Clean Code

Thomas Sundberg

Consultant, Developer

Stockholm, Sweden

Sigma Solutions AB

thomas.sundberg@sigma.se

SIGMA

Clean Code

Goal and expected outcome

- An understanding of why readable and maintainable code is the most important outcome of your profession
- Know that software development is a craft that requires craftsmanship
- Understand that technical excellence is important

SIGMA

Clean Code Definition

No clear definition

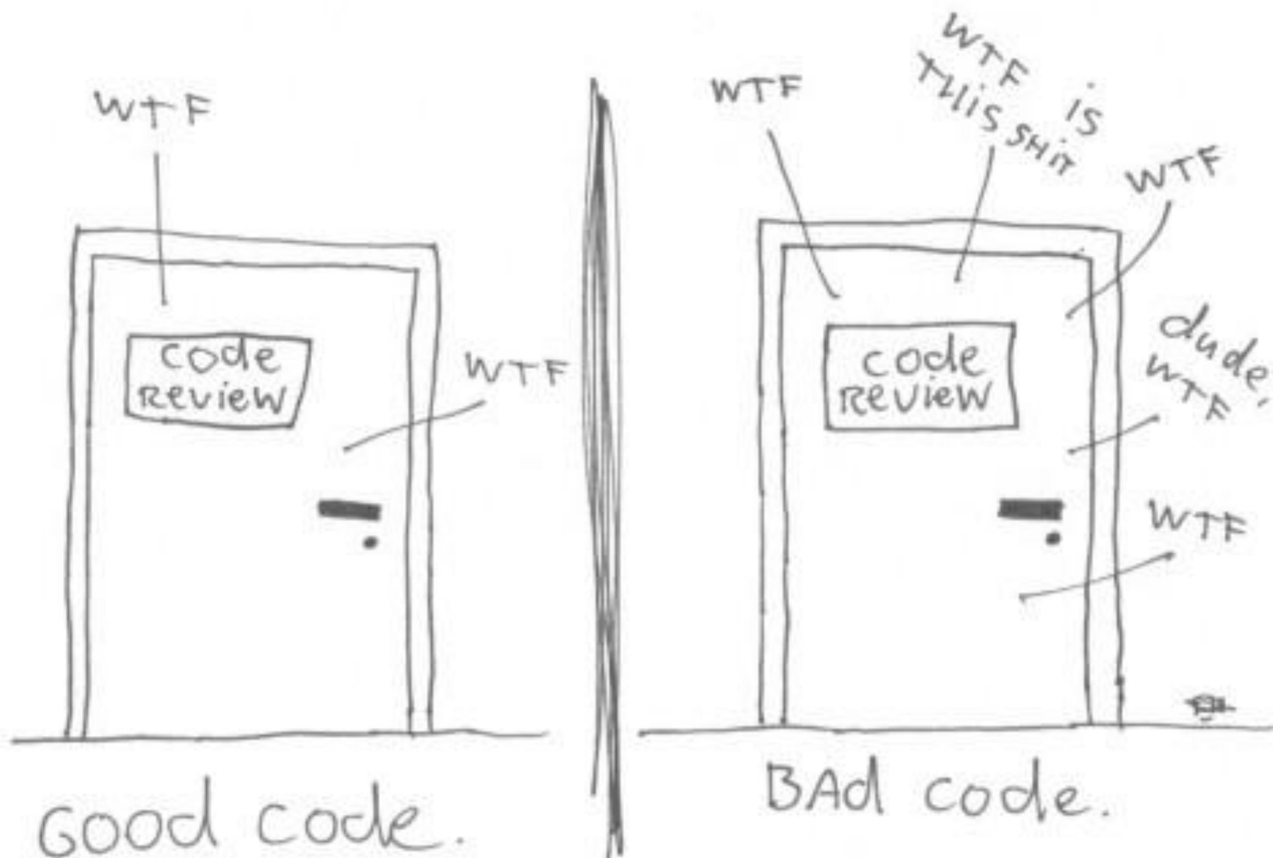
The result of experience

Different modern schools of Jujutsu

- Danzan Ryu
- German Ju-Jutsu
- Goshin Jujitsu
- Hakko Ryu
- Hakko Denshin Ryu
- Hokutoryu jujutsu
- Jukido Jujitsu
- Ketsugo Jujutsu
- Kumite-ryu Jujutsu
- Miyama Ryu
- Purple Dragon Don Jitsu Ryu System
- Sanuces Ryu
- Shorinji Kan Jiu Jitsu (The Jitsu Foundation)

SIGMA

The only valid measurement of code quality: WTFs/minute



(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

SIGMA

Clean Code

There will be code

Bad code

Clean Code

- We are authors
  - Our code must
    - Execute
    - Be correct
    - Be easy to read

Clean Code

The boy scout rule:

Always leave the camp ground cleaner than you found it

Translates to:

Always leave the code in a better shape than you found it

Meaningful Names

Use intention revealing names

Why is it hard to tell what this code is doing?

