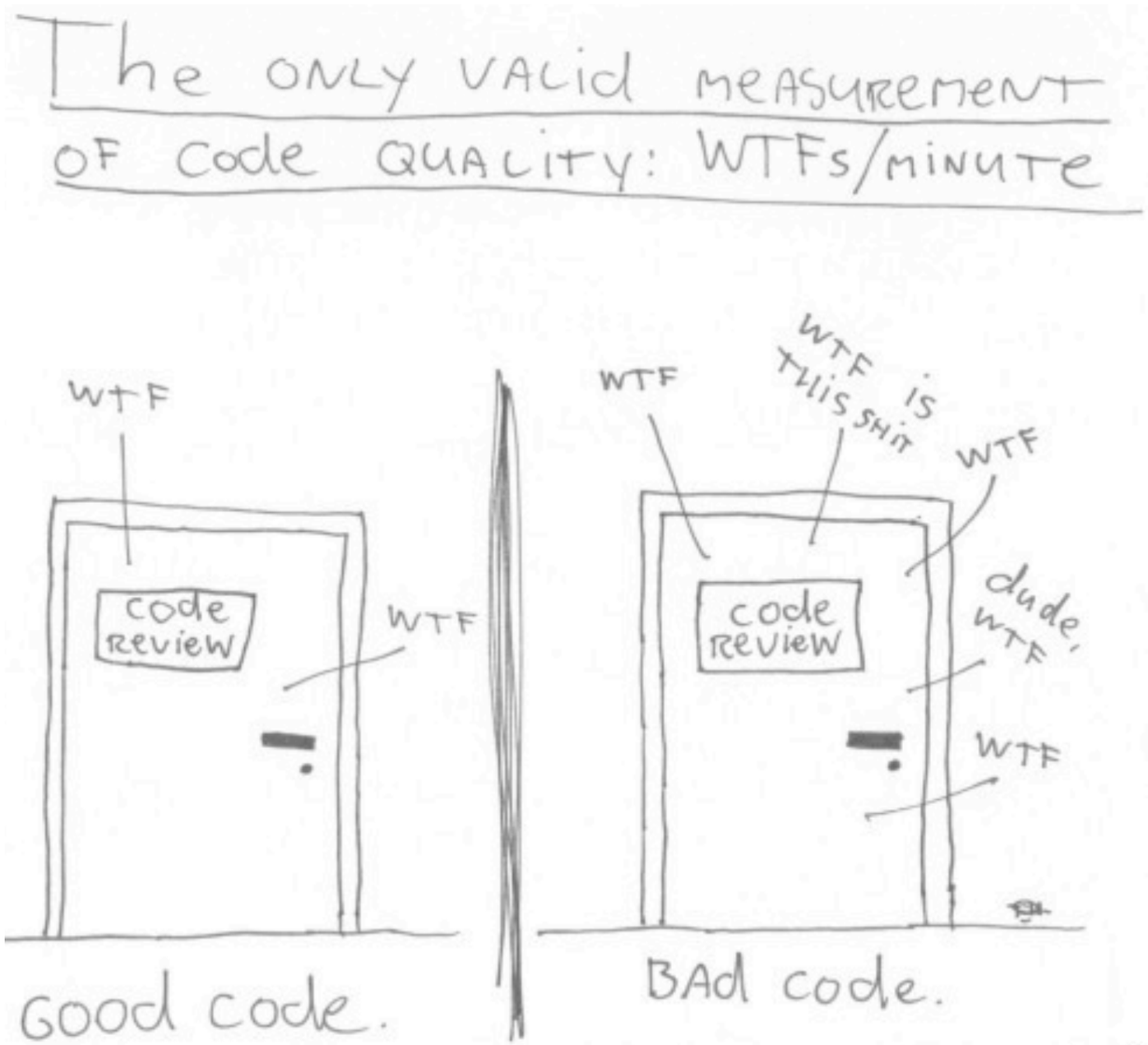
Clean Code

@mariosangiorgio

Why?


