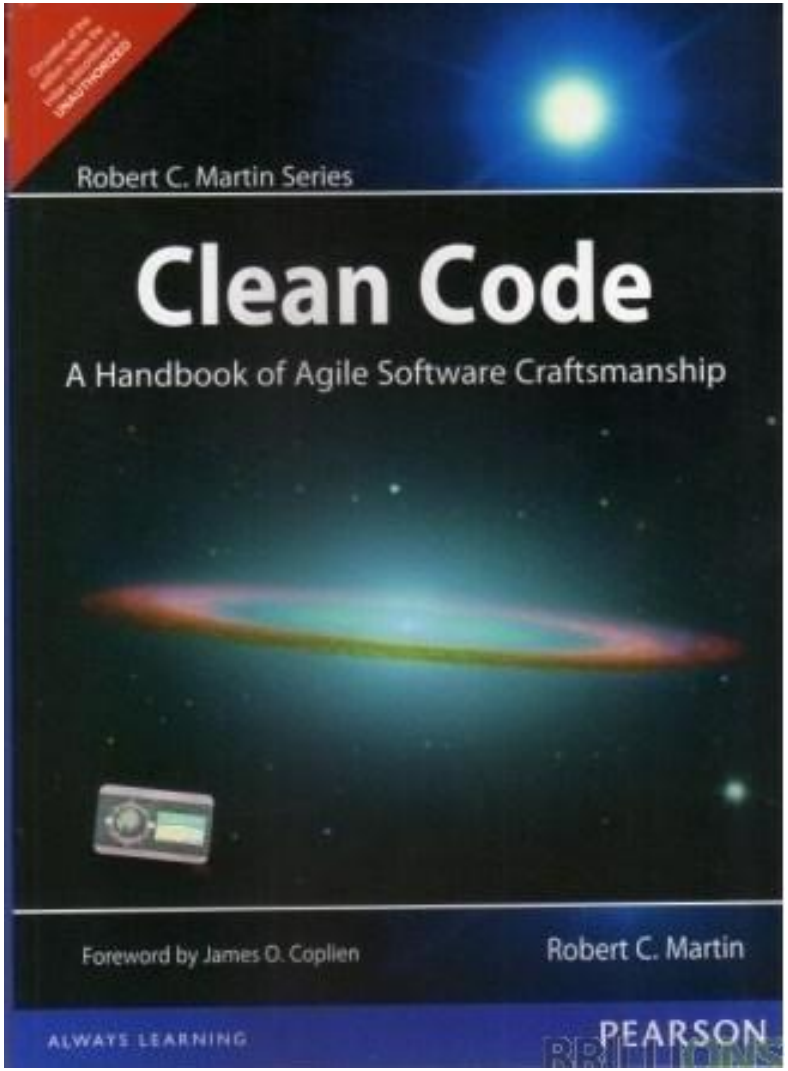
Clean Code

Matthew Renze



Robert C. Martin Series

Clean Code

A Handbook of Agile Software Craftsmanship

Foreword by James O. Coplien

Robert C. Martin

ALWAYS LEARNING

PEARSON

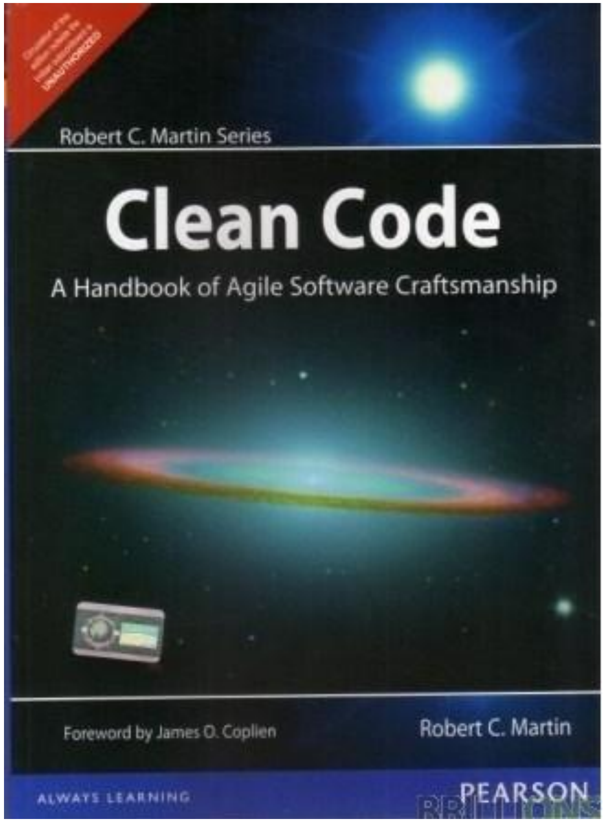
Robert C. Martin (aka. Uncle Bob)