?

```
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

The function can be re-written as

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

SIGMA

Meaningful Names

It is possible to simplify it even more

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

SIGMA

Meaningful Names

Make meaningful distinctions

Where will the chars end up?

```
public void copyChars(char a1[], char a2[]) {
    ...
}
```

Meaningful Names

Make meaningful distinctions

The implementation:

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

Meaningful Names

Make meaningful distinctions

Rename the arguments source and target:

```
public void copyChars(char target[], char source[])
```

SIGMA

Meaningful Names

Pronounceable names

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

```
class Customer {
    private Date generatedTimestamp;
    private Date modificationTimestamp;
    private final String recordId = "102";
}
```

Meaningful Names

Searchable names

- We often search for names

Meaningful Names

Encodings

Hungarian notation

```
private String descriptionString;
private String description;
```

Member prefix

```
m_description
_description
```

Interfaces and Implementations

```
IShapeFactory
ShapeFactory
ShapeFactoryImpl
```

SIGMA

Meaningful Names

- Mental mapping
  - Domain problem names
  - Solution problem names

Meaningful Names

Class names

- Nouns
- Customer, Account, Chair

Method names

- Verb
- save, deleteAccount

Don't be funny

SIGMA

Meaningful Names

One word per concept

- fetch
- retrieve
- get

Choose one and stick to it

Meaningful Names

Add meaningful context

```
String firstName;
String lastName;
String street;
String state;
```

SIGMA

Meaningful Names

Add meaningful context

```
String state;
```

Somewhere else and alone, who can tell what it means?

Meaningful Names

Add meaningful context

Solution:

```
String addrFirstName;
String addrLastName;
String addrStreet;
String addrState;
```

SIGMA

Meaningful Names

Add meaningful context

Better:

```
class Address {
    private String firstName;
    private String lastName;
    private String street;
    private String state;
}
```

SIGMA

Functions

Small!

A small function is easy to overview

A function should always fit the screen

Aim for a vertical size of one digit

Functions

- Do one thing
  - Single Responsibility Principle – SRP
    - There should only be one reason for a function to change

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

SIGMA

Functions

One level of abstraction per function

```
public String extractTicketNumber(String address) {
    String bookingConfirmation = fetchPageContent(address);
    StringTokenizer stringTokenizer = new StringTokenizer(bookingConfirmation);
    String ticketNumber = "";
    while (stringTokenizer.hasMoreElements()) {
        String token = stringTokenizer.nextToken();
        if (token.equalsIgnoreCase("Number") && stringTokenizer.hasMoreTokens()){
            return stringTokenizer.nextToken();
        }
    }
    return ticketNumber;
}
```

Functions

One level of abstraction per function

Refactor:

```
public String extractTicketNumber(String address) {
  String bookingConfirmation = fetchPageContent(address);
  String ticketNumber =
                scrapeTicketNumber(bookingConfirmation);
  return ticketNumber;
}
```

Functions

One level of abstraction per function

Refactor:

```
private String scrapeTicketNumber(String bookingConfirmation) {
    StringTokenizer stringTokenizer = new StringTokenizer(bookingConfirmation);
    String ticketNumber = "";
    while (stringTokenizer.hasMoreElements()) {
        String token = stringTokenizer.nextToken();
        if (token.equalsIgnoreCase("Number") && stringTokenizer.hasMoreTokens())
    {
            return stringTokenizer.nextToken();
        }
    }
    return ticketNumber;
}
```

SIGMA

Functions

Switch statements

- Long
- Does many things, typically N
- Can they be avoided?

Functions

Switch statements

```
public Money calculatePay(Employee employee) {
    switch (employee.type()) {
        case Employee.COMMISSIONED:
            return calculateCommissionedPay(employee);
        case Employee.HOURLY:
            return calculateHourlyPay(employee);
        case Employee.SALARY:
            return calculateSalaryPay(employee);
        default:
            throw new
  InvalidEmployeeType(employee.type());
    }
}
```

SIGMA

Functions

Switch statements

```
public boolean isPayDay();
public void deliverPay(Money pay);
```

SIGMA

Functions

Switch statements

Polymorphic objects

```
public abstract class Employee {
    public abstract boolean isPayDay();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}
```

Functions

Switch statements

Polymorphic objects

```
public interface EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord employeeRecord);
}
```

Polymorphic objects

