```
// When I wrote this, only God and I understood what I was doing
// Now, God only knows
                                                                  // Magic. Do not touch.
// somedev1 - 6/7/02 Adding temporary tracking of Login screen
// somedev2 - 5/22/07 Temporary my ass
                                                                  <!-- Here be dragons -->
// I am not responsible of this code. They made me write it, against my will.

// Dear future me. Please forgive me. I can't even begin to express how sorry I am.

// no comments for you. it was hard to write so it should be hard to read

// John! If you'll svn remove this once more, I'll shut you, for God's sake!
// That piece of code is not “something strange”! That is THE AUTH VALIDATION.

// This procedure is really good for your dorsolateral prefrontal cortex.

// Abandon all hope ye who enter beyond this point
                                                       // Catching exceptions is for communists
// Peter wrote this, nobody knows what it does, don't change it!

// if i ever see this again i'm going to start bringing guns to work
                                                                 // Happy debugging, suckers

const int TEN=10; // As if the value of 10 will fluctuate...
```

Evosoft Hungary
Mind That Code

Contents

Code maintenance costs

Code maintenance costs

The Equation of Software Design:

$$D = \frac{V}{E}$$

D - Desirability
V - Value
E - Effort

Code maintenance costs

The Equation of Software Design:

$$D = \frac{V_n + V_f}{E_i + E_f}$$

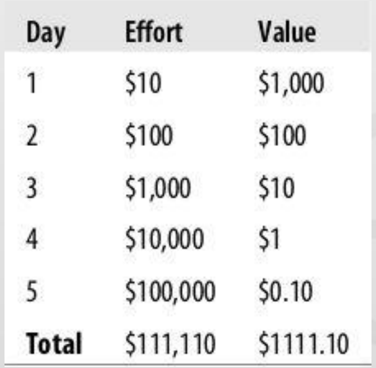
D	- Desirability
Vn	- Value now
Vf	- Future value
Ei	- Effort of implementation
Em	- Effort of maintenance

Code maintenance costs

The Equation of Software Design:

$$D = \frac{V_n + V_f}{E_i + E_f}$$

D	- Desirability
Vn	- Value now
Vf	- Future value
Ei	- Effort of implementation
Em	- Effort of maintenance



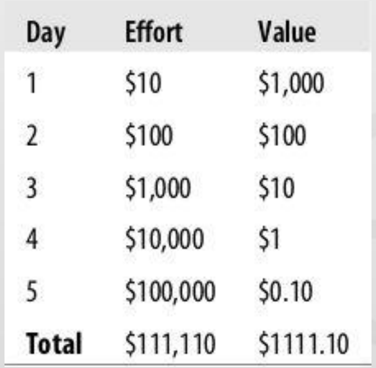
Day	Effort	Value
1	$10	$1,000
2	$100	$100
3	$1,000	$10
4	$10,000	$1
5	$100,000	$0.10
Total	$111,110	$1111.10

Code maintenance costs

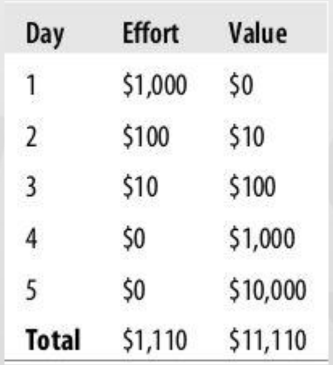
The Equation of Software Design:

$$D = \frac{V_n + V_f}{E_i + E_f}$$

D	- Desirability
Vn	- Value now
Vf	- Future value
Ei	- Effort of implementation
Em	- Effort of maintenance



Day	Effort	Value
1	$10	$1,000
2	$100	$100
3	$1,000	$10
4	$10,000	$1
5	$100,000	$0.10
Total	$111,110	$1111.10



Day	Effort	Value
1	$1,000	$0
2	$100	$10
3	$10	$100
4	$0	$1,000
5	$0	$10,000
Total	$1,110	$11,110

Code maintenance costs

The Equation of Software Design:

$$D = \frac{V_n + V_f}{E_i + E_f}$$

D	- Desirability
Vn	- Value now
Vf	- Future value
Ei	- Effort of implementation
Em	- Effort of maintenance

Day	Effort	Value
1	\$10	\$1,000
2	\$100	\$100
3	\$1,000	\$10
4	\$10,000	\$1
5	\$100,000	\$0.10
Total	\$111,110	\$1111.10

Day	Effort	Value
1	\$1,000	\$0
2	\$100	\$10
3	\$10	\$100
4	\$0	\$1,000
5	\$0	\$10,000
Total	\$1,110	\$11,110

