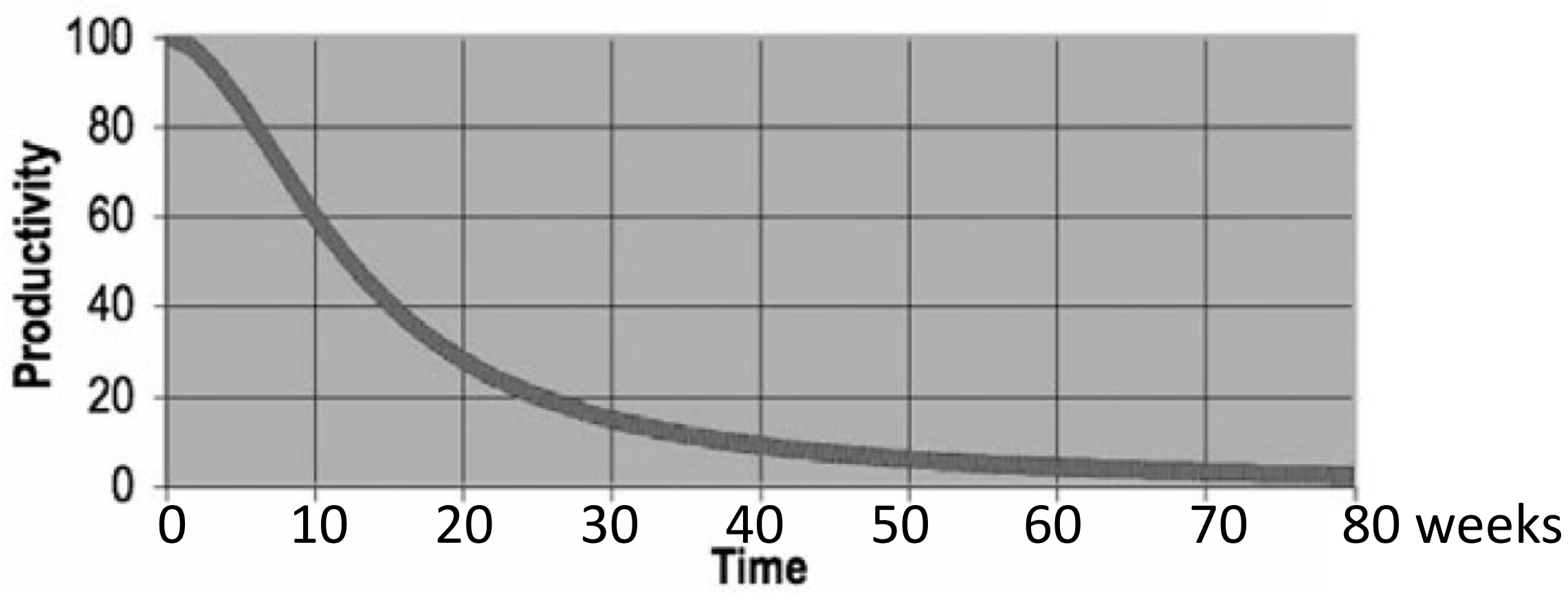
# Why writing clean code is important?

Broken Windows Metaphor:

“A building with broken windows looks like nobody cares about it. So other people stop caring. They allow more windows to become broken. Eventually they actively break them.”

Dave Thomas and Andy Hunt

# Why writing clean code is important?



# Fundamental Areas

- Meaningful names
- Functions
- Comments
- Formatting
- Testing

# Meaningful Names

- variables
- functions
- arguments
- classes
- packages
- source files
- and the directories that contain them

# Meaningful Names

- Use intention-revealing names
- Avoid disinformation
- Make meaningful distinctions
- Use pronounceable names
- Use searchable names
- Avoid encodings
- Don’t be cute
- Pick one word per concept
- Add meaningful context

# Use Intention-Revealing Names

- Choosing good names takes time, but saves more than it takes.
- Change the names when you find better ones.
- The names should answer all the big questions:
  - Why it exists?
  - What is does?
  - How it is used?

