CODING

- Programming Principles
- Coding Conventions

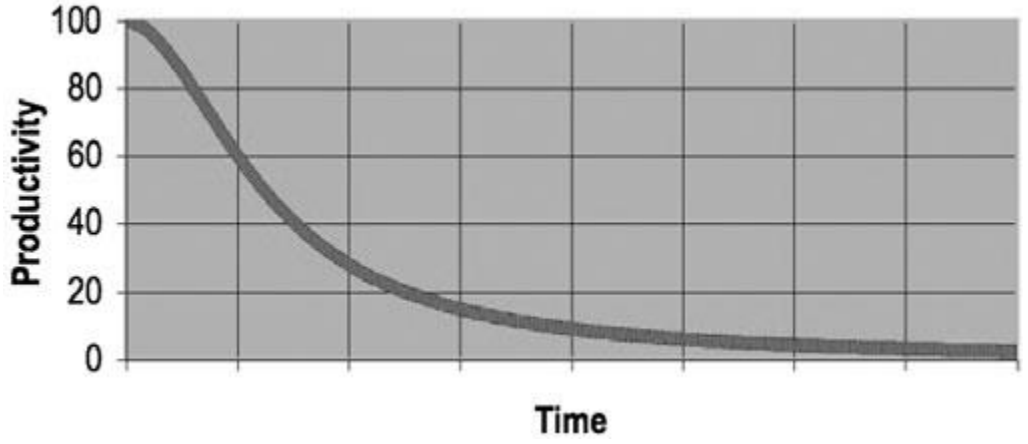
- First, you are a programmer. Second, you want
- to be a better programmer. Good. We need better programmers

- we should be concerned about models and requirements instead
- specifying requirements in such detail that a machine can execute them *is*
- *programming*. Such a specification *is code*.

Bad Code

- *was the bad*
- *code that brought the company down.*
- Have *you* ever been significantly impeded(obstruction/waiting) by bad code? If you
- It impeded the programmer also called “wading”
- Company can increase productivity by count

Company Add more employees for productivity Productivity vs Time



- New engineers are new will take time in understanding design

Attitude

- Shy
- So too it is unprofessional for programmers to bend to the will of managers who don’t
- understand the risks of making messes.
- what if you were a doctor and had a patient who demanded
- that you stop all the silly hand-washing in preparation for surgery because it was taking
- too much time?2 Clearly the patient is the boss; and yet the doctor should absolutely refuse
- to comply. Why? Because the doctor knows more than the patient about the risks of disease
- and infection. It would be unprofessional (never mind criminal) for the doctor to
- comply with the patient.

The Primal Conundrum – difficult question

- Programmers face a conundrum of basic values. All developers with more than a few years
- experience know that previous messes slow them down. And yet all developers feel
- the pressure to make messes in order to meet deadlines. In short, they don't take the time
- to go fast!
- True professionals know that the second part of the conundrum is wrong. You will *not*
- make the deadline by making the mess. Indeed, the mess will slow you down instantly, and
- will force you to miss the deadline. The *only* way to make the deadline—the only way to
- go fast—is to keep the code as clean as possible at all times

The Art of Clean Code?

- Writing clean code requires the disciplined use of a myriad(large type)
- little techniques applied
- through a painstakingly acquired sense of “cleanliness.” This “code-sense” is the key.
- Some of us are born with it. Some of us have to fight to acquire it. Not only does it let us
- see whether code is good or bad, but it also shows us the strategy for applying our discipline
- to transform bad code into clean code.
- A programmer without “code-sense” can look at a messy module and recognize the
- mess but will have no idea what to do about it. A programmer *with* “code-sense” will look
- at a messy module and see options and variations. The “code-sense” will help that programmer
- choose the best variation and guide him or her to plot a sequence of behavior
- preserving transformations to get from here to there.
- In short, a programmer who writes clean code is an artist who can take a blank screen
- through a series of transformations until it is an elegantly coded system.

What Is Clean Code?

- There are probably as many definitions as there are programmers. So I asked some very
- well-known and deeply experienced programmers what they thought

Bjarne Stroutstrup

- Elegant Pleasant, efficient
- Bjarne uses the word “elegant.” That’s
- quite a word! The dictionary in my MacBook®
- provides the following definitions: *pleasingly*
- *graceful and stylish in appearance or manner; pleasingly ingenious and simple.* Notice the
- emphasis on the word “pleasing.” Apparently Bjarne thinks that clean code is *pleasing* to
- read. Reading it should make you smile the way a well-crafted music box or well-designed
- car would.
- Bjarne also mentions efficiency—*twice*.
- Bad code *tempts* the mess to grow!

