
NEW! STRONGER THAN DIRT
ENTER.
tech with a heartbeat
{Clean Code}
CAUTION: EVEN BAD CODE CAN FUNCTION
40 FL OZ (1.25 QT) 1.18L

NAMES

MEANINGFUL NAMES

- Names are **essential** in software development.
- The names of variables, functions, classes and package should **reveal their intent**, avoid **disinformation**, be **pronounceable**, be **searchable**, avoid **encodings** and avoid **mental mapping**.
- *“Say what you mean, and mean what you say!”*

INTENTION-REVEALING NAMES

- Names of variables, functions, classes (etc.) should answer all the big questions:
  1. Why do they exist?
  2. What are they supposed to do?
  3. How it is does it?

- It is even possible to tell a variable's, function's or classe's scope by properly choosing a name (***without suffixes, though ;)***).

AVOID DISINFORMATION

- Do not leave **false clues** that obscure the meaning of the code and not use names which **meaning vary from context to context**.

```
1  double hp; //vs
2  double hypotenuse;
```

- Avoid choosing names that give **incorrect/inaccurate** information about how a variable works or is meant to be used.

```
1  auto valuesList = Values.GetValuesStack();
```

ENCODING

- Usually encoding provides extra information about a variable type or scope;
- Encoding adds an extra burden of deciphering the name;
  - Unnecessary mental burden when trying to solve a problem;
  - Above it all: encoded names are hardly pronounceable and are easy to mis-type;
  - Example: Hungarian Notation.

EXAMPLE

```
1  uint8_t d;
2  char myVariable;
3  static theObject sr_rlrtk;
4  vector<int> listOfEntriesNotFoundInTheQueryResult;
5  // Pointer to an orthogonality-calculation function
6  // accepting a set of *NORMALIZED* stokes vectors
7  void (*callback)(vector<stokes>, bool, int);
```

UNCLE BOB'S NAMING GUIDELINES

- Class names should be noun or noun phrases;
  - Customer; WikiPage; Account; AddressParser;

- Avoid using buzz words like Manager, Processor, Data or Info;

- *A Class name should not be a verb!*

UNCLE BOB'S NAMING GUIDELINES

- Method names should have verbs or verbal phrases;
  - postPayment; deletePage; save;

- Accessors, mutates and predicates should be named from their values and prefixed with set, get, is, etc.
  - setContext; getServerStatus, isClientConnected;

UNCLE BOB'S NAMING GUIDELINES

- Booleans should have predicates:
  - isDurty; isUpdated; isEmpty

- Those read nicely along with if statements:

```
1  If (userWallet.isEmpty)
```

UNCLE BOB'S NAMING GUIDELINES

- Enumerations are usually named after adjectives:

```
1  enum state {
2    headless,
3    inactive,
4    detached,
5    connected,
6    halted,
7    Offline,
8  };
```

THE SCOPE LENGTH RULE

- The longer the scope of a variable, the longer its name must be!
  - Variables declared and used in a small scope can have very small names (even one letter);
  - Member attributes, on the other hand, should have longer and descriptive names.

- Functions and classes follow the opposite rule!
  - Public functions should have short names;
  - Derived classes have longer names, because they should have adjectives added to their names!

ENTER.
tech with a heartbeat

FUNCTIONS

FUNCTIONS

- Functions are the first line of organization of any program.
- They provide modularity and reusability of code.
- “The first rule is that they should be small. *The second rule is that they should be smaller than that!*”
- Functions should be *transparently obvious* and *should tell a story*.

❗ REMEMBER ❗

Functions should be no bigger than 4 or 5 lines.

KEEPING YOUR FUNCTIONS SMALL

- The code inside `if`, `else` and `while` statements should be a function call;
    - It makes the code inside those statements one line long.
    - It keeps the inclosing code small and serves as documentation because the function called within the block have a nicely descriptive name.

DO ONE THING

- *Functions should do one thing. They should do it well; and they should do it only!*

- Your function is doing one thing if you can't extract a function out of it.

- Extract until you get a function which name can only be a description of its implementation.