```
public Employee makeEmployee(EmployeeRecord employeeRecord) {
   switch (employeeRecord.type()) {
      case EmployeeRecord.COMMISSIONED:
         return new CommissionedEmployee(employeeRecord);
      case EmployeeRecord.HOURLY:
         return new HourlyEmployee(employeeRecord);
      case EmployeeRecord.SALARY:
         return new SalariedEmployee(employeeRecord);
      default:
        throw new invalidEmployeeType(employeeRecord.type());
       }
   }
}
```

Functions

- Use descriptive names
  - Long names
  - Short names

Functions

Function arguments

- Zero arguments – Niladic

- One argument – Monadic
  - Question
  - Operation
  - Flag

SIGMA

Functions

Function arguments

Two arguments - Dyadic functions

```
Point p = new Point(1, 2);

assertEquals(x, y);

assertEquals(expected, actual);

assertExpectedEqualsActual(expected, actual);
```

SIGMA

Functions

Function arguments

Three arguments – Triads

```
assertEquals(foo, bar, baz);
```

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

SIGMA

Functions

Function arguments

Argument objects

```
class Address {
    private String firstName;
    private String lastName;
    private String street;
    private String state;
}
```

SIGMA

Functions

Function arguments

Verbs and keywords

```
verb(noun);

write(name);

writeField(name);

assertExpectedEqualsActual(expected, actual);
```

Functions

Have no side-effects

Never have unexpected side effects

```
public boolean checkPassword(String userName, String password){
    User user = UserGatweway.findByname(userName);
    if (user != User.NULL) {
        String codedPhrase = user.getPharseEncodedByPassword();
        String phrase = cryptographer.decrypt(codedPhrase,
  password);
        if ("Valid password".equals(phrase)) {
            Session.initialize();
            return true;
        }
    }
     return false;
}
```

SIGMA

Functions

Have no side-effects

Never have unexpected side effects

```
public boolean checkPasswordAndInitializeSession(String userName,
   String password)
```

Functions

Have no side-effects

Output arguments

```
public void moveOneStepLeft(Point currentPosition) {
    currentPosition.x--;
}
```

```
public Point moveOneStepLeft(Point currentPosition) {
    currentPosition.x--;
    return currentPosition;
}
```

SIGMA

Functions

Have no side-effects

Output arguments

```
private Point currentPosition;

public void moveOneStepLeft() {
    currentPosition.x--;
}
```

Functions

Command query separation

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

```
if (set("userName", "Thomas"))
```

Functions

- Don't repeat yourself – DRY
  - Never duplicate any code!

SIGMA

Functions

Structured programming

- Dijkstras rules
  - One entry point
  - One exit point
  - No break
  - No continue

Ok to break Dijkstras rules, our functions are small enough so we can overview them

SIGMA

Functions

How do you write functions like this?

Test Driven Development – TDD

- Start with a test
- Write some production code until the test is satisfied
- Refactor your code
- A new test
- More production code
- Refactor your code

SIGMA

Comments

Comments do not make up for bad code

- Bad code should not be commented,
  it should be rewritten

Explain yourself in code

Comments

Good Comments

Intention:

```
// This is our best attempt to achieve a
// race condition when testing with many threads
```

Todo may be reasonable to leave if we expect to fix them soon

```
// TODO We will not need this when we have fixed bla bla
```

Javadoc in public API

SIGMA

Comments

Bad Comments

Redundant comments

```
// we will add a [illegible] figure to the list of figures
figures.add(new Figure());
```

Misleading comments

Comments that haven't been updated or are just plain wrong

Comments

Bad Comments

Mandated comments on every function

```
/**
 * Default constructor
 */
public Figure() {
}
```

Do not use a comment when you can use a function or a variable

Comments

Bad Comments

Closing comments

```
private void badEndingComment() {
    int index = 0;
    try {
        while (index < 50) {
            index--;
        } // while
    } // try
    catch (Exception e) {
        e.printStackTrace();
    } // catch
} // badEndingComment
```

SIGMA

Comments

Comment your code when appropriate

Comment wisely and always ask yourself:

- “How can I rewrite this so I don’t have to write a comment?”

SIGMA

Error Handling

- Exceptions
  - Exceptions rather than return codes
  - Unchecked exceptions
  - Provide context with exceptions
  - Define exception in terms of a caller's needs

Error Handling

Null

- Don't return null

- Don't pass null

Clean Code

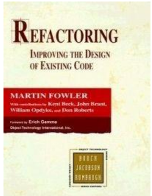
The boy scout rule:

Always leave the code in a better shape than you found it

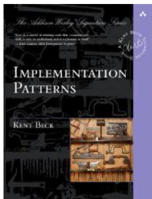
Kaizen – Continuous Improvement

Resources

Clean Code – Robert C. Martin,
ISBN 0132350882



Refactoring – Martin Fowler,
ISBN 0201485672



Implementation Pattern – Kent Beck
ISBN 0321413091

Blogg - http://thomassundberg.wordpress.com
thomas.sundberg@sigma.se

SIGMA