
—

# Clean Code

UNCLE BOB’S SCHOOL OF CLEAN CODE

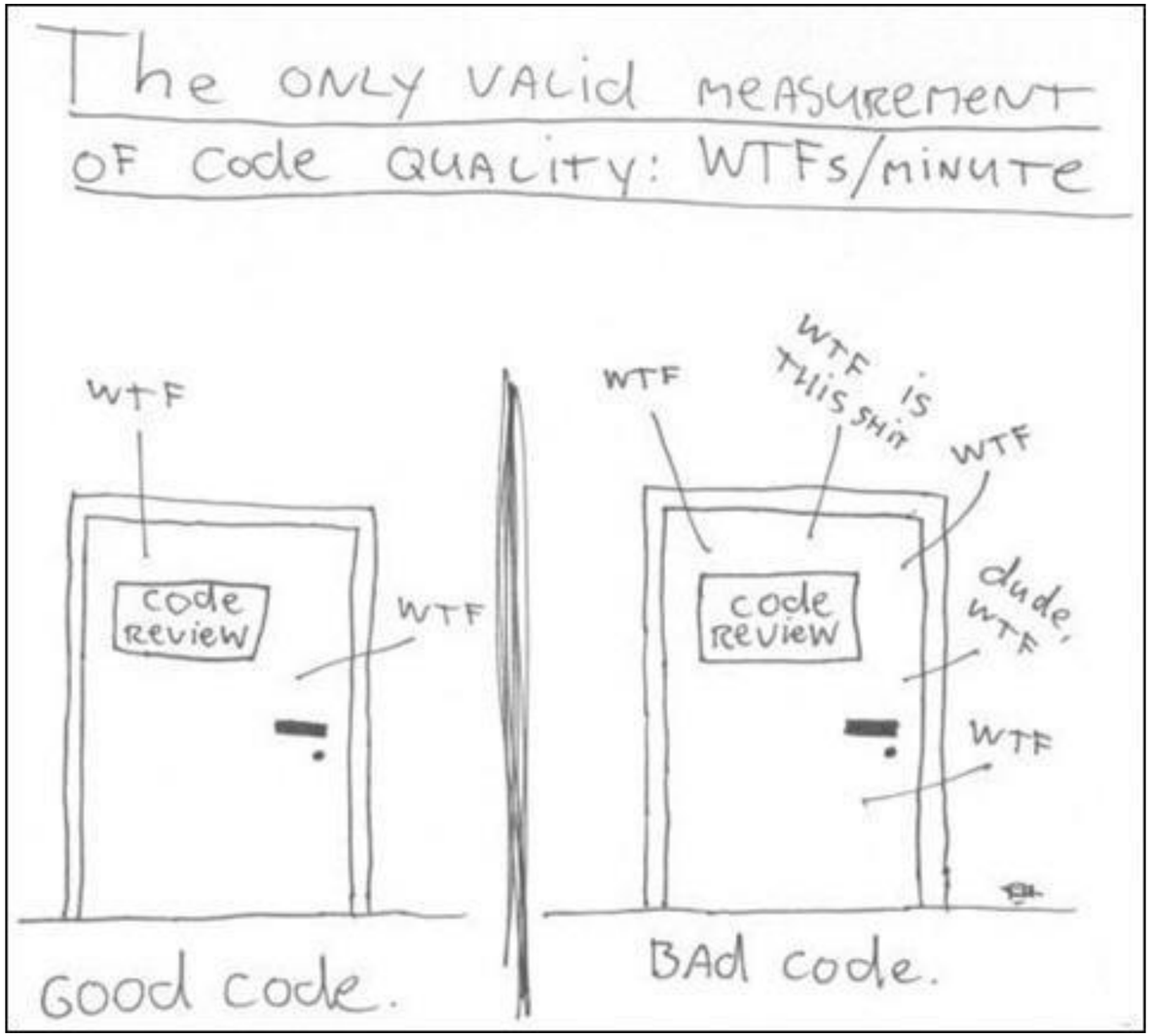
# Agenda

1. CLEAN CODE
2. MEANINGFUL NAMES
3. FUNCTIONS
4. CLASSES
5. ERROR HANDLING
6. COMMENTS
7. FORMATTING
8. UNIT TESTS
9. RESOURCES

1

# Clean Code

# Which Door Do We Prefer?



# There Will Be Code

Someone will say that the code is no longer the issue! That we should be concerned about models and requirements instead. Indeed some have suggested that we are close to the end of code. That soon all code will be generated instead of written (GitHub Copilot). The programmers won't be needed because business people will generate programs from specification.