The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is THIS SHIT
WTF
code review
dude, WTF
WTF
Bad code.

Goals

Readable, maintainable and extendable code

Meaningful names

A simple example

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

A simple example

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

This code is quite simple but what does it do?

A simple example

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

This code is quite simple but what does it do?

Looking at it we can't tell what it is actually doing!

A simple example

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

Is this code any better?

A simple example

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

What about this?

A simple example

What we have done:

A simple example

What we have done:

used intention revealing names

`flaggedCells` rather than `list1`

A simple example

What we have done:

used intention revealing names

`flaggedCells`

rather than `list1`

replaced *magic numbers* with constants

`cell[STATUS_VALUE]`

rather than `x[0]`

A simple example

What we have done:

used intention revealing names	`flaggedCells` rather than `list1`
replaced *magic numbers* with constants	`cell[STATUS_VALUE]` rather than `x[0]`
created an appropriate abstract data type	`Cell cell` rather than `int[] cell`

Another example

```
int d;
```

What does it mean?
Days? Diameter? ...

Another example

```
int d;
```

What does it mean?
Days? Diameter? ...

```
int d; //elapsed time in days
```

Is this any better?

Another example

```
int d;
```

What does it mean?
Days? Diameter? ...

```
int d; //elapsed time in days
```

Is this any better?

```
int elapsedTimeInDays;
```

What about this?

Functions

Do one thing

```
public bool isEdible() {
  if (this.ExpirationDate > Date.Now &&
      this.ApprovedForConsumption == true &&
      this.InspectorId != null) {
    return true;
  } else {
    return false;
  }
}
```

How many things is the function doing?

Do one thing

```
public bool isEdible() {
  if (this.ExpirationDate > Date.Now &&
      this.ApprovedForConsumption == true &&
      this.InspectorId != null) {
    return true;
  } else {
    return false;
  }
}
```

1.check expiration
2.check approval
3.check inspection
4.answer the request

How many things is the function doing?

Do one thing

```
public bool isEdible() {
    return isFresh()    &&
                isApproved() &&
                isInspected();
}
```

Now the function is doing one thing!

Do one thing

```
public bool isEdible() {
    return isFresh()    &&
                isApproved() &&
                isInspected();
}
```

Now the function is doing one thing!

A change in the specifications turns
into a single change in the code!

Don’t mix levels of abstraction

```
public void doTheDomesticThings() {
    takeOutTheTrash();
    walkTheDog();
    for (Dish dish : dirtyDishStack) {
        sink.washDish(dish);
        teaTowel.dryDish(dish);
    }
}
```

```
public void doTheDomesticThings() {
    takeOutTheTrash();
    walkTheDog();
    doTheDishes();
}
```

Which one is easier to read and understand?

Separate commands and queries

Commands should **only** do something (One thing)

Queries should **only** answer something

```
public class Car{
    private boolean isOn

    public void turnOn(){
        isOn = true;
    }

    public boolean isOn(){
        return isOn;
    }
}
```

AVOID SIDE EFFECTS!

Use exceptions

```
public int foo(){
   ...
}
```

```
public void bar(){
   if(foo() == OK)
      ...
   else
      // error handling
}
```

Use exceptions

Errors have to be encoded

```
public int foo(){
   ...
}

public void bar(){
   if(foo() == OK)
      ...
   else
      // error handling
}
```

Use exceptions

```
public int foo(){
   ...
}

public void bar(){
   if(foo() == OK)
      ...
   else
      // error handling
}
```

Errors have to be encoded

Checks (when performed) require a lot of code

Use exceptions

```
public int foo(){
   ...
}

public void bar(){
   if(foo() == OK)
      ...
   else
      // error handling
}
```

Errors have to be encoded

Checks (when performed) require a lot of code

It's harder to extend such programs

Use exceptions

```
public void foo()
      throws FooException{
   ...
}

public void bar(){
   try{
      foo();
      ...
   } catch(FooException){
      // error handling
   }
}
```