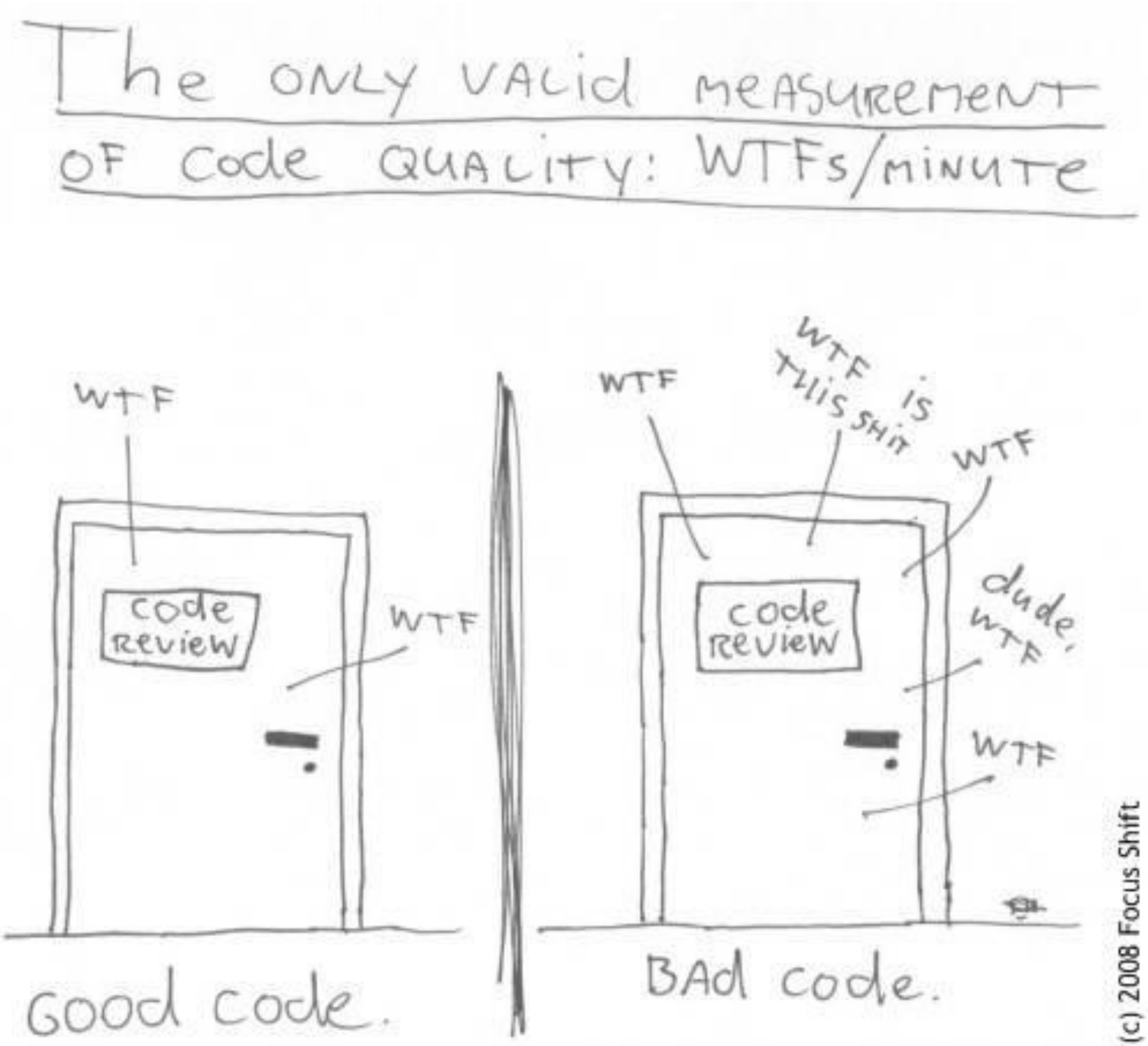
About Me

- Independent software consultant
- 14 years of professional Agile software development experience
- Data-driven desktop, server, and web apps
  - Web-based GIS data warehouse
  - Energy data ETL application
  - Global data management system
  - Intelligent lighting control systems
  - Open-source data explorer

Overview

- Clean Code
- Names
- Functions
- Classes
- Comments
- Process



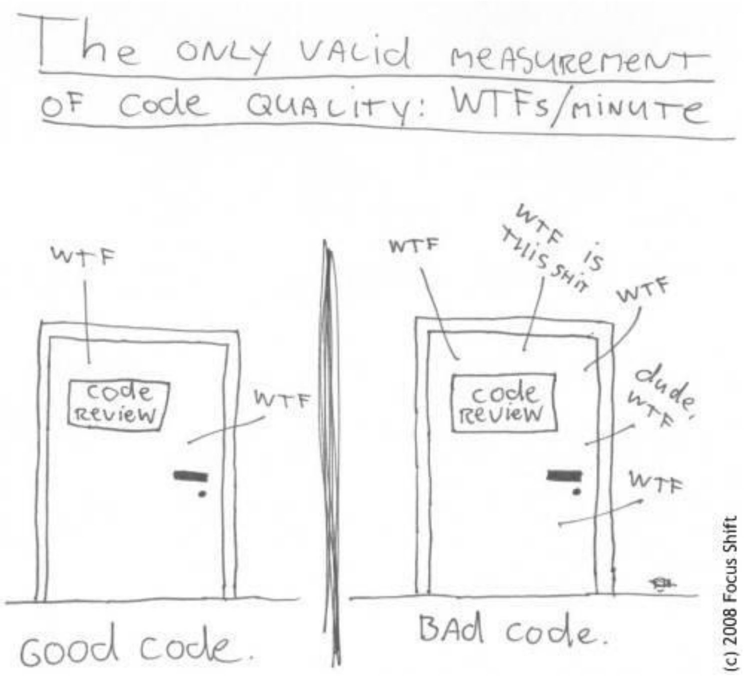


(c) 2008 Focus Shift

Source: http://www.bernie-eng.com/blog/wp-content/uploads/2011/03/code.jpg

What is Bad Code?

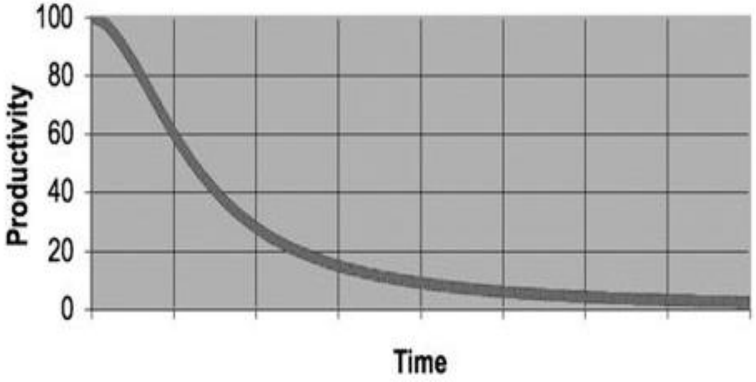
- Code that is:
  - Difficult to read
  - Difficult to maintain
  - Difficult to understand
  - Contains bugs
  - Contains surprises



The Total Cost of Owning a Mess

- Code starts clean and productivity is high
- Bad code accumulates
- Productivity decreases and eventually hits zero
- Ends with "The Grand Redesign in the Sky"
- Bad code can bring down a whole company

Productivity vs. Time



Source: Clean Code

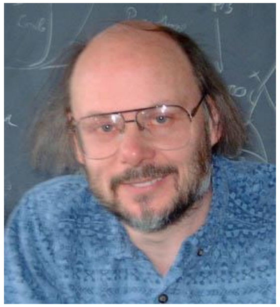
No Broken Windows

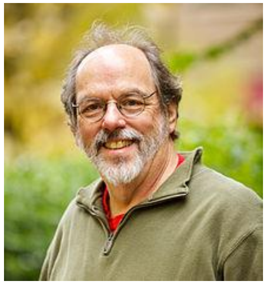
- Theory in Psychology
- No broken windows is stable
- One broken window leads to more broken windows
- One broken window is a tipping point

Source: http://www.outsidethebeltway.com/ earthquakes-economists-and-the-broken-window-fallacy

The Way We Avoid a Mess is By Keeping Our Code Clean

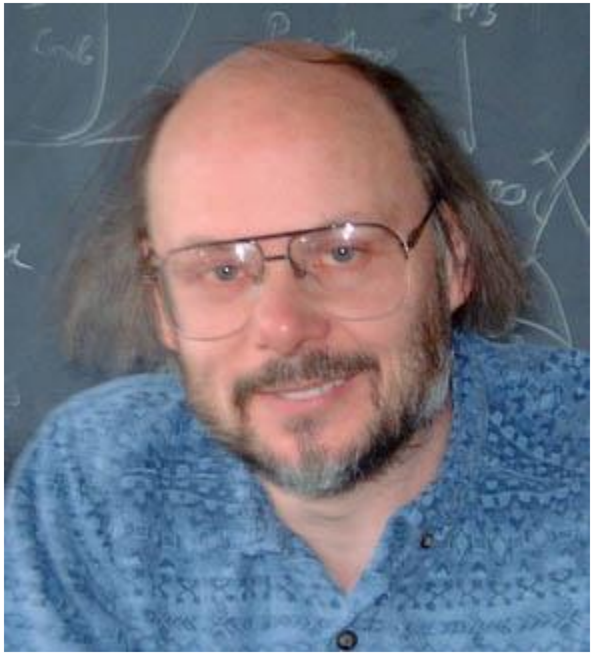
What is Clean Code?





What is Clean Code?

- Logic should be straight-forward
- Dependencies minimal to ease maintenance
- Error handling complete
- Performance optimal
- Does one thing well

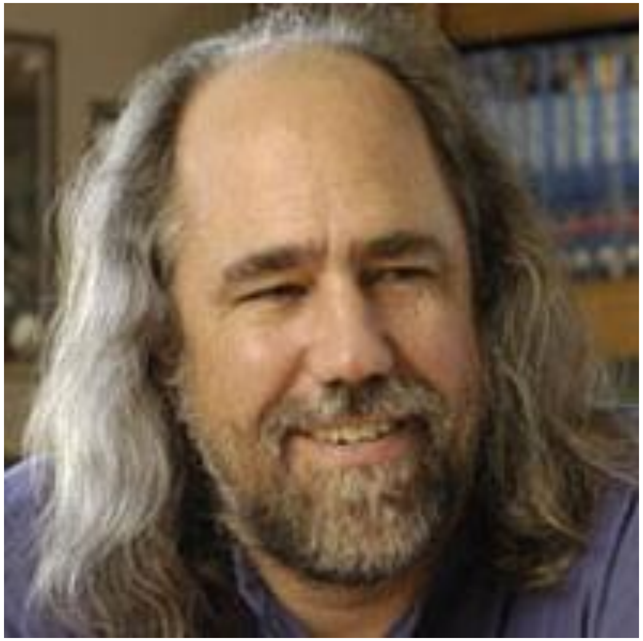


Bjarne Strostrup

Inventor of C++

What is Clean Code?