SWITCH STATEMENTS

- By definition they do N things.
- Cannot always be replaced, but sometimes it can be replaced by polymorphic classes.
    - Solution: Implement an abstract factory to hide the switch statement.
- Avoid using `switch` statements when possible.

FUNCTION ARGUMENTS

- *The fewer the better*. Ideally **zero**, but it is not always possible so use at most **two**.
  - Functions with three arguments are considerably harder to understand.
- Should not be used as the output of a function.
  - Functions should give their returns values via return not via their arguments.

❗ REMEMBER ❗

Argument names should form a verb/noum pair.

`Ex: Write(String name)`

COMMAND AND QUERY SEPARATION

- *Functions should* **do** smething or **answer** something, **never both**!

```
1  if(set(“username”, “uncle bob”))
```

  - *What does the return of set means?*

- It can be improved look like this

```
1  if(AttributeExists(“username”)
2     setAttribute(“username”, “uncle bob”)
```

ENTER.
tech with a heartbeat

COMMENTS

COMMENTS

- Comments are used to compensate for our failure to express ourselves in code.
- They are always failure.
- Only the code can tell the truth of what it is doing.

COMMENTS DO NOT MAKE UP FOR BAD CODE

- Bad code is a good motivation for writing comments.
- Clear and expressive code with a feel comments is far superior than complex and commented code.

GOOD COMMENTS

- Legal comments

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

GOOD COMMENTS

- Informative comments

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

GOOD COMMENTS

- Explanation of intent

```
1  //This is our best attempt to get a race condition
2  //by creating large number of threads.
3  for (int i = 0; i < 25000; i++) {
4      WidgetBuilderThread widgetBuilderThread =
5      new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
6      Thread thread = new Thread(widgetBuilderThread);
7      thread.start();
8  }
```

GOOD COMMENTS

- Warning consequences

```
1   // Don't run unless you
2   // have some time to kill.
3   public void _testWithReallyBigFile() {
4       writeLinesToFile(10000000);
5       response.setBody(testFile);
6       response.readyToSend(this);
7       String responseString = output.toString();
8       assertSubString("Content-Length: 1000000000", responseString);
9       assertTrue(bytesSent > 1000000000);
10  }
```

GOOD COMMENTS

- TODO comments

```
1  //TODO-MdM these are not needed
2  // We expect this to go away when we do the
3  // checkout model protected
4
5  VersionInfo makeVersion() throws Exception
6  {
7          return null;
8  }
```

GOOD COMMENTS

- Amplification

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

BAD COMMENTS

- Mumbling

```
1  public void loadProperties() {
2      try {
3          String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
4          FileInputStream propertiesStream = new FileInputStream(propertiesPath);
5          loadedProperties.load(propertiesStream);
6      }
7      catch(IOException e) {
8          // No properties files means all defaults are loaded
9      }
10 }
```

BAD COMMENTS

- Redundant comments

```
1  // Utility method that returns when this.closed is true. Throws an exception
2  // if the timeout is reached.
3  public void waitForClose(final long timeoutMillis) throws Exception {
4      if(!closed) {
5          wait(timeoutMillis);
6          if(!closed)
7          throw new Exception("MockResponseSender could not be closed");
8      }
9  }
```

BAD COMMENTS

- Mandated comments

```
1   /** *
2   * @param title The title of the CD
3   * @param author The author of the CD
4   * @param tracks The number of tracks on the CD
5   * @param durationInMinutes The duration of the CD in minutes */
6   public void addCD(String title, String author, int tracks, int durationInMinutes) {
7           CD cd = new CD();
8           cd.title = title;
9           cd.author = author;
10          cd.tracks = tracks;
11          cd.duration = duration;
12          cdList.add(cd);
13  }
```

BAD COMMENTS

- Position marker comments

```
1  // Actions ////////////////////////////////
2  ...
3
4  //------------------------
5  ...
6  //========================
```

BAD COMMENTS

- Closing brace comments

```
1   public void loadProperties() {
2       try {
3           for(int i=0; i< entry.size; i++){
4           {
5                       . . .
6           }
7       } // try
8       catch(IOException e) {
9           . . .
10      } // catch
11  }
```

BAD COMMENTS

- Commented out code

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
        writeEnd();
        this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
```

BAD COMMENTS

- Too much information

```
/*
RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
Part One: Format of Internet Message Bodies
section 6.8. Base64 Content-Transfer-Encoding
The encoding process represents 24-bit groups of input bits as output strings of 4
encoded characters. Proceeding from left to right, a 24-bit input group is formed by
concatenating 3 8-bit input groups. These 24 bits are then treated as 4 concatenated 6-
bit groups, each of which is translated into a single digit in the base64 alphabet. When
encoding a bit stream via the base64 encoding, the bit stream must be presumed to be
ordered with the most-significant-bit first. That is, the first bit in the stream will be
the high-order bit in the first 8-bit byte, and the eighth bit will be the low-order bit
in the first 8-bit byte, and so on.
*/
```

BAD COMMENTS

- Not obvious connection

```
1  /*
2  start with an array that is big enough to hold all the
3  pixels * (plus filter bytes), and an extra 200 bytes for header info
4  */
5  this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

ENTER.
tech with a heartbeat

FORMATTING

FORMATTING

- We should take care that our code is nicely formatted.

- By choosing a simple set of rules that govern the format of the code.

VERTICAL FORMATING

- How big should a source file be?
  - No clear answer to this question, but small files are usually easier to understand than larger files. **So keep your files small**.

THE NEWSPAPER METAPHOR

- The further you read more detail you get: We want our source files to be read like a newspaper article
- The top most part of the code should provide the high-level concepts and algorithms
- Detail should increase as we move downwards

VERTICAL DISTANCE

- Concepts that are closely related should be kept vertically close to each other;

- Closely related concepts should not be separated into multiple files unless you have a very good reason for…

VARIABLE DECLARATION

- Should be declared as close to their usage as possible;
- Control variables in loops should be declared within the loop statement.

HORIZONTAL FORMATTING

- How wide should be our code lines.
- Keep your lines short, within a limit of 80 characters, but it is okay between 100 and 120.
- You should never be able to scroll to the right.

BREAKING INDENTATION

- Never make 1 line if statements or function implementations

```
1   if (file.isreadable) { return true; }
```

- Dummy scopes
  - Add the dummy body is indented and surrounded by braces. Make the semicolon visible;

```
1   while (dis.read(buf, 0, readBufferSize) != -1)
2   ;
```

ENTER.
tech with a heartbeat

DATA ABSTRACTION

DATA ABSTRACTION

- Hiding the implementation is all about abstraction.
- We don’t want to expose the details of our data; We want to express our data in abstract terms.
- And this is not accomplished by using interfaces and/or getters and setters.
- **The worst option is blindly add getters and setters**.

DATA/OBJECT ANTI-SYMMETRY

- Objects hide their data behind abstractions and expose functions that operate on that data
- Data structures expose their data and have no meaningful functions
- *They are virtually opposites!*
- And have their advantages in different situations.

HYBRIDS

- Are objects that are half object and half data struct.
- They have functions that do significant things, and they also have either public variables or public acessors and mutators.
- This kind of classes makes it hard to add new functions, but also make it hard to add new data structures.
- Worst of both worlds. Avoid creating them.

ENTER.
tech with a heartbeat

ERROR HANDLING

ERROR HANDLING

- Use exceptions rather than return codes
- Returning error codes, makes the caller immediately check the return to see if it is an error or a normal return.
- **Using exceptions separates the main code from the exception handling.**

WRITE YOUR TRY-CATCH STATMENTS FIRST

- The code executed in a try statement can be aborted at any time.
  - In a try-catch-finally statement, we are forcing our code to leave this block in a consistent state, no matter what happens inside the try block.

- Writing try-catch-finally statement first helps you to define what the user of that code should expect, no matter what goes wrong with the code executed in the try block.

PROVIDE CONTEXT WITH YOUR EXCEPTIONS

- Each exception you throw should provide enough context to determine the source and location of the error.

- Create an informative text message and pass them along with your exceptions.
  - Mention the operation that failed and the type of failure.

DON'T RETURN NULL

- If we return null we are essentially creating work for ourselves and forcing problems upon our callers.
  - One missing null check can make your application go out of control.
- In this cases, try to throw an exception or a Special Case object.
  - Like an empty list, for example (instead of a null).

DON'T PASS NULL

- Passing null to objects is even worse.
- In most languages, there's no real good way to deal null with passing.

ENTER.
tech with a heartbeat

BOUNDARIES

BOUNDARIES

- We seldom control all the software in our systems.
- Sometimes we have to integrate third party code into our own and it should happen cleanly.

USING THIRD-PARTY CODE

- There's a natural tension between the provider of an interface and the user of an interface.
- Providers: *Strive for broad applicability so they can work in many environments and appeal to a wide audience*.
- Users: *want the interface to work on their particular needs.*
- **And this tension can cause problems at the boundaries of our systems**.

USING THIRD-PARTY CODE

- Consider Java Map. If the application needs a Map of sensors:

```
1   Map sensor = new HashMap();
```

- If the application needs to access a sensor we need to:

```
1   Sensor s = (Sensor)sensors.get(sensorId);
```

- Which is not clean code!

USING THIRD-PARTY CODE

- A better way would be by using java generics that transform it to:

```
1   Map<Sensor> sensor = new HashMap<Sensors>();
```

- If the application needs to access a sensor we need to:

```
1   Sensor s = sensors.get(sensorId);
```

- However, we still depend on the Map interface which can be a problem if the map interface ever changes.

USING THIRD-PARTY CODE

- The solution to this problem is to wrap it in its own class:

```
1  Public class Sensors {
2          private Map sensors = new HashMap();
3
4          public Sensor getById(String id) {
5                  return (Sensor) sesnors.get(id);
6          }
7
8          . . .
9  }
```

- It hides the interface at the boundary Map.

USING THIRD-PARTY CODE

- This new interface is also tailored to meet the needs of the application and it results in code that is easier to understand and harder to misuse.

- The sensors class can enforce the design and business rules.

- **Avoid returning boundary interfaces like a map, or sending it as return of public APIs**.

ENTER.
tech with a heartbeat

TEST DRIVEN DEVELOPMENT

TEST DRIVEN DEVELOPMENT

- Test driven development is relatively new, it became popular in the mid 90s.

- The agile and TDD movements have encouraged many programmers to write automated unit tests, but many have missed some of the more subtle and important points on doing it good.

THE THREE LAWS OF TDD

- TDD asks us to write test first, before writing production code
  - **First law**: You may not write production code until you have written 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 current failing tests.

Working according to the laws, you will write dozens of tests every day and those tests will cover virtually all your production code.

KEEPING TEST CLEAN

- Tests must change as the production code evolves.
- The dirtier the test the harder they are to change.
- The more tangled the test code, the more likely it is that you will spend more time cramming new tests into the suite than it takes to write the new production code.
- Test code is just as important as production code. It requires thought, design, and care.

CLEAN TEST

- Three things makes a clean test:
  - ***Readability***, ***readability*** and ***readability***.

- It is more important in unit test than in production code and what makes it readable are clarity, simplicity and density of expression.

A DUAL STANDARD

- Test code must be simple, succinct and expressive but it need not to be as efficient as production code. Prioritize readability than efficiency in your test code.

ONE ASSERT PER TEST

- Tests designed in this way come to a single conclusion that is quick and easy to understand. It is not a rule though it is a guide line. But it is important that the number of asserts in a test ought to be minimized.

SINGLE CONCEPT PER TEST

- Another rule is that we want to test a single concept in each test function. We don’t want long test functions that go testing one miscellaneous thing after the another.

- So minimize the number of asserts per concept and test just on concept per test function.

F.I.R.S.T

- Clean test follow the following 5 other rules.

  - **Fast**: Tests should be fast, they should run quickly. If they are too slow, you probably will not run them frequently.
  - **Independent**: The tests should not depend on each other. One test should not set up conditions to the next. When they depend on each other, the first one to fail cause a cascade of downstream failures.
  - **Repeatable**: Test should be repeatable in any environment. You should be able to run the tests in the production environment.
  - **Self-validating**: The test should have a boolean output. Either pass or fail.
  - **Timely**: Test should be written in a timely fashion. Unit test should be written just before the production code that makes them pass. If you write tests after, you may find the production code hard to test. You may not design the production code to be testable.