Use exceptions

```
public void foo()
      throws FooException{
   ...
}
```

No need to mix return values and control values

```
public void bar(){
   try{
      foo();
      ...
   } catch(FooException){
      // error handling
   }
}
```

Use exceptions

```
public void foo()
      throws FooException{
   ...
}
```

```
public void bar(){
   try{
      foo();
      ...
   } catch(FooException){
      // error handling
   }
}
```

No need to mix return values and control values

Cleaner syntax

Use exceptions

```
public void foo()
      throws FooException{
   ...
}

public void bar(){
   try{
      foo();
      ...
   } catch(FooException){
      // error handling
   }
}
```

No need to mix return values and control values

Cleaner syntax

Easier to extend

Don’t Repeat Yourself

```
public void bar(){
  foo(“A”);
  foo(“B”);
  foo(“C”);
}
```

```
public void bar(){
  String [] elements = {“A”, “B”, “C”};

  for(String element : elements){
    foo(element);
  }

}
```

DO NOT EVER COPY AND PASTE CODE

Don’t Repeat Yourself

```
public void bar(){
  foo(“A”);
  foo(“B”);
  foo(“C”);
}
```

```
public void bar(){
  String [] elements = {“A”, “B”, “C”};

  for(String element : elements){
    foo(element);
  }

}
```

Logic to handle the elements
it’s written once for all

DO NOT EVER COPY AND PASTE CODE

Comments

Explain yourself in the code

Which one is clearer?

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

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

Comments

GOOD	BAD
API Documentation	Redundant
Explanation of intent	Obsolete
Clarification	Code commented-out

Other code smells

What we don’t want to see in your code

The bloaters

Something in your code grow too large

Long methods and large classes

Primitive obsession and too much parameters

Single responsibility principle violated

It is a symptom of bad design

Primitive obsession

```
public Class Car{
    private int red, green, blue;

    public void paint(int red, int green, int blue){
        this.red   = red;
        this.green = green;
        this.blue  = blue;
    }
}
```

```
public Class Car{
    private Color color;

    public void paint(Color color){
        this.color = color;
    }
}
```

The OO abusers

Object orientation is not fully exploited

Switch statements on objects

Refused bequest

Alternative classes with different interfaces

It is better to use polymorphism

Poor class hierarchy design

Switch vs polymorphism

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

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

Refused bequest

Subclass doesn't use superclass methods and attributes

```
public abstract class Employee{
    private int quota;
    public  int getQuota();
    ...
}

public class Salesman extends Employee{ ... }
```

```
public class Engineer extends Employee{
    ...
    public int getQuota(){
        throw new NotSupportedException();
    }
}
```

Engineer does not use quota. It should be pushed down to Salesman

The change preventers

Something is making hard to change the code

Divergent change	A class has to be changed in several parts
Shotgun surgery	A single change requires changes in several classes

The dispensables

The code contains something unnecessary

A class is not doing enough

Class not providing logic

Unused or redundant code

It isn’t something useful

The couplers

Some classes are too tightly coupled

Feature Envy

Misplaced responsibility

Inappropriate Intimacy

Classes should know as little as possible about each other

Message Chains

Too complex data access

Feature Envy

```
public class Customer{
    private PhoneNumber mobilePhone;

    ...

    public String getMobilePhoneNumber(){
        return  “(” +
                mobilePhone.getAreaCode() + “)” +
                mobilePhone.getPrefix() + “-” +
                mobilePhone.getNumber();
    }
}
```

Feature Envy

```
public class Customer{
    private PhoneNumber mobilePhone;

    ...

    public String getMobilePhoneNumber(){
        return  “(” +
                mobilePhone.getAreaCode() + “)” +
                mobilePhone.getPrefix() + “-” +
                mobilePhone.getNumber();
    }
}
```

```
    public String getMobilePhoneNumber(){
        return  mobilePhone.toFormattedString();
    }
```

Message chains

```
a.getB().getC().getD().getTheNeededData()
```

```
a.getTheNeededData()
```

Law of Demeter: Each unit should only talk with friends