- Simple and direct
- Reads like well-written prose
- Never obscures the designers intent
- Full of crisp abstractions
- Contains straight-forward lines of control



Grady Booch

Co-inventor of UML

What is Clean Code?

- Readable by others
- Has unit and acceptance tests
- Has meaningful names
- Provides one way of doing one thing
- Has minimal dependencies

Dave Thomas

Co-Author of
The Pragmatic Programmer

What is Clean Code?

It has one overarching property: It looks like it was written by someone who cares

Michael Feathers

Author of

Working Effectively with Legacy Code

What is Clean Code?

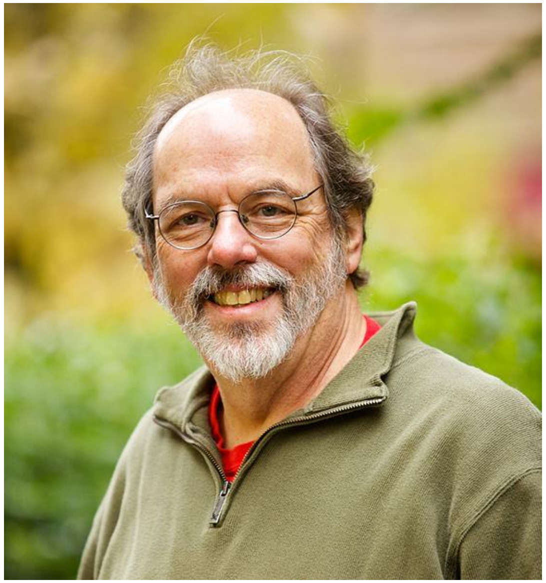
- Runs all the tests
- Expresses all the design ideas in the system
- Minimizes the number of entities
- Minimizes duplication
- Expresses ideas clearly

Ron Jeffries

Co-inventor of XP

What is Clean Code?

"You know you are working on clean code when each routine you read turns out to be pretty much what you expected."



Ward Cunningham

Inventor of the Wiki
Co-inventer of XP

What is Clean Code?

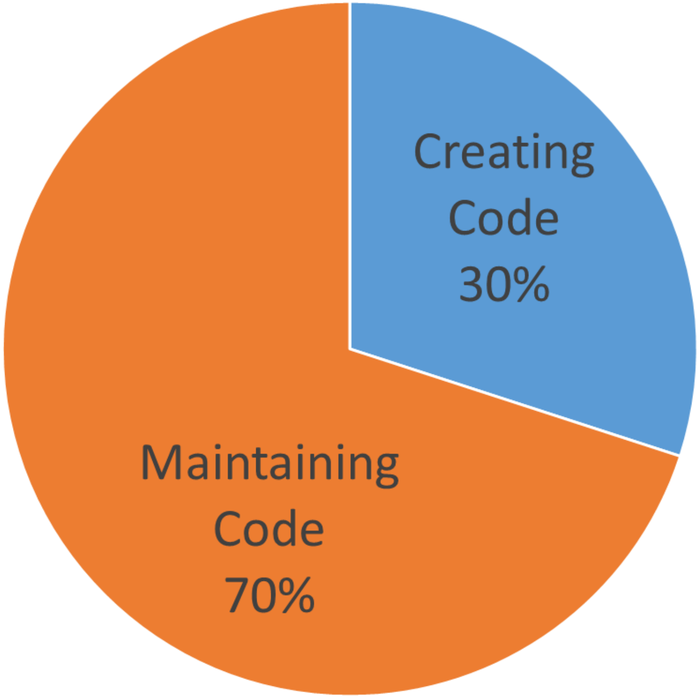
- Clean code is:
  - Simple
  - Readable
  - Understandable
  - Maintainable
  - Testable
- Clean code is a philosophy of writing code for the reader

Matthew Renze

Genius, Billionaire, Playboy,
Philanthropist :)

Why Should We Invest in Clean Code?

- Development costs:
  - Creating code ≈ 30%
  - Maintaining code ≈ 70%

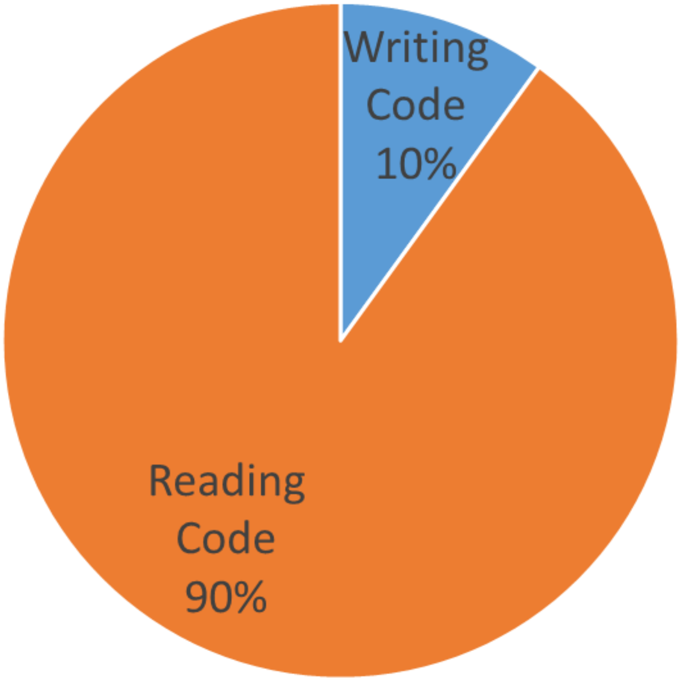


Sources:

- Barry Boehm - Software Engineering Economics, Prentice Hall
- Schach, R., Software Engineering, Fourth Edition, McGraw-Hill
- Glass, Robert, Frequently Forgotten Fundamental Facts about Software Engineering

Why Should We Invest in Clean Code?

- Time spent:
  - Writing Code ≈ 10%
  - Reading code ≈ 90%



Source: Clean Code

Clean Code is an Investment

- Making code easy to read makes it easier to:
  - Write new code
  - Maintain old code
- Invest in code readability

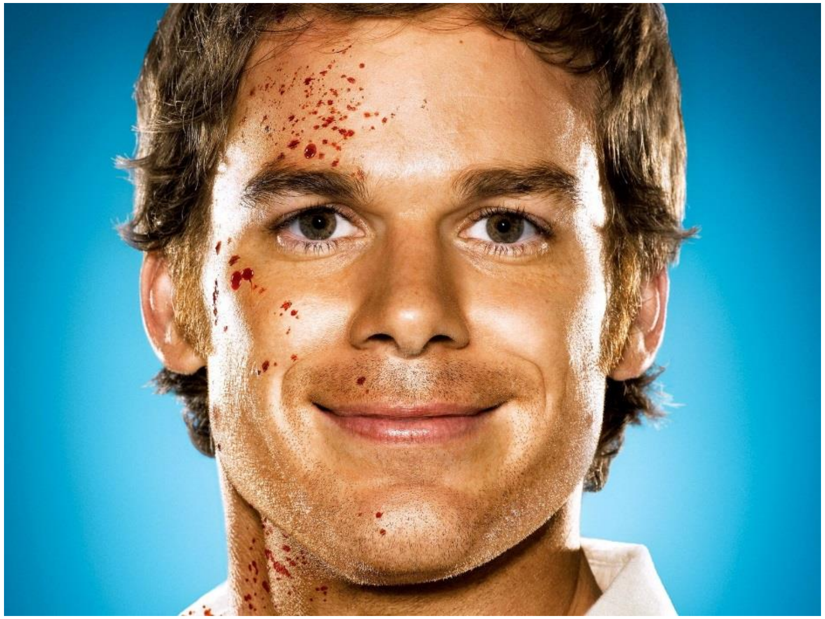
How Do You Write Clean Code?