# Use Intention-Revealing Names

- If a name requires a comment, then it doesn't reveal it's intent

```
int elapsedTimeInDays;
```

better than

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

# Use Intention-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;
}
```

# Use Intention-Revealing Names

1. What kind of things are in theList?
2. What is the significance of the zeroth subscript of an item in theList?
3. What is the significance of the value 4?
4. How would I use the list being returned?

# Use Intention-Revealing Names

```
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 Intention-Revealing Names

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}
```

# Avoid disinformation

- avoid leaving false clues that obscure the meaning of code
- avoid words whose entrenched meanings vary from our intended meaning:

  *hp*, *aix* and *sco* would be disinformative variable names because they are the names of Unix platforms and variants

# Avoid disinformation

- do not use names that vary in small ways:

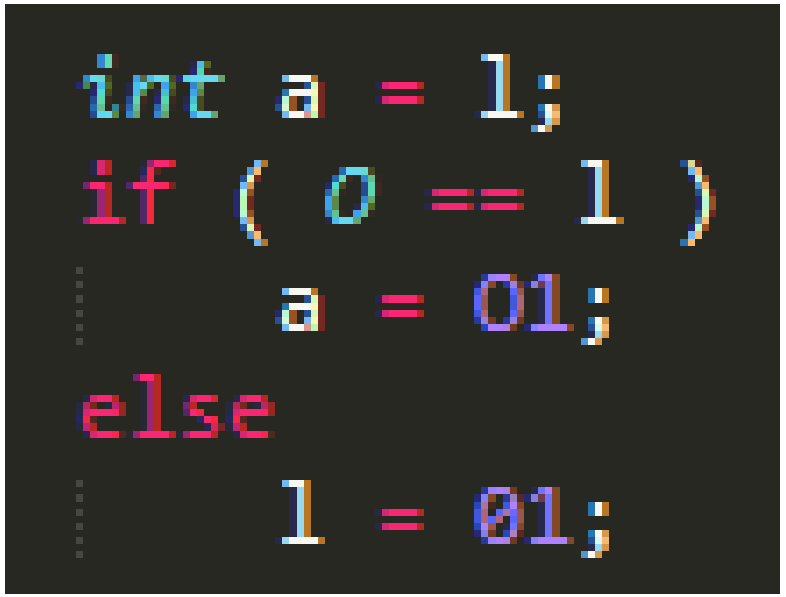
How long does it take to spot the difference between the following two variable names?

XYZControllerForEfficientHandlingOfStrings

XYZControllerForEfficientStorageOfStrings

# Avoid disinformation

- A truly awful example would be the use of lower-case L or upper-case O as variable names because they look almost entirely like the constants 1 and 0, respectively.



# Make meaningful distinctions

- do not arbitrarily change names solely to satisfy a compiler or interpreter
- if names must be different, then they should also mean something different
- distinguish the names in a such a way that the reader knows what the difference offer, not as follows:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

# Make meaningful distinctions

- don’t name a variable theBunny only because you already have a variable named bunny
- the word “variable” should never appear in a variable name, the word “table” – in a table name, the word “string” – in a string name, and so on.
- noise words are another meaningless distinction:
  - Product versus ProductInfo versus ProductData
  - Customer versus CustomerObject

# Make meaningful distinctions

- number-series naming (a1, a2, ..., aN) is the opposite of intentional naming

```
public static void copyChars(char a1[], char a2[]) {
    for (int i = 0; i < a1.length; i++) {
        a2[i] = a1[i];
    }
}
```

```
public static void copyChars(char source[], char destination[]) {
    for (int i = 0; i < source.length; i++) {
        destination[i] = source[i];
    }
}
```

# Use pronounceable names

- humans are good at words; a significant part of our brain is dedicated to the concept of words and we should use it
- if you can’t pronounce it, then you can’t discuss it without sounding like an idiot
- use names composed of verbs and nouns rather than using abbreviations

# Use pronounceable names

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

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

# Use searchable names

- single letter names and numeric constants are not easily located across a body of text
- easier to find MAX_CLASSES_PER_STUDENT than 7
- letter e – the most common letter in the English language – is likely to show up in every passage of text in every program
- single letter names can only be used as local variables inside short methods
- the length of a name should correspond to the size of its scope

# Use searchable names

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

```
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 encodings

- do not encode type or scope information into the variable name
  - encoded names are seldom pronounceable
  - encoded names can be easily mis-typed
  - there exists the risk to keep the type encoding in the variable name after the variable type has changed