NONSENSE!

We will NEVER be rid of code, because code represents the details of the requirements.

These machines will have to be able to understand us so well that they can translate vaguely specified needs into perfectly executing programs that precisely meet those needs. This will NEVER happen! Not even humans, with all their intuition and creativity, have been able to create successful systems from the vague feelings of their customers.

# Bad Code

## Bad code can bring companies down

We've all looked at the mess we've just made and then have chosen to leave it for another day. We've all felt the relief of seeing our messy program work and deciding that a working mess is better than nothing. We've all said we'd go back and clean it up later. But LATER EQUALS NEVER.

## Requirements

The most prominent facts in software development: REQUIREMENTS ALWAYS CHANGE (especially in the agile world).
New requirements, new changes (requirements change in ways that thwart the original code).
The requirements make life hell if we have bad code.

## The cost of owning a mess

Every change we make to the code breaks two or three other parts of the code. No change is trivial. Over time the mess becomes so big and so deep and so tall, we can not clean it up. There is no way at all.
In the end we end up complaining about deadlines, managers, intolerant customers, etc... But the only fault is ours!
The only way to make the deadline - the only way to go fast - is to keep the code as CLEAN as possible at all the times.

# Why Clean Code?

## Reading code > writing code

The ratio of time spent reading vs. writing is well over 10:1. So we are constantly READING old code as part of the effort to write new code.
Because the ratio is so high, we want the reading of code to be easy, even if it makes the writing harder.
Making the code easy to read actually makes it easier to write. There is no escape from this logic. You cannot write code if you cannot read the surrounding code.

## The boy scout rule

Leave the campground cleaner than you found it!
If we all checked-in our code a little cleaner than when we checked it out, the code simply could not rot. (change one variable name, eliminate one small bit of duplication, etc...)

## What is clean code?

Clean code is code that has been taken care of. Someone has taken the time to keep it simple and orderly. They have paid appropriate attention to details. They have cared.
How to take care of your code? We'll see that in the next slides.

2

# Meaningful Names

# Why Meaningful Names?

## Names are everywhere in software

We name our variables, our functions, our arguments, classes, and packages. We name source files and the directories that contain them. We name and name and name. Because we do so much of it, we'd better do it well.

Choosing good names takes time but saves more than it takes. So take care with your names and change them when you find better ones.

One difference between a smart programmer and a professional programmer is that the professional understands that clarity is king. Professionals use their powers for good and write code that others can understand.

Don't be afraid to spend time choosing a name.
Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name. A long descriptive name is better than a long descriptive comment.

# Use

## Use intention-revealing names

The name of a variable, function, or class, should answer ALL the big questions. It should tell you why it exists, what it does, and how it is used. If name requires a comment, then the name does not reveal its intent.

```
int d; // elapsed time in days        - reveals nothing
int elapsedTimeInDays;                - reveals everything
```

## Use pronounceable names

If you can't pronounce it, you can't discuss it without sounding like an idiot.
This matters because programming is a social activity!

```
Date genymdhms;                       - can not be pronounced
Date generationTimestamp;
```

## Use searchable names

Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

# Avoid

## Avoid disinformation

Beware of using names which vary in small ways.
XYZControllerForEfficientHandlingOfStrings and XYZControllerForEfficientStorageOfStrings - it is hard to spot the difference.
A truly awful example of disinformative names would be the use of lower-case L or uppercase o as variable names, especially in combination. Look like the constant one and zero.

## Avoid noninformative words

Number-series naming (a1, a2, ... aN) is the opposite of intentional naming. Such names are not disinformative - they are noninformative; they provide no clue to the author's intention.