- Write code for the *reader*
- Not for the *author*
- Not for a *machine*

How Do You Write Clean Code?

“Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live!”

- Author Unknown



Names

Choose Names Thoughtfully

- Names are important
- We use names to:
  - Organize ideas
  - Communicate ideas
  - Search for things
- Choose your names well and appropriately

Use Intention-Revealing Names

- Names should reveal their intent
- Avoid unclear names
- If you need a comment to understand a name, then it’s a bad name

```
// Bad
int d;  // days in queue

// Good
int daysInQueue;
```

```
// Bad
private int Process();

// Good
private int ParseIdFromFile();
```

Use Names from the Domains

- Problem Domain
  - Business language
- Solution Domain
  - Developer language

```
// Problem Domain
public class Customer {}

public void AddAccount();


// Solution Domain
public class Factory {}

public void AddToJobQueue();


// Both Domains
public class CustomerFactory {}

public void AddAccountToJobQueue();
```

Avoid Disinformation

- Avoid misleading info
- Avoid subtle differences
- Avoid easy to confuse names

```
// Bad
ISet<Customer> customerList;

// Bad
ControllerForSavingOfCustomers()
ControllerForStorageOfCustomers()

// Bad
int a = l;
if (O == l)
    a = 1;
else
    l = 0;
```

Use Pronounceable Names

- Our brains think in spoken words
- Use names that are pronounceable
- Avoid acronyms and abbreviations unless they are well known

```
// Bad – Not pronounceable names
public class DtaRcrd102
{
    private DateTime genymdhms;
    private DateTime modymdhms;
    private string pszqint = "102";
}

// Good – Pronounceable names
public class Customer
{
    private DateTime generationTimestamp;
    private DateTime modificationTimestamp;
    private string recordId = "102";
}
```

Avoid Encodings

- No Hungarian notation
- No module prefixes
- No interface prefixes

```
// Bad - Hungarian notation
private Button btnClickMe;

private int intTotal = 0;


// Bad - Module prefixes
private int m_SomeField = 0;


// Bad - Interface prefixes
public interface IShapeFactory


// Good – No interface prefix
public interface ShapeFactory

public class CircleFactory
    : ShapeFactory
```

Class Names

- Should be a noun or noun phrase
- Avoid fuzzy names

```
// Good - Noun or noun phrase
public class Customer
public class Account
public class AddressParser


// Bad - Fuzzy names
public class Manager
public class Processor
public class Stuff
```

Method Names

- Should be a verb or verb phrase
- Avoid fuzzy names
- Boolean functions should be predicates

```
// Good - Verb or verb phrase
public void AddCustomer()
public void DeleteAccount()
public string ParseAddress()


// Bad - Fuzzy names
public string Process()
public void DoWork()


// Good - Boolean predicates
public bool IsValid()
public bool HasAccount()
```

Length of Variable Names Should Increase with Scope

- Range variables
- Method variables
- Field variables
- Global variables

```
// Good - Very short range variable names
for (int i = 0; i < 10; i++) {}

list.Sum(p => p.GetAmount());


// Good - Short method variable names
var balance = GetAccountBalance();


// Good - Longer field variable names
private int totalAccountBalance = 0;


// Good - Very long global variable
public int totalBalanceInAllBankAccounts;
```

Length of Method Names Should Decrease with Scope

- Public method
- Private methods

```
// Good - Short public method names
public void GetCustomers();

public void Save();


// Good - Longer private method names
private string ParseHtmlFromFile()

private int GetIdFromAccountHolder()
```

Length of Class Names Should Decrease with Scope

- Public class
- Private class
- Derived class

```
// Good - short public class name
public class Account

// Good - longer private class name
private class AccountNumberGenerator

// Good - longer derived class name
public class SavingsAccount : Account
```

Functions

Functions Should Be Small

- Small functions are:
  - Simple
  - Easier to read
  - Easier to understand
  - Easier to test
  - Contain less bugs
  - Self-documenting

How Small?

- Most evidence says:
  - Less than 20 lines
- Uncle Bob says:
  - Less than 10 lines
  - Average around 3 to 6 lines or so
- Yes, I know… it sounds impossible and crazy

Large Functions are Where Classes Go to Hide

Functions Should Do One Thing

- Single-Responsibility Principle
- Do one thing well
- Avoid “and” / “or” in explanation of purpose
- Keeps functions small

Source: http://www.wengerna.com/giant-knife-16999

One Level of Abstraction per Function

- Functions should only operate at one level of abstraction
- Separation of concerns
- Separate levels of abstraction into separate functions

```
// Good - Separate levels of abstraction
public Html RenderHtml()

private string RenderHtmlBody()

private string RenderHtmlElement()

private char RenderHtmlElementClosingTag()
```

Minimize the Number of Arguments

- Arguments add complexity
- Minimize the number of arguments
- More than three then group into a logically coherent object

```
// Try to minimize the # of arguments
public void GetNiladic() {}

public void GetMonadic(object arg1)

public void GetDyadic(object arg1, object arg2)

public void GetTriadic(
    object arg1, object arg2, object arg3)

public void GetPolyadic(
    object arg1, object arg2, object arg3, ...)
```

Avoid Flag Arguments

- Flag arguments use Boolean parameter to switch behavior
- Create two methods instead
- Make behavior explicit

```
// Bad – Flag arguments
public void Render(bool useColor)

// Good – No flag arguments
public void RenderInColor()
public void RenderInGrayScale()
```

Avoid Output Arguments

- Functions should:
  - Take parameters as their input
  - Return a single value as their output
- Output arguments:
  - Are more difficult to reason about
  - Force you to do a double-take

```
// Bad - Uses ‘out’ argument
public void AppendFooter(out Report report)

AppendFooter(out report);


// Good - No ‘out’ argument
public ReportBuilder AppendFooter()

reportBuilder.AppendFooter();
```

Avoid Side Effects

- Command methods should:
  - Modify state
  - *Not* return a value
- Query methods should:
  - Return a value
  - *Not* modify state
- Query modified state is called a “side effect”
- Side effects are lies!