# Avoid encodings

- don’t prefix member variables with m_ because:
  - the functions and classes should be small enough not to need them
  - you should be using an editing environment that highlights or colorizes members to make them distinct
  - people quickly learn to ignore the prefix to see the meaningful part of the name

# Class & Method Names

- class names should have noun or noun phrase names like *Customer*, *WikiPage*, *Account*, and *AddressParser*
- method names should have verb or verb phrase names describing the action they execute like *postPayment*, *deletePage*, or *save*

# Class & Method Names

- accessors, mutators, and predicates should be named for their value and prefixed with get, set, and is respectively.

```
string name = employee.getName();

customer.setName("Mike");

if (paycheck.isPosted())
    ...
```

# Don’t be cute

- choose clarity over entertainment value
- colloquialisms, slang and culture-dependent jokes will not be understood by everyone
- too clever names will be memorable only to people who share the author’s sense of humor and only as long as they remember the joke
  - DeleteItems() instead of HolyHandGrenade()
  - kill() instead of whack()
  - abort() instead of eatMyShorts()

# Pick One Word Per Concept

- pick one word for one abstract concept and stick with it in order to ensure the consistency of your system

It is confusing to have *fetch*, *retrieve*, and *get* as equivalent methods of different classes.

Or *controller*, *manager* and *driver*.

# Add Meaningful Context

- if a set of variables are related, place them in context by enclosing them in well-named functions, structures, classes, and namespaces
- as a last resort the variables can be prefixed with the same prefix
- if firstName, lastName, street, houseNumber, city, state, and zipcode form an address, context can be added by
  - using prefixes: addrFirstName, addrLastName, addrStreet, addrHouseNumber, addrCity, addrState, and addrZipcode
  - or, even better, using a data structure named Address

# Functions

- Small
- Do one thing
- One level of abstraction

# Functions

- First Rule:

  Functions should be small!

- Second Rule:

  Functions should be smaller than this!

# Functions should be small

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite) throws Exception {

    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}
```

# Functions should be small

- blocks within if statements, else statements, while statements and so on should be one line long, probably a function call
- this adds documentary value because the function called within the block can have a nicely descriptive name
- functions should not be large enough to hold nested structures
- the indent level of a function should not be greater than one or two

# Functions should do only one thing

Functions should do one thing.

They should do it well.

They should do it only.

# Functions should do only one thing

- if you can extract another function from it with a name that is not merely a restatement of its implementation, then it does more than one thing
- if you have only one answer for the question: “What does the function do?”, then it does only one thing

# Functions should have one level of abstraction

- we write functions to decompose a larger concept into a set of steps at the next level of abstraction
- if a function does only those steps that are one level below the stated name of the function, then the function is doing one thing
- The code should be read like a top-down narrative (The Stepdown Rule)

# Functions Arguments

- based on the number of arguments, the functions can be:
  - niladic – zero number of arguments (ideal case)
  - monadic – 1 argument
  - dyadic – 2 argumnets
  - triadic – 3 arguments (should be avoided where possible)
  - polyadic – more than 3 arguments (require very special justification and shouldn’t be used anyway)

# Functions Arguments

- common monadic forms (avoid those who don’t follow these forms):
  - asking a question about the argument

  boolean fileExists(“MyFile”);
  - operating on the argument, transforming it into something else and returning it

  InputStream fileOpen(“MyFile”);
  - events – 1 input arguments and 0 output arguments, alters the state of the system

  void passwordAttemptFailedNtimes(int attempts);

# Functions Arguments

- avoid flag arguments because:
  - complicates the signature of the method
  - violates Single Responsibility Principle (the function does one thing if the flag is true and another if the flag is false)

# Functions Arguments

- when a function seems to need more than two or three arguments, it's likely that some of those arguments ought to be wrapped into a data structure of their own

```
Circle makeCircle(double x, double y, double radius);