## Avoid noise words

Name instead NameString, Customer instead CustomerObject, money instead moneyAmount, account instead accountData.

# Pick One Word per Concept

Pick one word for one abstract concept and stick with it. For instance, it's confusing to have fetch, retrieve, and get as equivalent methods of different classes. Likewise, it's confusing to have a controller and a manager and a driver in the same code base.

# Methods, Classes, and Interfaces

## Method names

Methods are verbs of that language.
Methods should have verb or verb phrase names like postPayment, deletePage, or saveSomething.
The smaller and more focused a function is, the easier it is to chose a descriptive name

## Class names

Classes are the nouns of that language.
Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, AddressParser...
Avoid words like Manager, Processor, Data, or Info in the name of a class. A class name should not be a verb!

## Interfaces and implementations

Uncle Bob prefer not using I in front of the interface name, ShapeFactory instead of IShapeFactory.
But I don't agree, I'm using I in front of interface (and many of you will use it).

# Don’t

## Don't be cute
Don't use humor.

## Don't pun
Using same word for two purposes. One word per concept.

## Don't add gratuitous context
If the imaginary application called "Gas Station Deluxe", it is a bad idea to prefix every class with GSD.

3

# Functions

# Small!

The first rule of functions is that they should be small.
The second rule of functions is that they should be smaller than that.
Functions should hardly ever be 20 lines long.

# Blocks and Indenting

The blocks within if statements, else statements, while statements, and so on should be one line long. Probably that line should be a function call.

# Do One Thing

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

Sections within functions - This is an obvious symptom of doing more than one thing.
Functions that do one thing cannot be reasonably divided into sections.

## Have No Side Effects

Side effects are lies. Your function promises to do one thing, but it also does other hidden things.
Functions should either do something or answer something, but not both.

# Function Arguments

The ideal number of arguments for a function is zero.
Next comes one, followed by two. The arguments should be avoided where possible.
More than three requires very special justification and then shouldn't be used anyway.
When function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own.

Arguments are even harder from a testing point of view. With more than two arguments, testing every combination of appropriate values can be daunting.

## Flag Arguments

Flag arguments are ugly.
Passing a boolean into a function is a truly terrible practice. It immediately complicates the signature of the method, loudly proclaiming that this function does more than one thing. It does one thing if the flag is true and another if the flag is false!

4

# Classes

# Class Organization

A class should begin with a list of variables. Public static constants, if any, should come first. Then private static variables, followed by private instance variables.

Public functions should follow the list of variables. We like to put the private utilities called by a public function right after the public function itself. This follows the stepdown rule and helps the program read like a newspaper article.

# Classes Should Be Small

The first rule of classes is that they should be small.
The second rule of classes is that they should be smaller than that.

The name of a class should describe what responsibilities it fulfills. In fact, naming is probably the first way of helping determine class size. If we cannot derive a concise name for a class, then it's likely too large.
We should also be able to write a brief description of the class in about 25 words, without using the words "if", "and", "or", or "but".

We want our systems to be composed of many small classes, not a few large ones.

# Encapsulation

We like to keep our variables and utility functions private.

# Cohesion

A class in which each variable is used by each method is maximally cohesive.
When cohesion is high, it means that the methods and variables of the class are co-dependent and hang together as a logical whole.

When classes lose cohesion, split them!
So breaking a large function into many smaller functions often gives us the opportunity to split several smaller classes out as well. This gives our program a much better organization and a more transparent structure.

# SOLID

Next presentation.

5

# Error Handling

# Use Exceptions Rather Than Return Codes

The calling code is cleaner. Its logic is not obscured by error handling.

# Provide Context With Exceptions

You get stack trace from any exception: however, a stack trace can't tell you the intent of the operation that failed.
Create informative error messages and pass them along with your exceptions.

# Don’t Return Null!

The first on the list of things we do that invite errors is returning null!
All it takes is one missing null check to send an application spinning out of control.

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.

Too many null checks affects the code readability. Avoid working with nulls.

# 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 expect you to pass null, you should avoid passing null in your code whenever possible.

The rational approach is to forbid passing null by default.

6

# Comments

# Don’t Comment Unused Code – Delete It!

Git: *keeps an easily accessible version of your entire code history*

Developer: *comments out unused code but keeps it just in case*

Git:

# Don’t Comment Bad Code - Rewrite It!

Nothing can be quite so helpful as a well-placed comment. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation. Inaccurate comments are far worse than no comments at all.

The proper use of comments is to compensate for our failure to express ourself in code. Comments are always failures.

Why am I so down on comments? Because they lie. Not always, and not intentionally, but too often.
Truth can only be found in one place: THE CODE. Only the code can truly tell you what it does. So focus on clean code, not commenting.

## Comments do not make up for bad code

One of the more common motivations for writing comments is bad code.
We see a mess and we say to ourselves, "Ooh, I'd better comment that!" No! You'd better clean it!

# Explain Yourself in Code

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

Why not something like this?

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

# Good Comments

Some comments are necessary or beneficial. But keep in mind, that the only truly good comment is the comment you found a way not to write it.

1. Legal comment (company copyright in the beginning of file)
2. Informative comment
3. Explanation of intent
4. Clarification
5. Warning of consequences
6. TODO comments (TODOs are jobs that the programmer thinks should be done, but for some reason can't do at the moment.)
7. Amplification (A comment may be used to amplify the importance of something that may otherwise seem inconsequential.)
8. Javadocs/summary in public APIs (But this also can be just as misleading, nonlocal, and dishonest as any other kind of comment.)

# Bad Comments

Most comments fall into this category. Usually they are crutches or excuses for poor code.

1. Mumbling (If you decide to write a comment, then spend the time necessary to make sure it is the best comment you can write.)
2. Redundant comments
3. Misleading comments
4. Mandated comments (It is just plain silly to have a rule that says that every function must have a javadoc/summary, or every variable must have a comment.)
5. Journal comments (you have a version control that does it for you.)
6. Noise comments (these comments are so noisy that we learn to ignore them.)
7. Don't use a comment when you can use a function or a variable
8. Position markers
9. Closing brace comments
10. Commented-out code (you have a version control that saves it for you.)
11. Nonlocal information (don't offer systemwide information in the context of a local comment.)
12. Too much information
13. Function headers (short functions don't need much description. A well-chosen name for a small function that does one thing is usually better than a comment header.)
14. Javadocs/summary in nonpublic code

7

# Formatting

# The Purpose of Formatting

First of all, let's be clear. Code formatting is important. Code formatting is about communication, and communication is the professional developer's first order of business.

The functionality that you create today has a good chance of changing in the next release, but the readability of your code will have a profound effect on all the changes that will ever be made.

A good software system is composed of a set of documents that read nicely.

# Vertical Formatting

Small files - 200 lines long with upper limit of 500? It should be considered very desirable.

1. Vertical openness between concepts
   Blank lines between block of properties, constructors, methods...

2. Vertical distance
   Concepts that are closely related should be kept vertically close to each other.

3. Variable declarations
   In the function, variables should be declared as close to their usage as possible. Local variables should appear at the top of each function.
   Control variables for loops should usually be declared within the loop statement.

4. Instance variables
   Should be declared at the top of the class. They are used by many, if not all, of the methods of the class.

5. Dependent functions
   If one function calls another, they should be vertically close, and the caller should be above the calee, if at all possible.

6. Vertical ordering
   We expect low-level details to come last.

# Horizontal Formatting

How wide a line should be? Max 100-120.

1. Horizontal openness and density
   Surround the assignment operators with space to accentuate them.
   Separate arguments within the function call parenthesis to accentuate the comma and show that the arguments are separate.

2. Horizontal alignment
   Don't put tabs to align things horizontally (just one whitespace is enough between them).

3. Indentation
   A source file is a hierarchy rather like an outline.
   Without indentation, programs would be virtually unreadable by humans.

8

# Unit Tests

# Clean Tests

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)

Test code is just as important as production code. It requires thought, design, and care. It must be kept as clean as production code.

If you don't keep your tests clean, you will lose them. And without them you lose the very thing that keeps your production code flexible.

Readability is perhaps even more important in unit tests than it is in the production code.

# Tests Namings

Project - [ProjectUnderTest].UnitTests

Class - [ClassName]Tests

Method - [UnitOfWorkName]_[ScenarioUnderTest]_[ExpectedBehavior]

*UnitOfWorkName* - The name of the method or group of methods or class you're testing (usually a method).

*ScenarioUnderTest* - The condition under which the unit is tested (such as "bad login", "invalid user"...). You could describe the parameters being sent to the method. Or the initial state of the system when the unit of work is invoked (such "no user exist", "user already exists"...)

*ExpectedBehavior* - What you expect the tested method to do under the specified conditions (ScenarioUnderTest). This could be one of three possibilities:
- return a value (a real value, or exception)
- change the state of the system
- call a third-party system.

The method name explains the test like:
When I call method X with Y, then it should do Z.

# AAA (Arrange, Act, Assert)

Known also as Build – Operate – Check (Uncle Bob - Clean Code book)
Also similar to GWT (Given – When – Then)

*Arrange* objects, creating and setting them up as necessary.

*Act* on an object/SUT.

*Assert* that something is as expected.

# F.I.R.S.T.

- Fast - Tests should be fast! They should run quickly. When tests run slow, you won't want to run them frequently.

- Independent - Tests should not depend on each other! One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. Isolated!

- Repeatable - Tests should be repeatable in any environment! You should be able to run them in the production environment, in the QA environment, on your laptop while riding home on the train without a network.

- Self-Validating - The tests should have a boolean output, either they pass or fail!

- Timely - The tests need to be written in a timely fashion. Unit tests should be written JUST BEFORE the production code that make them pass!

# Only One...

## One test class per tested class

- You want to be able to quickly locate all tests for a specific class (or one test class per unit of work under test).
- If we are compelled to split unit tests for a class across multiple files, that may be an indication that the class itself is poorly designed (the class doesn't follow the SOLID principles).

## One/single concept per test

- We don't want long test functions that go testing one miscellaneous thing after another.

## One assert per test

- The first time an assert fails, it actually throws a special type of exception that is caught by the test runner. That also means no other lines below the line that just failed will be executed.
- Testing only one concern - A concern is a single end result from a unit of work: a return value, a change to system state, or a call to a third-party object.

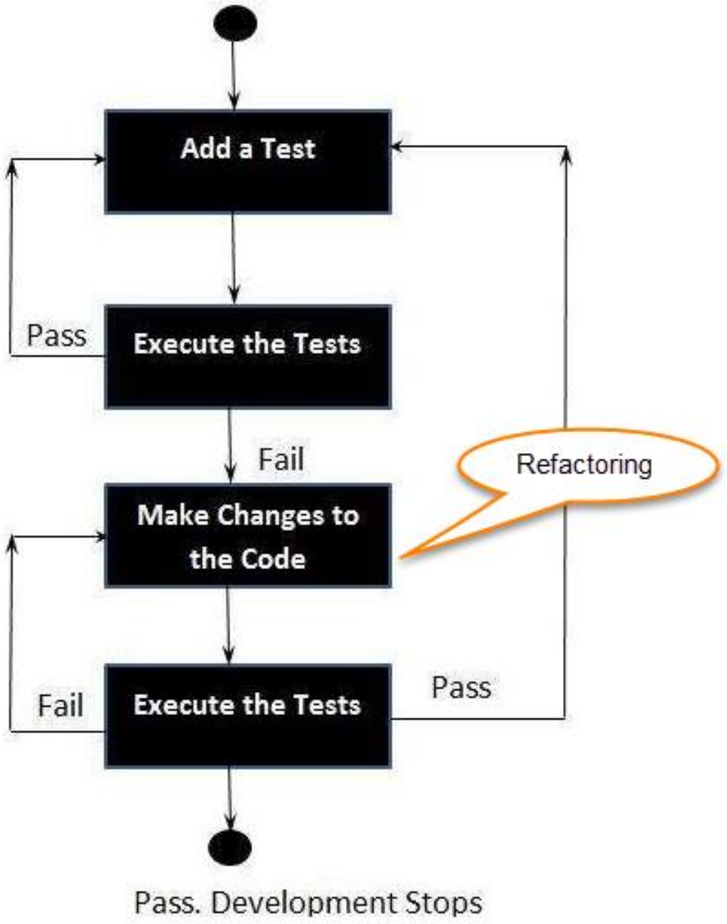
## One mock per test

- In a test where you test only one thing, there should be no more than one mock object. All other fake object will act as stubs.
- Having more than one mock per test usually means you're testing more than one thing.