May cause Dizziness
May cause headache
DO NOT take with Nitrates.
Some medicines can change the effect of this medicine. Check with your doctor or pharmacist before taking any other medicine

Command-Query Separation

Command

- Does something
- Changes state
- Should not return a value (ideally)

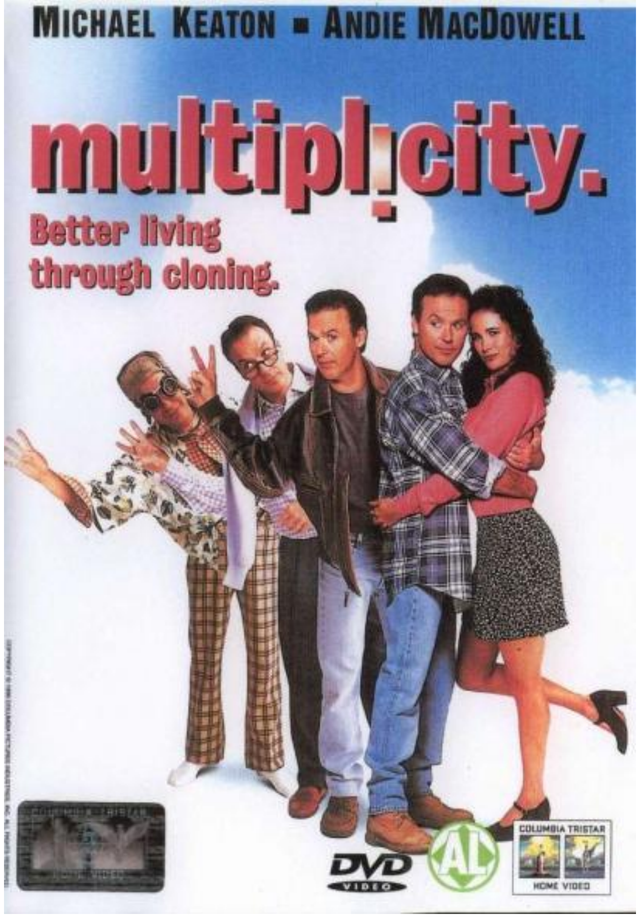
Query

- Answers a question
- Does not change state
- Returns a value

Avoid mixing the two!

Avoid Duplication

- Duplication is the root of all evil in software
- Most software paradigms were created to minimize duplication
- Functions exist to avoid duplication
- Don’t Repeat Yourself!


MICHAEL KEATON ■ ANDIE MACDOWELL
multiplicity.
Better living through cloning.
DVD VIDEO
AL
COLUMBIA TRISTAR HOME VIDEO

Source: Sony Pictures Home Entertainment

Use Functions to Enhance Readability

- Functions are named chunks of code
- Use functions to name cohesive units of code
- This enhances code readability

```
// Bad – One giant chunk of code
public void CreateReport()
{
 ... Giant block of code ...
}

// Good – Uses small named functions
public void CreateReport()
{
    CreateHeader();
    CreateBody();
    CreateFooter();
}
```

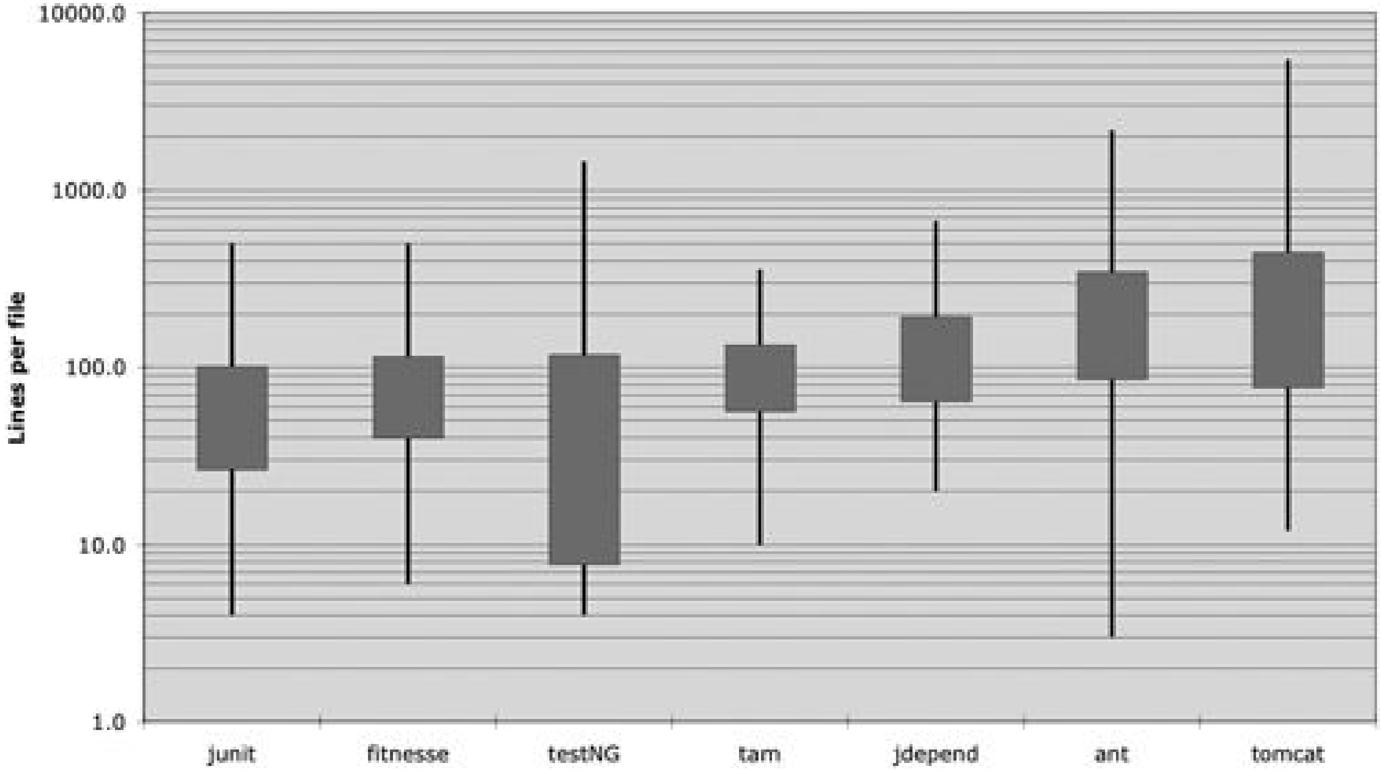
Classes, Objects, and Data Structures

Classes Should Be Small

- Similar benefits as small functions
- Single-Responsibility Principle
  - Slightly different than SRP for functions
  - One Reason to Change
  - Cohesion / Coupling

How Small?