In conclusion

- Clean code is *focused*. Each
- function, each class, each module exposes a single-minded attitude that remains entirely
- undistracted, and unpolluted, by the surrounding details.

Grady Booch, author of *Object Oriented Analysis and Design with Applications*

- *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.*
- *Readability*

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.*

- Code,
- without tests, is not clean. No matter how elegant it is, no matter how readable and accessible,
- if it hath not tests, it be unclean.
- Dave uses the word *minimal* twice. Apparently he values code that is small, rather
- than code that is large. Indeed, this has been a common refrain throughout software literature
- since its inception. Smaller is better.
- Dave also says that code should be *literate*. This is a soft reference to Knuth's *literate*
- *programming*.4 The upshot is that the code should be composed in such a form as to make
- it readable by humans.

Ron Jeffries, author of *Extreme Programming Installed* and *Extreme Programming Adventures in C#*

- *In recent years I begin, and nearly end, with Beck's*
- *rules of simple code. In priority order, simple code:*
- *• 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, functions, and the lik*

Meaningful Names

- Names are everywhere in software. We name our variables, our functions, our arguments,
- classes, and packages. We name our source files and the directories that contain them. We
- name our jar files and war files and ear files. We name and name and name. Because we do

What follows are some simple rules for creating good names.

- **Use Intention-Revealing Name**
- Choosing names that reveal intent can make it much easier to understand and change
- code. What is the purpose of this code?
- public List<int[]> getThem() {
- List<int[]> list1 = new ArrayList<int[]>();
- for (int[] x : theList)
- if (x[0] == 4)
- list1.add(x);
- return list1;
- }
- Why is it hard to tell what this code is doing? There are no complex expressions.
- Spacing and indentation are reasonable. There are only three variables and two constants
- mentioned. There aren't even any fancy classes or polymorphic methods, just a list of arrays (or so it seems).
- The problem isn't the simplicity of the code but the *implicity* of the code (to coin a
- phrase): the degree to which the context is not explicit in the code itself. The code implicitly
- requires that we know the answers to questions such as:
- **1.** What kinds 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?**s**

- Say that we're working in a mine sweeper game. We find that the board is a list of
- cells called theList. Let's rename that to gameBoard.
- Each cell on the board is represented by a simple array. We further find that the zeroth
- subscript is the location of a status value and that a status value of 4 means "flagged." Just
- by giving these concepts names we can improve the code considerably:
- public List<int[]> getFlaggedCells() {
- List<int[]> flaggedCells = new ArrayList<int[]>();
- for (int[] cell : gameBoard)
- if (cell[STATUS_VALUE] == FLAGGED)
- flaggedCells.add(cell);
- return flaggedCells;

Avoid disinformation – inconsistent spelling

- A truly awful example of disinformative names would be the use of lower-case L or
- uppercase O as variable names, especially in combination. The problem, of course, is that
- they look almost entirely like the constants one and zero, respectively.
- int a = l;
- if (O == l)
- a = O1;
- else
- l = 01;

- Class , function, method name
- Use problem domain names
- Add meaningful Context
- Avoid mindful mappings

Functions

- Small , Do one thing,
- Indented in Blocks
- Levels of Abstraction
- Avoid arguments (testing point of view)

Comments

- Inaccurate comments are far worse than no comments at all.
- **Comments Do Not Make Up for Bad Code**
- One of the more common motivations for writing comments is bad code. We write a module
- and we know it is confusing and disorganized. We know it's a mess. So we say to ourselves,
- "Ooh, I'd better comment that!" No! You'd better clean it!
- Clear and expressive code with few comments is far superior to cluttered and complex
- code with lots of comments. Rather than spend your time writing the comments that
- explain the mess you've made, spend it cleaning that mess.
- **Explain Yourself in Code**

Good Comments

- Informative
- To DO
- Warning of consequences
- Clarifying
- Explanation of intent

Bad Comments

- Noisy(unnecessary)
- Journal
- Redundant
- Closing Brace
- Too much information

Code Formatting

- Vertical Formatting(Newspapers)
- Horizontal Formatting

Error Handling

- Use Exceptions
- Unchecked Exception instead of checked

- **Conclusion**
- Clean code is readable, but it must also be robust. These are not conflicting goals. We can
- write robust clean code if we see error handling as a separate concern, something that is
- viewable independently of our main logic. To the degree that we are able to do that, we can
- reason about it independently, and we can make great strides in the maintainability of our
- code.