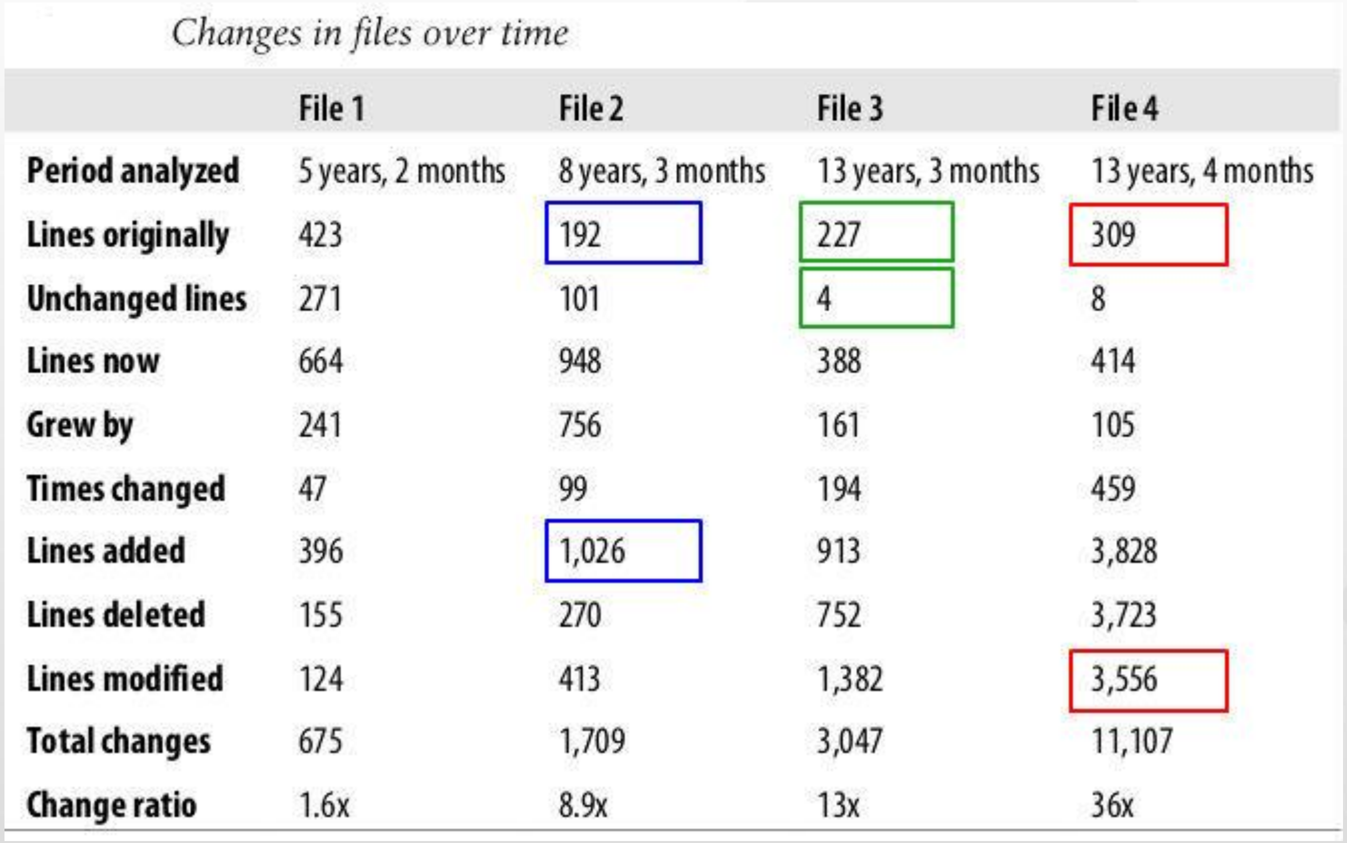
Day	Effort	Value
1	\$1	\$0
2	\$2	\$2
3	\$3	\$4
4	\$4	\$6
5	\$5	\$8
Total	\$15	\$20

$$E_m = E_u + E_c + E_t + E_d$$

Eu	- Effort of understanding
Ec	- Effort of changing
Et	- Effort of testing
Ed	- Effort of deploying

$$E_m = E_u + E_c + E_t + E_d$$

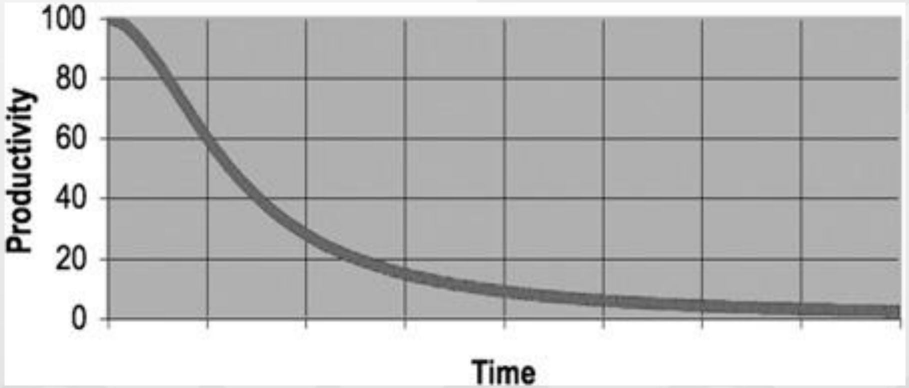
Eu - Effort of understanding
Ec - Effort of changing
Et - Effort of testing
Ed - Effort of deploying



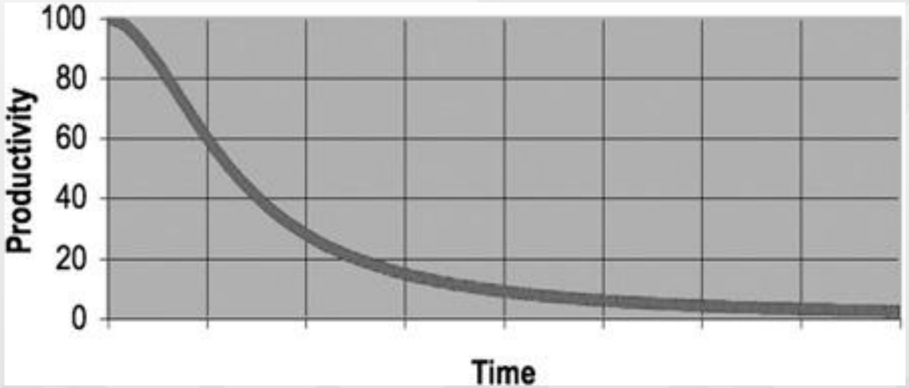
Changes in files over time

	File 1	File 2	File 3	File 4
Period analyzed	5 years, 2 months	8 years, 3 months	13 years, 3 months	13 years, 4 months
Lines originally	423	192	227	309
Unchanged lines	271	101	4	8
Lines now	664	948	388	414
Grew by	241	756	161	105
Times changed	47	99	194	459
Lines added	396	1,026	913	3,828
Lines deleted	155	270	752	3,723
Lines modified	124	413	1,382	3,556
Total changes	675	1,709	3,047	11,107
Change ratio	1.6x	8.9x	13x	36x

When software is hard to create or modify, programmers spend most of their time focusing on making things "just work," and less time focusing on helping the user.