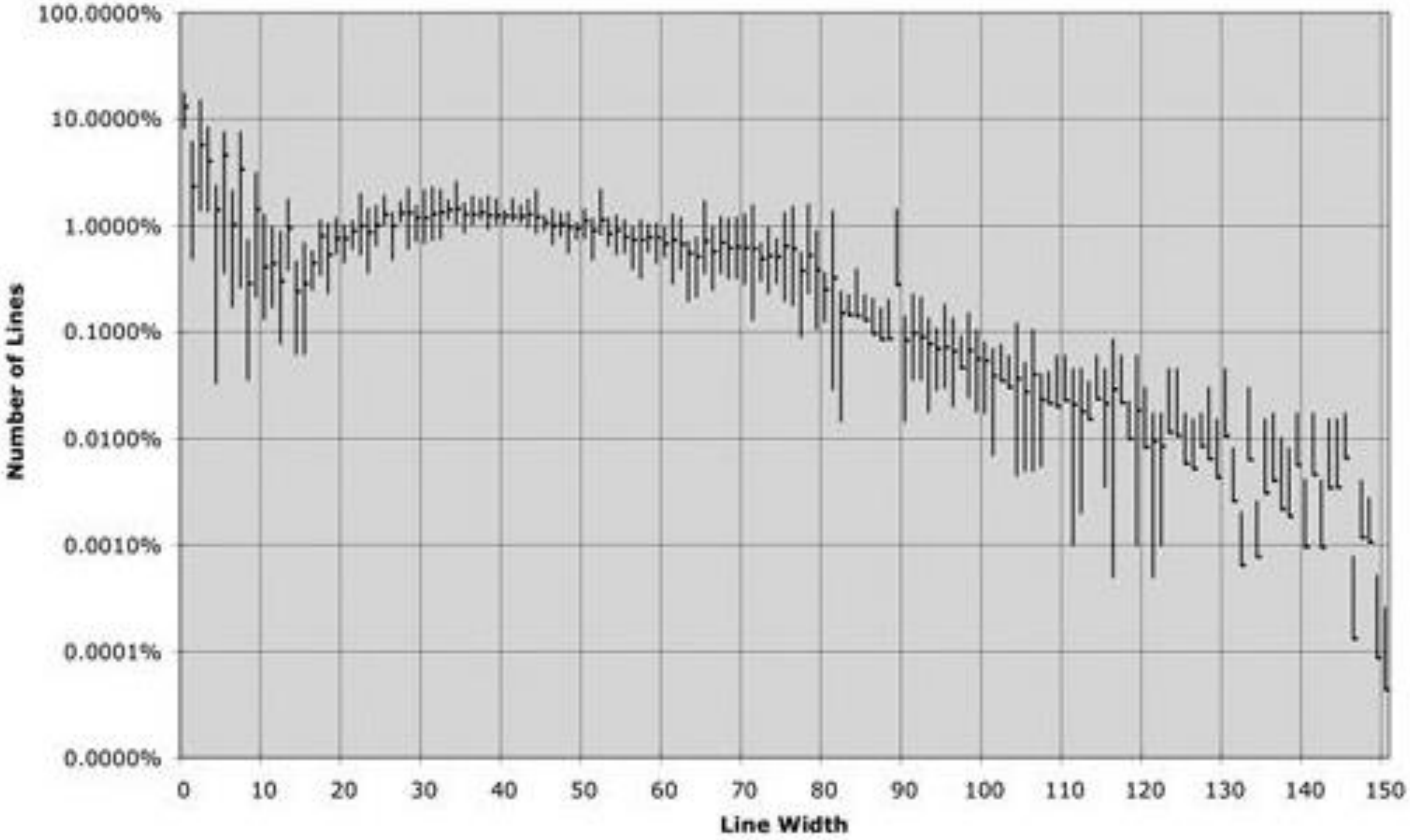
Figure 5-1 File length distributions LOG scale (box height = sigma)



Source: Clean Code

Classes Should Be Narrow

Figure 5-2 Java line width distribution



Source: Clean Code

Law of Demeter

- A method *f* of class *C* should only call the methods of:
  - *C*
  - An object created by *f*
  - An object passed into *f*
  - An instance variable in *C*
- Talk to your friends… don't talk to strangers

```
// Bad - Law of Demeter violation
var rent = customer.Pocket.Wallet
     .Money.GetRentMoney();

// Good - No violation
var rent = customer.GetRentMoney();
```

Source: Athens Banner-Herald

Object vs. Data Structure

Object

- Hides data
- Exposes behavior

```
public class Rectangle
{
    private double x;
    ...
    public double GetX() {
        return x;
    }
    ...
    public double GetArea() {
        return width * height;
    }
}
```

Data Structure

- Exposes data
- Has no behavior

```
public struct Rectangle
{
    public double X;
    public double Y;
    public double Width;
    public double Height;
}
```

Avoid Hybrid Object/Structures

- Half object / half data structure hybrids are bad
- Typically the result of confused design
- They are the worst of both worlds

Source: http://www.layoutsparks.com/1/147428
/alien-resurrection-scary-dreadful-31000.html

Have a Consistent Order to Your Classes

1. Constants
2. Fields
3. Properties
4. Constructors
5. Public Methods
6. Private Methods

```
public class SomeClass
{
    private const int SomeConst = 123;

    private int _someField;

    private int SomeProperty
    {...}

    public SomeClass()
    {...}

    public void DoSomething()
    {...}

    private void DoSomethingElse()
    {...}
}
```

Choose the Right Abstractions for Classes

- Learn the design patterns
- Patterns are a language
- Class name should imply the pattern
- Provides a crisp set of abstractions

- Controller
- View
- View-Model
- Repository
- Factory
- Builder
- Adapter

Other Practices for Classes

- DRY Principle
- High Cohesion
- Low Coupling
- Dependency Injection
- Testability

Comments

Comments Represent a Failure

- Comments represent a failure to express ourselves in code
- Ideally we could explain everything in code
- Unfortunately this is not true
- Comments are used to compensate for this failure
- Comments are, at best, a necessary evil

Source: http://a.tgcdn.net/images/products/zoom/no_comment.jpg

Obsolete Comments Lie

- Comments have a tendency to become obsolete
- Obsolete comments are lies
- Code is truth
- Incorrect comments are worse than no comments at all

Source: http://pintaw.com/lie-to-me-tim-roth-wallpaper-2/

Explain Yourself in Code

It is better to explain intent in code than in comments

```
// Bad - Code explained in comment

// Check to see if the employee is eligible
for full benefits
if ((employee.Flags && HOURLY_FLAG)
    && (employee.Age > 65))


// Good - Code explains itself

If (employee.IsEligibleForFullBenefits())
```

Bad Comments

- Redundant comments
- Misleading comments
- Journal comments
- Closing brace comments
- Commented out code

```
// All of these comments are bad

// Opens the file
var file = File.Open();

// Returns day of month
private int GetDayOfWeek()

// 08-07-2013 – Fixed Bug (MLR)

Main()
{
 …
} // end main

// Zombie Code
// if (a == 1)
//     b = c + 1
```

Necessary Comments

- Legal comments
- Clarification
- Warnings
- TODO comments
- Public API documentation

```
// Copyright © 2013 Matthew Renze

// Trim is necessary to prevent a
// search term mismatch

// Warning: Slow running test

// TODO: Refactor to factory

/// <summary>
/// Opens the file for reading
/// </summary>
```

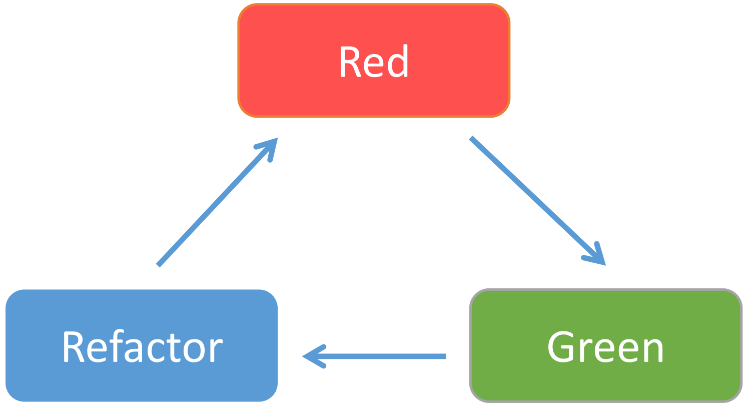
The Best Comment is No Comment at All

(but only if your code clearly explains itself)

The Process

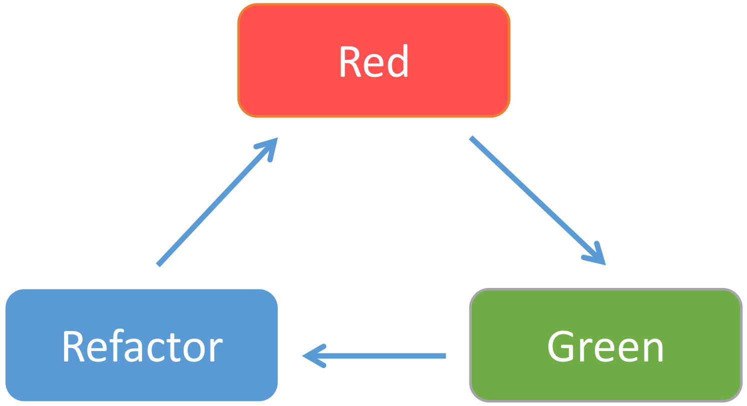
The Principles

- Test-Driven Development (TDD)
- Simplicity (KISS)
- Continuous Refactoring



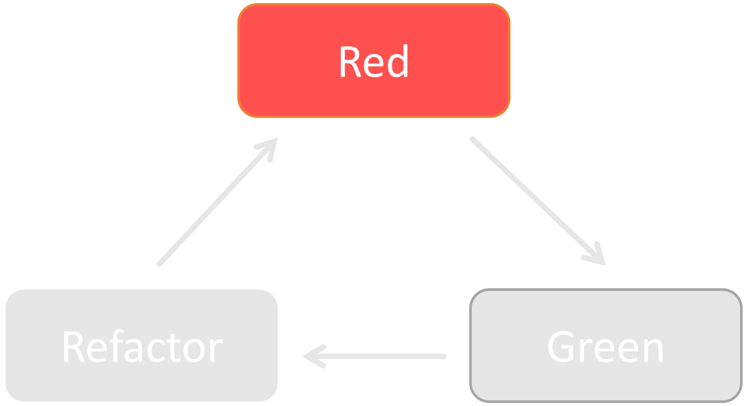
Test-Driven Development Process

1. Create a failing unit test
2. Code the simplest thing that could possibly work
3. Refactor until the code is clean



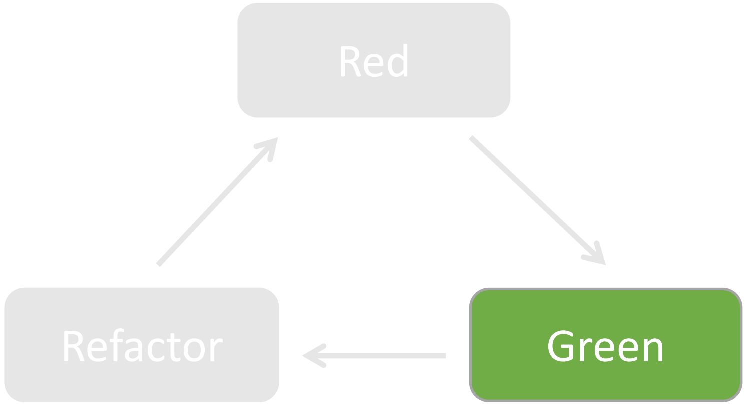
Test-Driven Development

- Starts with a test
- Tests drive the design
- Code evolves over time
- TDD produces:
  - Testable code
  - Maintainable code
  - Reliable code
  - Self-documenting code



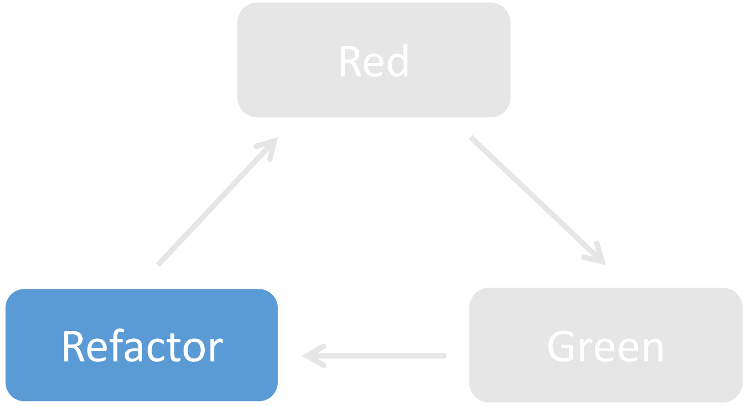
Simplicity

- We only code the simplest thing that makes the test pass
- Complexity is our enemy
- Just get the code working first



Refactoring

- Having working code is *not* the last step
- Refactor the code until it is clean
- Continuous refactoring is necessary for clean code
- All creative endeavors are iterative processes



Follow the Boy Scout Rule

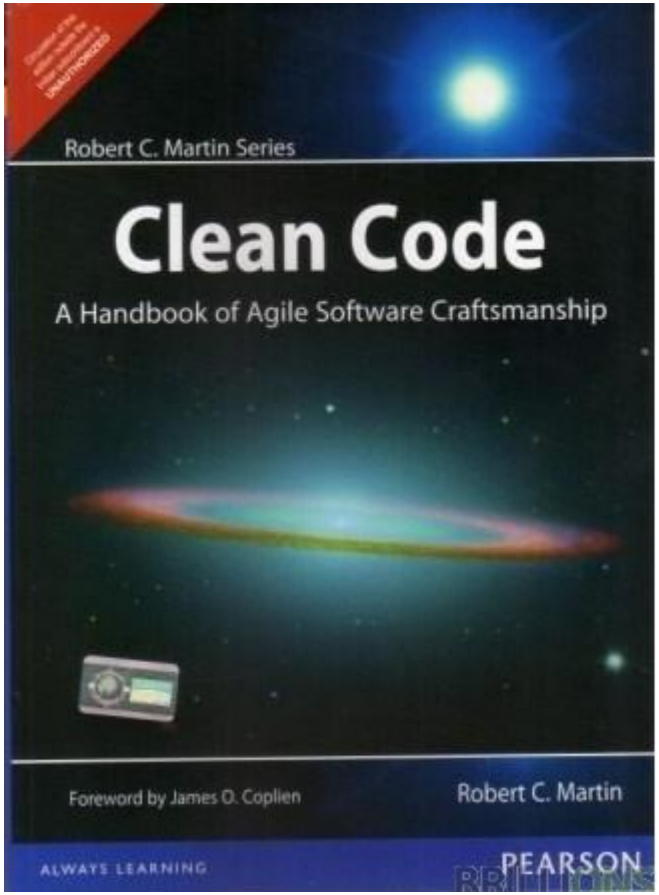
“Leave the campground just a little bit cleaner than you found it.”

– adapted from Robert Stephenson Smyth Baden-Powell’s farewell message to the scouts: “*Try and leave this world a little better than you found it.*”

Conclusion

Conclusion

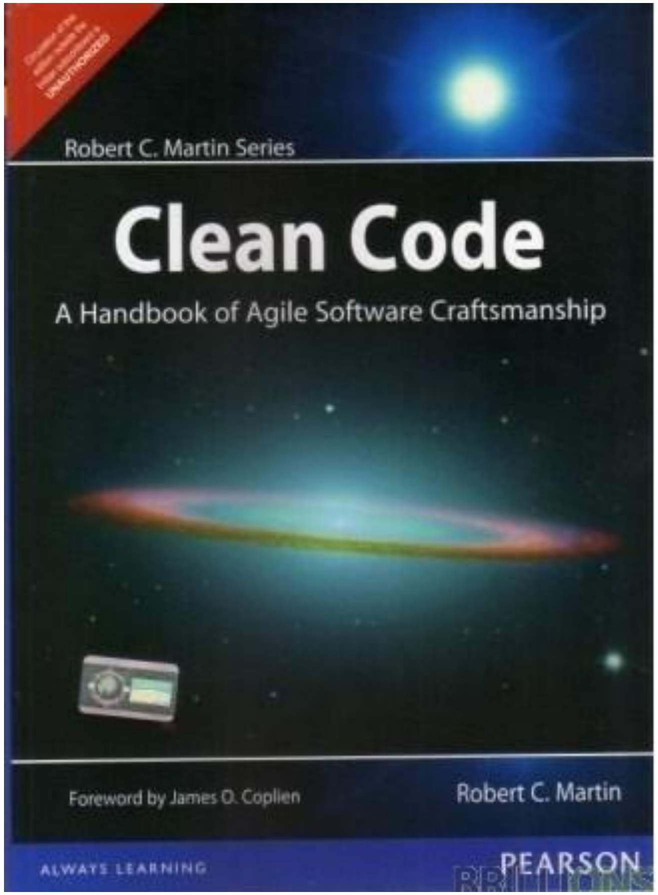
- Clean code is:
  - Simple
  - Readable
  - Understandable
  - Maintainable
  - Testable
- Clean code is a philosophy of writing code for the reader



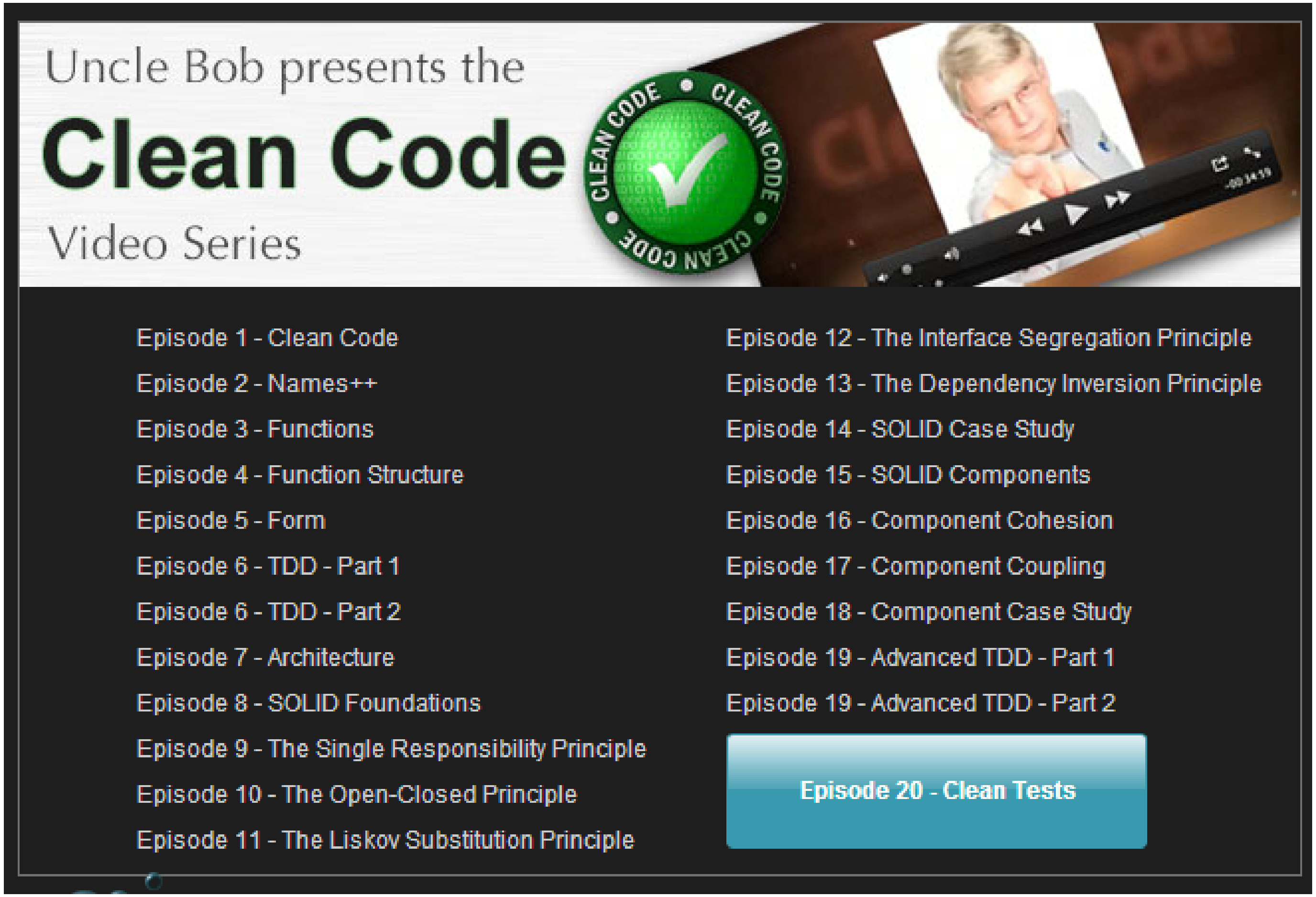
Conclusion

- Use intention revealing names
- Classes and functions should be small and do one thing well
- Use comments to express a failure to communicate in code
- The process is:
    1. Test First (TDD)
    2. Simplest solution
    3. Continuously refactor

Where to Go Next…



Where to Go Next…



Uncle Bob presents the

Clean Code

Video Series

Episode 1 - Clean Code

Episode 2 - Names++

Episode 3 - Functions

Episode 4 - Function Structure

Episode 5 - Form

Episode 6 - TDD - Part 1

Episode 6 - TDD - Part 2

Episode 7 - Architecture

Episode 8 - SOLID Foundations

Episode 9 - The Single Responsibility Principle

Episode 10 - The Open-Closed Principle

Episode 11 - The Liskov Substitution Principle

Episode 12 - The Interface Segregation Principle

Episode 13 - The Dependency Inversion Principle

Episode 14 - SOLID Case Study

Episode 15 - SOLID Components

Episode 16 - Component Cohesion

Episode 17 - Component Coupling

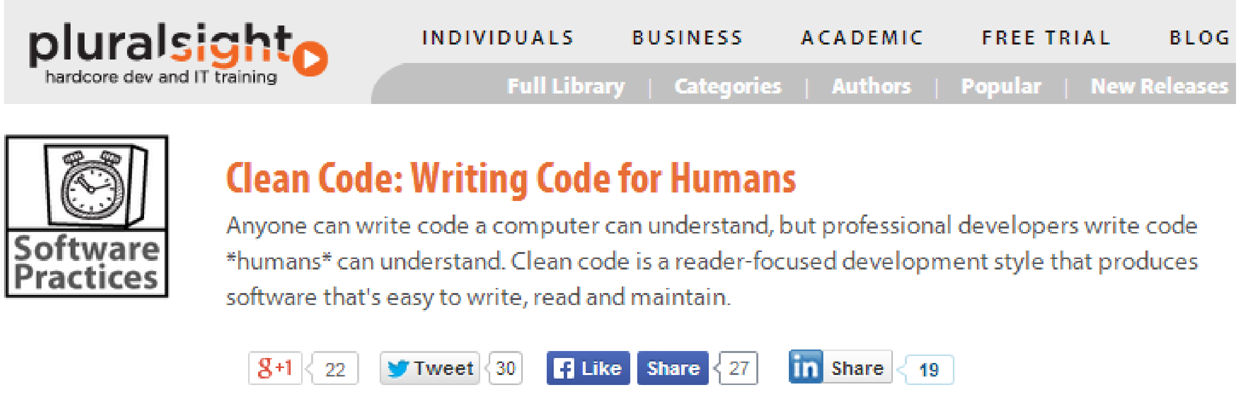
Episode 18 - Component Case Study

Episode 19 - Advanced TDD - Part 1

Episode 19 - Advanced TDD - Part 2

Episode 20 - Clean Tests

Where to Go Next…



Clean Code: Writing Code for Humans

Anyone can write code a computer can understand, but professional developers write code *humans* can understand. Clean code is a reader-focused development style that produces software that's easy to write, read and maintain.

http://pluralsight.com/training/Courses/TableOfContents/writing-clean-code-humans

Uncle Bob Wants You:

"To leave the campground just a little bit cleaner than you found it."

Contact Info

Matthew Renze
matthew@renzeconsulting.com

Renze Consulting
www.renzeconsulting.com

Data Explorer
http://www.data-explorer.com