# TDD - Test Driven Development

The three laws (rules) of TDD:
1. Don't write any production code until you have written a failing unit test.
2. Don't write more of a unit test than is sufficient to fail or fail to COMPILE (if that is a new method/class, if refactoring something, etc).
3. Don't write any more production code than is sufficient to pass the failing test.

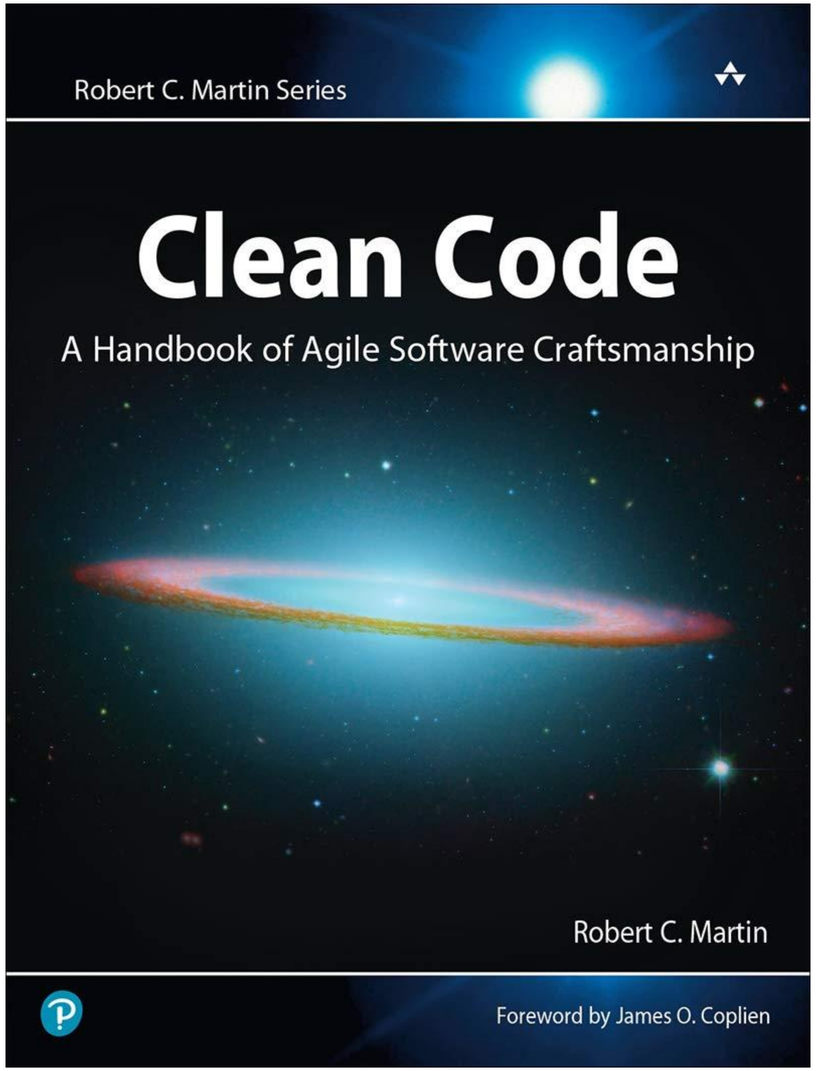
One of the biggest benefits of TDD is:
You're testing the test itself! First you're seeing the test fails and later you're seeing it passes. If you expect it to fail and it passes, you might have a bug in your test or you're testing the wrong thing.



9

# Resources

# Clean Code - Uncle Bob (Robert C. Martin)




endava