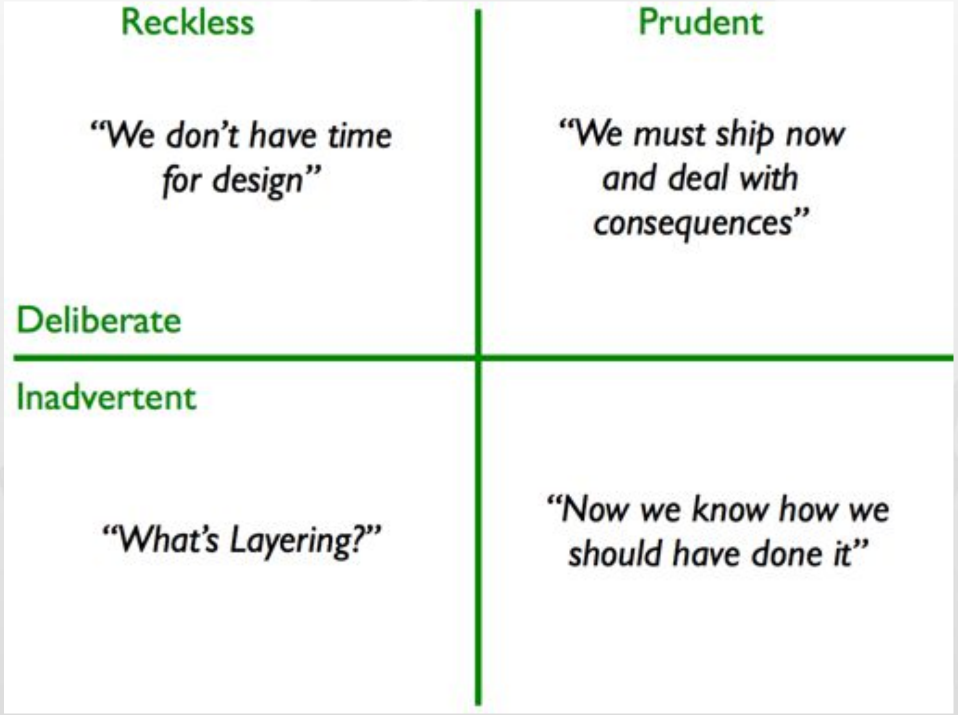
When software is hard to create or modify, programmers spend most of their time focusing on making things "just work," and less time focusing on helping the user.



This leads to **technical debts**!

Why Clean Code is important?

The Technical Debt Quadrant:


Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

LeBlanc’s law:

later = **NEVER**

LeBlanc’s law:

later = **NEVER**

This is how the software starts to rot!

LeBlanc's law:

later = **NEVER**

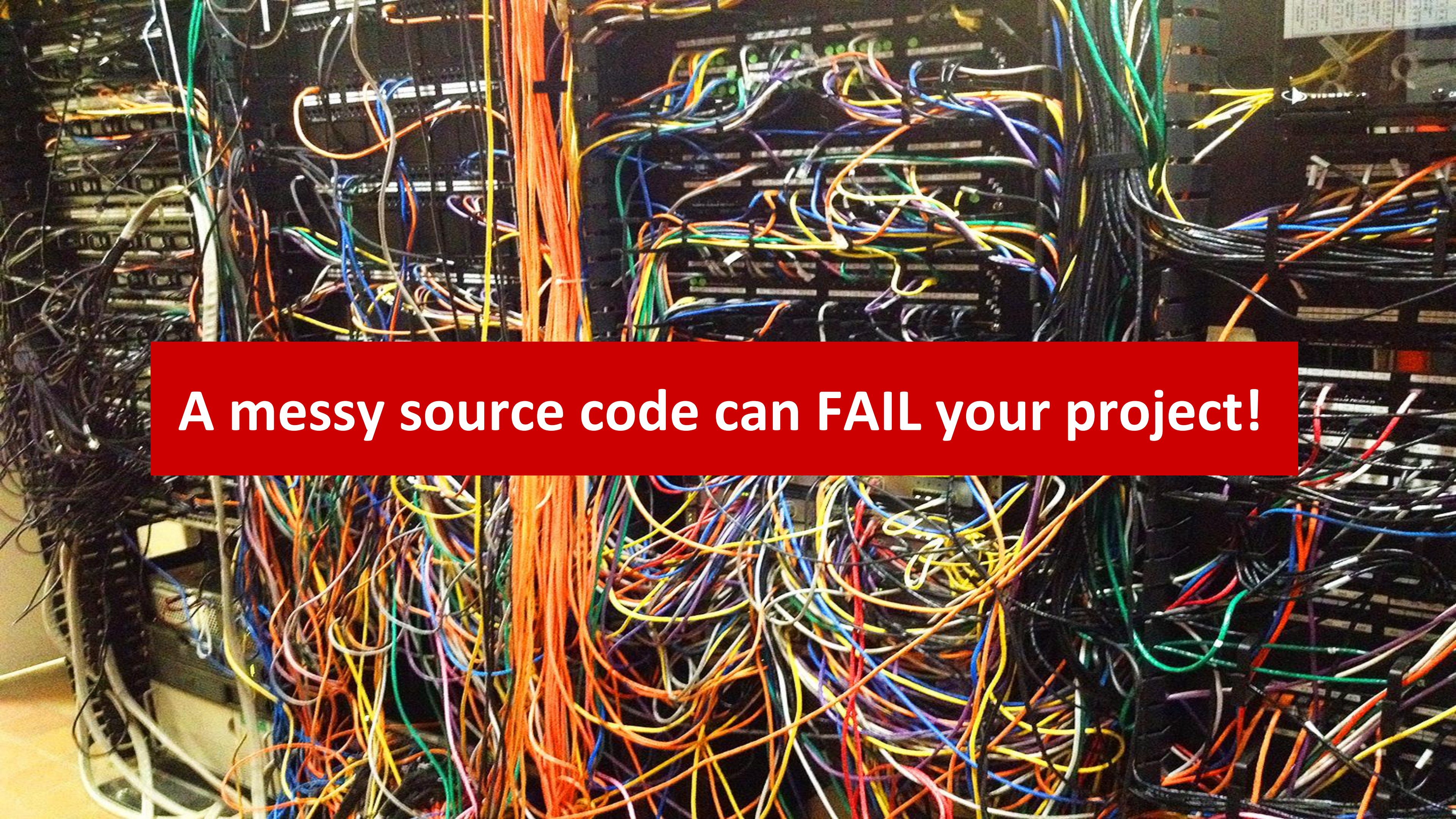
This is how the software starts to rot!

The Broken Window Theory:

Code smell

- Duplicated code
- Large class
- Cyclomatic complexity
- Downcasting
- Too many parameters
- Long method
- Big ball of mud
- Circular dependency
- Lasagna code
- Spaghetti code
- Magic numbers

- Copy and paste programming
- Premature optimization
- Reinventing the square wheel
- Tester Driven Development
- God object



A messy source code can FAIL your project!

Rushing makes you write bad code.

What are the reasons to rush?

Rushing makes you write bad code.

What are the reasons to rush?

- managers

Rushing makes you write bad code.

What are the reasons to rush?

- managers
- customers

Rushing makes you write bad code.

What are the reasons to rush?

- managers
- customers
- impossible schedules

Rushing makes you write bad code.

What are the reasons to rush?

- managers
- customers
- impossible schedules
- always changing requirements

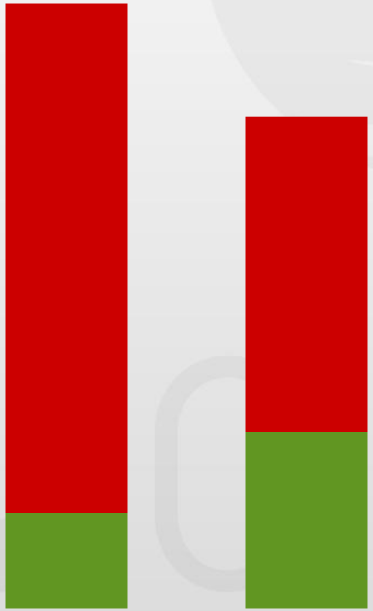
Rushing makes you write bad code.

What are the reasons to rush?

- ~~managers~~
- ~~customers~~
- ~~impossible schedules~~
- ~~always changing requirements~~

What if we could increase the productive time
by decreasing the time of reading and understanding source code?

read less



write more

How fast can you read the sentence below?

tHisisanOrmalseNtencewitHnorMalwordsthAtevErybOdycAnunderStaNd.

What about this?

This is a normal sentence with normal words that everybody can understand.

Smart developers write code only they can understand.

Smart developers write code only they can understand.

Professionals write clean code!

You are an @author who is responsible for communicating well with his readers.

You are an @author who is responsible for communicating well with his readers.

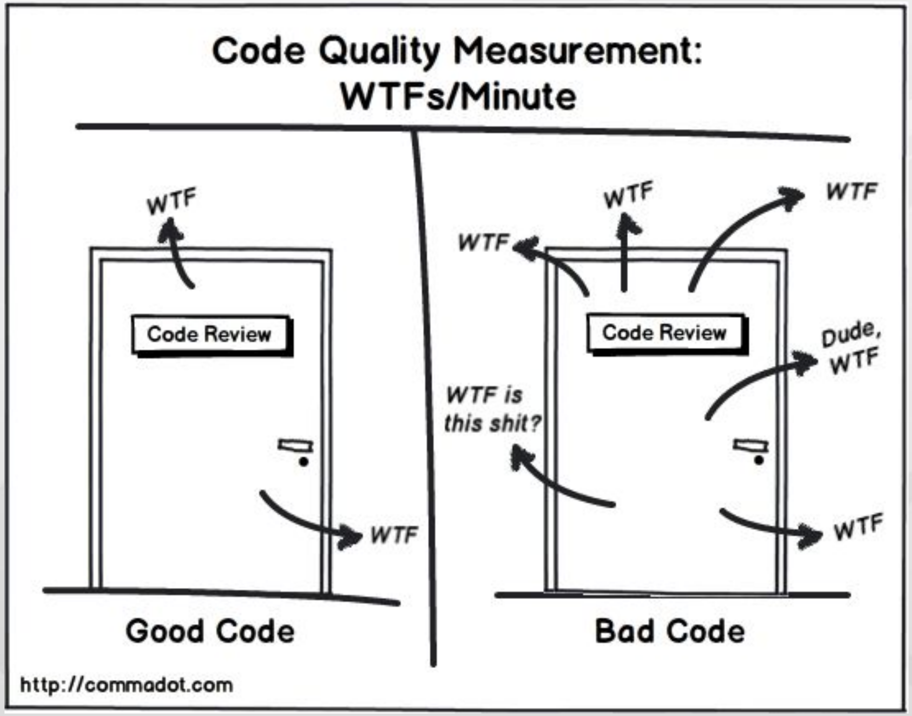
Making it easy to read actually makes it easier to write.

You are an @author who is responsible for communicating well with his readers.

Making it easy to read actually makes it easier to write.

"Clean code always looks like it was written by someone who cares."
/ Michael Feathers /

Clean Code principles



WHEN FACEPALM
World
IS NOT ENOUGH

Meaningful names

- reveal your intent

Meaningful names

- reveal your intent

```
/** Useful range constant. */
public static final int INCLUDE_NONE = 0;

/** Useful range constant. */
public static final int INCLUDE_FIRST = 1;

/** Useful range constant. */
public static final int INCLUDE_SECOND = 2;

/** Useful range constant. */
public static final int INCLUDE_BOTH = 3;
```

Meaningful names

- reveal your intent

```
/** Useful range constant. */
public static final int INCLUDE_NONE = 0;

/** Useful range constant. */
public static final int INCLUDE_FIRST = 1;

/** Useful range constant. */
public static final int INCLUDE_SECOND = 2;

/** Useful range constant. */
public static final int INCLUDE_BOTH = 3;
```

← No shit, Sherlock!

Meaningful names

- reveal your intent

```
public enum DateInterval {
     OPEN, CLOSED, OPEN_LEFT, OPEN_RIGHT
}
```

What about Enums?

Meaningful names

- reveal your intent

```
public List<int[]> getThem() {
     List<int[]> list1 = new ArrayList<int[]>();
          for (int[] x : theList)
                 if (x[0] == 4)
                          list1.add(x);
     return list1;
}
```

Meaningful names

- reveal your intent

```
public List<int[]> getThem() {
     List<int[]> list1 = new ArrayList<int[]>();
          for (int[] x : theList)
                 if (x[0] == 4)
                          list1.add(x);
     return list1;
}
```

WTF?!

Meaningful names

- reveal your intent

```
public List<int[]> getFlaggedCells() {
     List<int[]> flaggedCells = new ArrayList<int[]>();
          for (int[] cell : gameBoard)
                 if (cell[STATUS_VALUE] == FLAGGED)
                          flaggedCells.add(cell);
return flaggedCells;
}
```

Minesweeper

Meaningful names

- reveal your intent

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

Minesweeper

Meaningful names

- reveal your intent
- avoid disinformation

accountList -> accounts

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context

```
private void printGuessStatistics(char candidate, int count) {
      String number;
      String verb;
      String pluralModifier;

      if (count == 0) {
            number = "no";
            verb = "are";
            pluralModifier = "s";
      } else if (count == 1) {
            number = "1";
            verb = "is";
            pluralModifier = "";
      } else {
            number = Integer.toString(count);
            verb = "are";
            pluralModifier = "s";
      }

      String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier
      );
      print(guessMessage);
}
```

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context

```
public class GuessStatisticsMessage {
     private String number;
     private String verb;
     private String pluralModifier;

     public String make(char candidate, int count) {
          createPluralDependentMessageParts(count);
          return String.format("There %s %s %s%s",
               verb, number, candidate, pluralModifier );
     }

     private void createPluralDependentMessageParts(int count) {
          if (count == 0) {
               thereAreNoLetters();
          } else if (count == 1) {
               thereIsOneLetter();
          } else {
               thereAreManyLetters(count);
          }
     }

     private void thereAreManyLetters(int count) { .. }
     private void thereIsOneLetter() { .. }
     private void thereAreNoLetters() { .. }
}
```

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech

```
if (employee.isLate())
     employee.reprimand();
```

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech

```
getMockMvc().perform(mhsrb).andExpect(MockMvcResultMatchers.status().isBadRequest());

getMockMvc().perform(deleteRequest).andExpect(status().isBadRequest());
```

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech

```
public boolean set(String name, String value);

if (set("username","UserName")) {...}
```

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech

```
public void set(String name, String value) throw NotSetException {...}
public boolean isSet(String name) {...}

set("username","UserName");
if(isSet("username")) {...}
```

Rather throw exception if not succeeded, and check the state later.

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names

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

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names

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

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping

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

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping

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

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping
- avoid encodings, prefixes and abrs

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping
- avoid encodings, prefixes and abbreviations

```
IAccounts
oAccount
m_variable
```

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping
- avoid encodings, prefixes and abbreviations
- use static factory methods when constructors are overloaded

```
    Complex fulcrumPoint = new Complex(23.0);
```

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping
- avoid encodings, prefixes and abbreviations
- use static factory methods when constructors are overloaded

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping
- avoid encodings, prefixes and abbreviations
- use static factory methods when constructors are overloaded
- beware of using names which vary in small ways

```
XYZControllerForEfficientHandlingOfStrings     XYZControllerForEfficientStorageOfStrings
```

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping
- avoid encodings, prefixes and abbreviations
- use static factory methods when constructors are overloaded
- beware of using names which vary in small ways
- follow the Scope Rule

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping
- avoid encodings, prefixes and abbreviations
- use static factory methods when constructors are overloaded
- beware of using names which vary in small ways
- follow the Scope Rule

```
for(TestResult tr : configIssues) {
    Element element = createElement(d, tr);
}
```

What is d?

Meaningful names

- reveal your intent
- avoid disinformation
- add meaningful context
- parts of speech
- pronounceable names
- avoid mental mapping
- avoid encodings, prefixes and abbreviations
- use static factory methods when constructors are overloaded
- beware of using names which vary in small ways
- follow the Scope Rule

```
for(TestResult tr : configIssues) {
    Element element = createElement(document, tr);
}
```

Functions

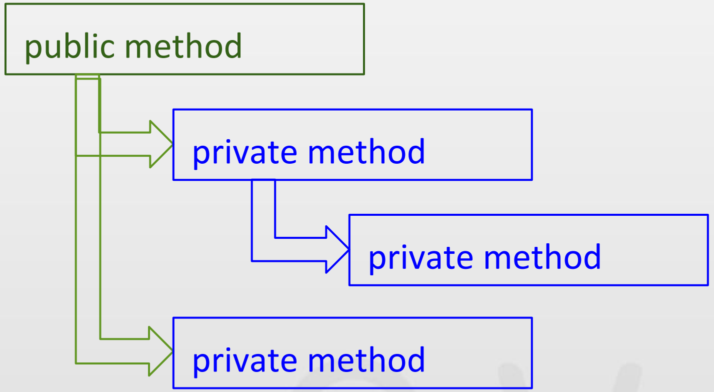
- Small!

Functions

- Small! 4 lines

Functions

- Small! 4 lines
- follow the Step Down Rule



Single Level of Abstraction

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names

Ward's principle:

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

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)

```
includeSetupPageInto(newPageContent) -> includeSetupPage()
```

A detail revealed

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)

```
includeSetupPageInto(newPageContent) -> includeSetupPage()
```

A detail revealed

```
assertEquals(message, expected, actual)
```

```
How many times have you read the
"message" and thought it was the
"expected"?
```

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements

Functions

```
public Money calculatePay(Employee e) throws InvalidEmployeeType {
     switch (e.type) {
          case COMMISIONED:
               return calculateCommisionedPay(e);
          case HOURLY:
               return calculateHourlyPay(e);
          case SALARIED:
               return calculateSalariedPay(e);
          default:
               throw new InvalidEmployeeType(e.type);
     }
}
```

Functions

```
public abstract class Employee {
     public abstract Money calculatePay();
}


public interface EmployeeFactory {
     public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}

...
```

Functions

```
...
public class EmployeeFactoryImpl implements EmployeeFactory {
     public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
          switch (r.type) {
               case COMMISIONED:
                    return new CommissionedEmployee(r);
               case HOURLY:
                    return new HourlyEmployee(r);
               case SALARIED:
                    return new SalariedEmployee(r);
               default:
                    throw new InvalidEmployeeType(r.type);
          }
     }
}
```

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)

```
public void open(File f, FileCommand c) {
     f.open();
     c.process(f);
     f.close();
}
```

This will not depend on a previous state of the system.

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation

```
User u = authorizer.login(username, password);
if (u != null) {
  ..
}
```

Authorizer should know the result of login.

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation

```
authorizer.login(username, password);
if (authorizer.isLoggedIn()) {
  ..
}
```

Tell don’t ask!

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation
- prefer (runtime) exceptions instead of returning error codes

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation
- prefer (runtime) exceptions instead of returning error codes

```
if (deletePage(page) == E_OK) {
  ...
}
```

Yikes!! Error processing could be separated from the happy path when you throw an exception.

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation
- prefer (runtime) exceptions instead of returning error codes
- extract try-catch blocks

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation
- prefer (runtime) exceptions instead of returning error codes
- extract try-catch blocks
- never return null

Unless you are a NullPointerException pervert!

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation
- prefer (runtime) exceptions instead of returning error codes
- extract try-catch blocks
- never return null
- avoid early exiting loops

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation
- prefer (runtime) exceptions instead of returning error codes
- extract try-catch blocks
- never return null
- avoid early exiting loops
- DRY (Don’t Repeat Yourself)

Forget copy-paste programming.

Functions

- Small! 4 lines
- follow the Step Down Rule
- use descriptive names
- minimize arguments (use at most 3, no booleans or nulls)
- avoid Switch statements
- avoid side effects (use functional programming)
- use Command Query Separation
- prefer (runtime) exceptions instead of returning error codes
- extract try-catch blocks
- never return null
- avoid early exiting loops
- DRY (Don’t Repeat Yourself)
- KISS (Keep It Simple, Stupid)

Comments

“Don’t comment bad code—rewrite it.” —Brian W. Kernighan and P. J. Plaugher

Comments

“Don’t comment bad code—rewrite it.”

- comments are failures of expressing yourself with code

Avoid them! Ignore them! Remove them!

Comments

“Don’t comment bad code—rewrite it.”

- comments are failures of expressing yourself with code
- inaccurate comments are far worse than no comments at all

Comments

“Don’t comment bad code—rewrite it.”

- comments are failures of expressing yourself with code
- inaccurate comments are far worse than no comments at all
- Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.

Comments

“Don’t comment bad code—rewrite it.”

- comments are failures of expressing yourself with code
- inaccurate comments are far worse than no comments at all
- Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.
- use functions or a variables instead of comments

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

Comments

“Don’t comment bad code—rewrite it.”

- comments are failures of expressing yourself with code
- inaccurate comments are far worse than no comments at all
- Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.
- use functions or a variables instead of comments

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

Comments

“Don’t comment bad code—rewrite it.”

- comments are failures of expressing yourself with code
- inaccurate comments are far worse than no comments at all
- Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.
- use functions or a variables instead of comments
- remove commented-out code

Comments

“Don’t comment bad code—rewrite it.”

- comments are failures of expressing yourself with code
- inaccurate comments are far worse than no comments at all
- Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.
- use functions or a variables instead of comments
- remove commented-out code
- no position markers, no html comments

Comments

"Don't comment bad code—rewrite it."

- comments are failures of expressing yourself with code
- inaccurate comments are far worse than no comments at all
- Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess.
- use functions or a variables instead of comments
- remove commented-out code
- no position markers, no html comments
- no TODO comments

Comments

“Don’t comment bad code—rewrite it.”

- comments are failures of expressing yourself with code
- inaccurate comments are far worse than no comments at all
- Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.
- use functions or a variables instead of comments
- remove commented-out code
- no position markers, no html comments
- no TODO comments
- good comments: legal comments
  informative comments (regexp)
  clarification, warning of consequences
  javadoc

Classes

- Small!

Classes

- Small! 1 responsability

Classes

- Small! 1 responsability
- high cohesion, low coupling

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter

```
o.getX().getY().doSomething() -> o.doSomething()
```

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Train Wrecks!

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter

Train Wrecks!

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
...
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
...
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

Train Wrecks!

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID

Single responsibility principle

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID

Single responsibility principle

Open-closed principle

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID

Single responsibility principle

Open-closed principle

Liskov substitution principle

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID

Single responsibility principle

Open-closed principle

Liskov substitution principle

Type-case is a violation of this principle:

```
if (shape instanceof Shape) ...
else if (shape instanceof Square) ...
else if (shape instanceof Rectangle) ...
```

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID

Single responsibility principle

Open-closed principle

Liskov substitution principle

Interface segregation principle

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID

Single responsibility principle

Open-closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID
- favor composition over inheritance

Classes

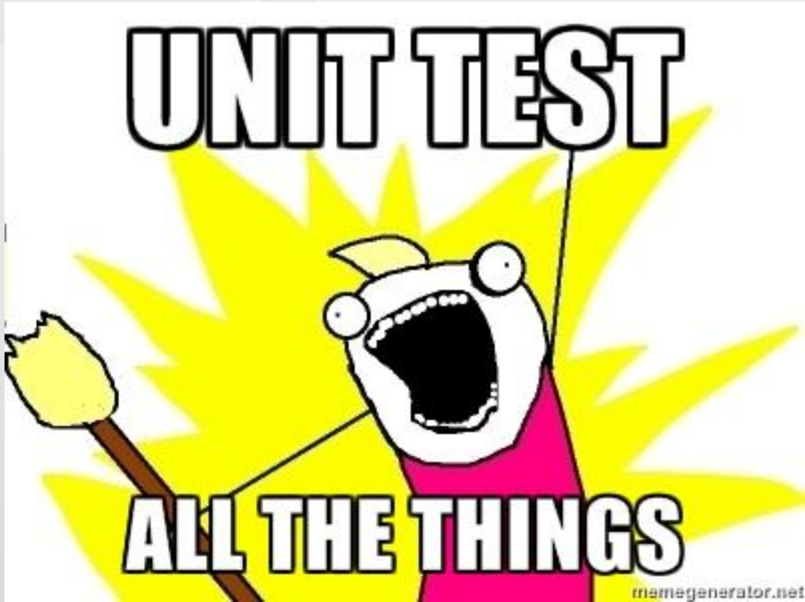
- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID
- favor composition over inheritance
- YAGNI (You Ain’t Gonna Need It)

Classes

- Small! 1 responsability
- high cohesion, low coupling
- don’t talk to strangers - Law of Demeter
- make it SOLID
- favor composition over inheritance
- YAGNI (You Ain’t Gonna Need It)
- beware of optimizations

Unit tests

- as important as production code (documentation)



Unit tests

- as important as production code (documentation)
- use domain-specific testing language

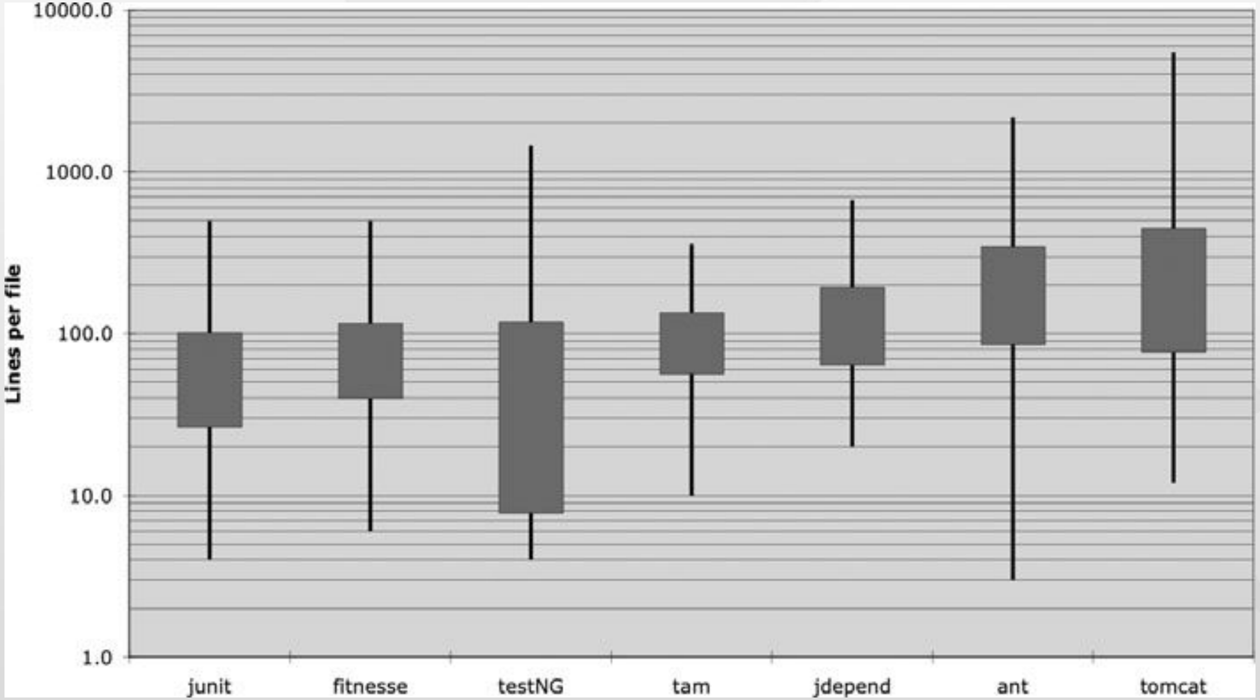
```
public void testGetDataAsXml() throws Exception {
     makePageWithContent("TestPageOne", "test page");

     submitRequest("TestPageOne", "type:data");

     assertResponseIsXML();
     assertResponseContains("test page", "<Test");
}
```

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design



Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design

The **3** laws of TDD:

You may not write production code until you have written a failing unit test.

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design

The **3** laws of TDD:

You may not write production code until you have written a failing unit test.

You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design

The **3** laws of TDD:

You may not write production code until you have written a failing unit test.

You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

You may not write more production code than is sufficient to pass the currently failing test.

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design

The **3** laws of TDD:

You may not write production code until you have written a failing unit test.

You may not write more of a unit test than is sufficient to fail, and not compiling is failing.

You may not write more production code than is sufficient to pass the currently failing test.

+1: Refactor (Red-Green-Refactor)

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design
- follow the Boy Scout Rule

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design
- follow the Boy Scout Rule
- test FIRST

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design
- follow the Boy Scout Rule
- test FIRST

Fast

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design
- follow the Boy Scout Rule
- test FIRST

Fast

Independent

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design
- follow the Boy Scout Rule
- test FIRST

Fast

Independent

Repeatable

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design
- follow the Boy Scout Rule
- test FIRST

Fast

Independent

Repeatable

Self-validating

Unit tests

- as important as production code (documentation)
- use domain-specific testing language
- single concept per test
- TDD leads to better design
- follow the Boy Scout Rule
- test FIRST

Fast

Independent

Repeatable

Self-validating

Timely

Takeaways

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.”

—Ray Ozzie, CTO, Microsoft Corporation

Takeaways

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.”

- Clarity is king!

Takeaways

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.”

- Clarity is king!
- don’t save (your) time on namings

Takeaways

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.”

- Clarity is king!
- don’t save (your) time on namings
- read less, write more

Takeaways

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.”

- Clarity is king!
- don’t save (your) time on namings
- read less, write more
- No comments!

Takeaways

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.”

- Clarity is king!
- don’t save (your) time on namings
- read less, write more
- No comments!
- DRY, KISS, YAGNI, SOLID, TDD, FIRST

Takeaways

"Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test."

- Clarity is king!
- don't save (your) time on namings
- read less, write more
- No comments!
- DRY, KISS, YAGNI, SOLID, TDD, FIRST
- refactoring is not a nice-to-have option

Takeaways

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.”

- Clarity is king!
- don’t save (your) time on namings
- read less, write more
- No comments!
- DRY, KISS, YAGNI, SOLID, TDD, FIRST
- refactoring is not a nice-to-have option
- later = NEVER

Takeaways

"Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test."

- Clarity is king!
- don't save (your) time on namings
- read less, write more
- No comments!
- DRY, KISS, YAGNI, SOLID, TDD, FIRST
- refactoring is not a nice-to-have option
- later = NEVER
- always improve

Takeaways

“Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build, and test.”

- Clarity is king!
- don’t save (your) time on namings
- read less, write more
- No comments!
- DRY, KISS, YAGNI, SOLID, TDD, FIRST
- refactoring is not a nice-to-have option
- later = NEVER
- always improve
- Boy Scout Rule

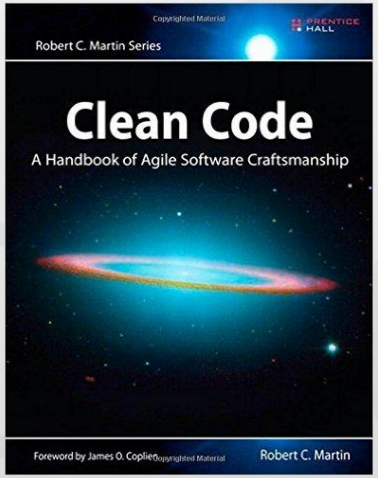
Code for humans, not machines

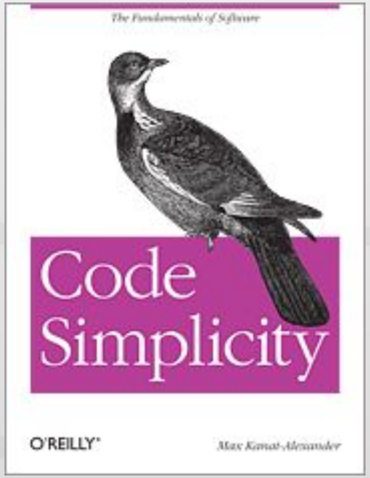
Think that the next person who reads your code is a chainsaw maniac.

If you don’t write clean code, you know your fate.

Resources

- Robert C. Martin: Clean code - A handbook of Agile Software Craftsmanship
- Max Kanat-Alexander: Code Simplicity
- [www.cleancoders.com]
- [www.clean-code-developer.hu] (.de)
- [Clean Code cheat sheet]
- [The essence of “Clean Code”]





evosoft

RUN A WINNING SYSTEM

Your powerful partner

Contact

Evosoft Hungary

Kaposvár utca 14-18

H-1117 Budapest

Web: www.evosoft.com

Counterpart

Zoltán Domahidi

O2.10 / Java Developer

Email: Zoltan.Domahidi@evosoft.com