Circle makeCircle(Point center, double radius);
```

- x and y are part of a concept that deserve a name of its own, so wrapping them in a separate data structure is a good idea.

# Comments

- comments are not “pure good”, but, at best, a necessary evil
- some comments are necessary or beneficial, but most comments are excuses for poor code or justifications for insufficient decisions
- we must have them because we cannot always figure out how to express ourselves in code (without them), but their use is not a cause for celebration

# Comments

- avoid comments
- explain most of your intent in code
- In many cases you must simply create a function that says the same thing as the comment you want to write
- which would you rather see?

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

or

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

# Comments

- position markers: sometimes programmers like to mark a particular position in a source file in order to gather certain functions together beneath a banner like this:

// Actions ////////////////////////////////////////////////////////

- closing brace comments can be avoided by shortening the functions
- nonlocal information: the comment must appear near the code it describes
- function header comments can be avoided by keeping the function small and choosing a good name for it

# Good Comments

- Legal comments (copyright and authorship statements at the start of each source code)
- Informative comments

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile("\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

- Explanation of intent
- Clarification

```
assertTrue(a.compareTo(a) == 0); // a == a
```

# Good Comments

- Warning of Consequences

```
// Don't run unless you
// have some time to kill.
public void _testWithReallyBigFile()
{
    writeLinesToFile(10000000);
    ...
}
```

- TODO Comments

```
//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

```
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
- 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");
    }
}
```

# Bad Comments

- Misleading comments
- Mandated comments (not each function has to have a documentation)

```
/**
*
* @param title The title of the CD
* @param author The author of the CD
* @param tracks The number of tracks on the CD
* @param durationInMinutes The duration of the CD in minutes
*/
public void addCD(String title, String author,
int tracks, int durationInMinutes) {
    CD cd = new CD();
    cd.title = title;
    cd.author = author;
    cd.tracks = tracks;
    cd.duration = duration;
    cdList.add(cd);
}
```

# Bad Comments

- Journal Comments
- Don’t use comments when you can use a function or a variable
- Position markers
- Closing brace comments
- Attributions
- Commented-out code

# Bad Comments

- Nonlocal information
- Too much information
- Inobvious connection
- Function headers

# Formatting

- Vertical Formatting: How big should a source file be?
- Vertical Openness Between Concepts: Nearly all code is read left to right and top to bottom.

Each line represents an expression or a clause, and each group of lines represents a complete thought.

Those thoughts should be separated from each other with blank lines.

# Formatting

- Vertical Distance: Concepts that are closely related should be kept vertically close to each other.
- Variable Declarations: Variables should be declared as close to their usage as possible. Because our functions are short, local variables should appear at the top of each function.
- Dependent Functions: If one function calls another, they should be vertically close, and the caller should be above the callee if possible. This gives the program a natural flow.

# Indentation

- to make the hierarchy of scopes visible, we indent the lines of source code in proportion to their position in the hierarchy
- statements at the level of the file are not indented at all
- methods within a class are indented one level to the right of the class
- Implementations of those methods are implemented one level to the right of the method declaration.
- Block implementations are implemented one level to the right of their containing block, and so on.

# Testing your code

The Three Laws of TDD – Test Driven Development:

- First Law: You may not write code until you have written a failing unit test.
- Second Law: You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- Third Law: You may not write more production code than is sufficient to pass the currently failing test.

# Summary Names

- N1: Choose Descriptive Names

Names in software are 90 percent of what make software readable. You need to take the time to choose them wisely and keep them relevant.

- N2: Use Long Names for Long Scopes

The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names.

- N3: Avoid Encodings

Names should not be encoded with type or scope information.

# Summary Functions

- F1: Too Many Arguments

Functions should have a small number of arguments.

- F2: Output Arguments

Readers expect arguments to be inputs, not outputs. If your function must change the state of something, have it change the state of the object it is called on.

- F3: Flag Arguments

Boolean arguments states that the function does more than one thing.

- F4: Dead Function

Methods that are never called should be discarded.

# Summary Comments

- C1: Inappropriate Information (e.g. author, last modified date)
- C2: Obsolete Comment
- C3: Redundant Comment (i++; // increment)
- C4: Poorly Written Code
- C5: Commented-Out Code

# Summary General

- G1: Inconsistency

If you do something a certain way, do all similar things in the same way.

- G2: Function Names Should Say What They Do
- G3: Replace Magic Numbers with Named Constants
- G4: Encapsulate Conditionals

Boolean logic is hard enough to understand without having to see it in the context of an if or while statement. Extract functions that explain the intent of the conditional.

- G5: Functions Should Do One Thing

# Bibliography


- The Clean Coder: A Code of Conduct for Professional Programmers, Robert C. Martin
- Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin