Clean Code

Benedikus H Tjuatja
David Setyanugraha
Fakhri
Lutvianes Advendianto

gdp
LABS

gdp LABS

- Help SISTER COMPANIES
- Incubate STARTUPS
- Constantly LEARNING

Founded in
JUNE 2012

123 PEOPLE & GROWING

SOFTWARE ENGINEER

PRODUCT MANAGER

UI/UX DESIGNER

PRODUCT CENTRIC

SYSTEM ENGINEER

DATA SCIENTIST

DATA ANALYST

WON 99 INTERNATIONAL & NATIONAL OUT OF 255 COMPETITIONS

CONTINUOUS LEARNING

SHARING

● JAKARTA

● BANDUNG

● YOGYAKARTA

OPENING SOON!

● SURABAYA & BALI

Overview

- Issues
- Solution
- Conclusion

Issues

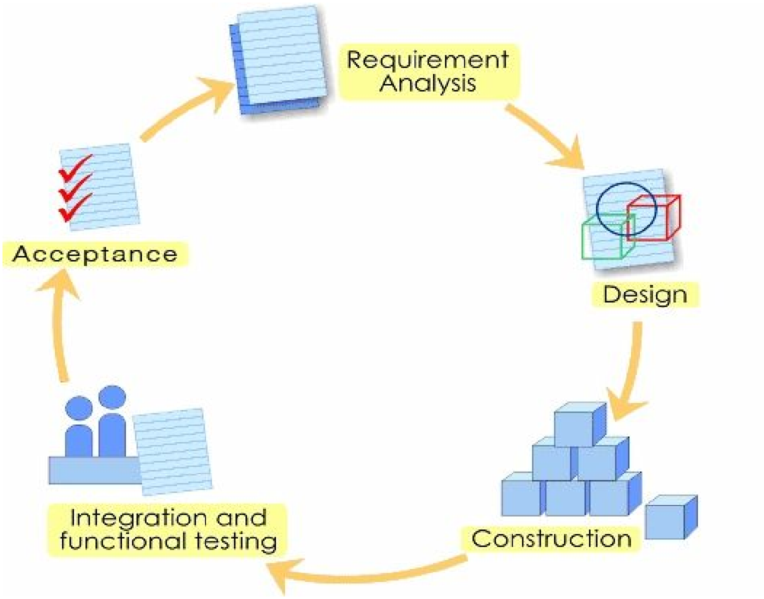
- Bad Code
- Great vs Bad Company
- Who is Responsible?

Bad Code

- Technical debt:
  the eventual consequences of poor software architecture and software development within a code base.
  - Rushed the product to the market
  - Had made a huge mess in the code
  - Added more features
  - The code got worse and worse

Bad Code

- LeBlanc's Law:
  Later equals never
  - They never go back to fix the bad code
- Sink projects, careers and companies
- Example: *netscape* and *myspace*

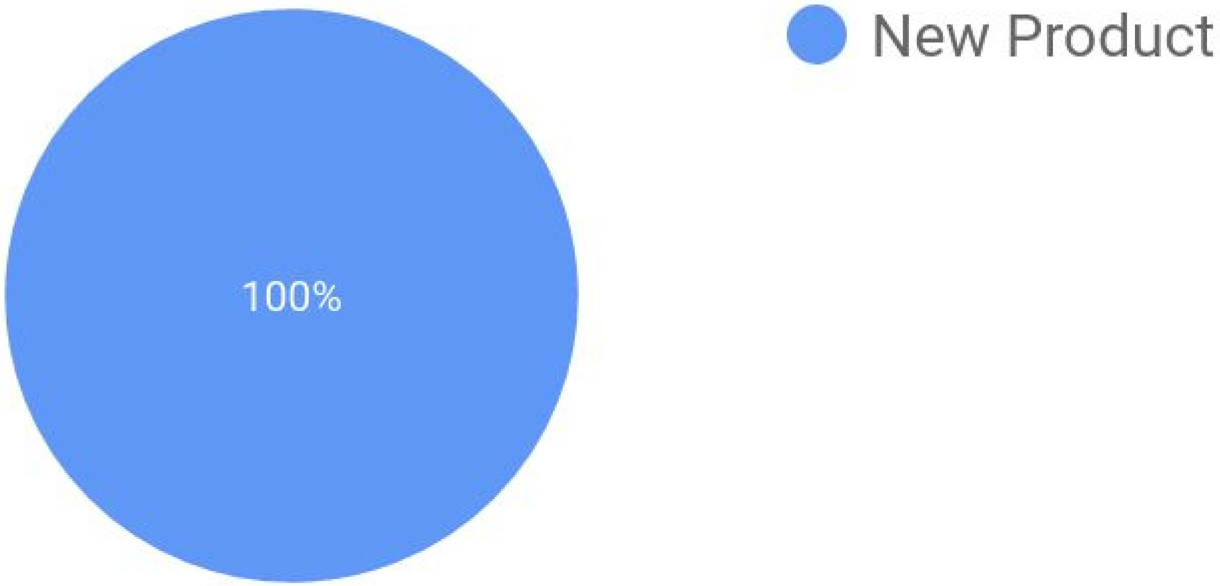


Death Spiral:

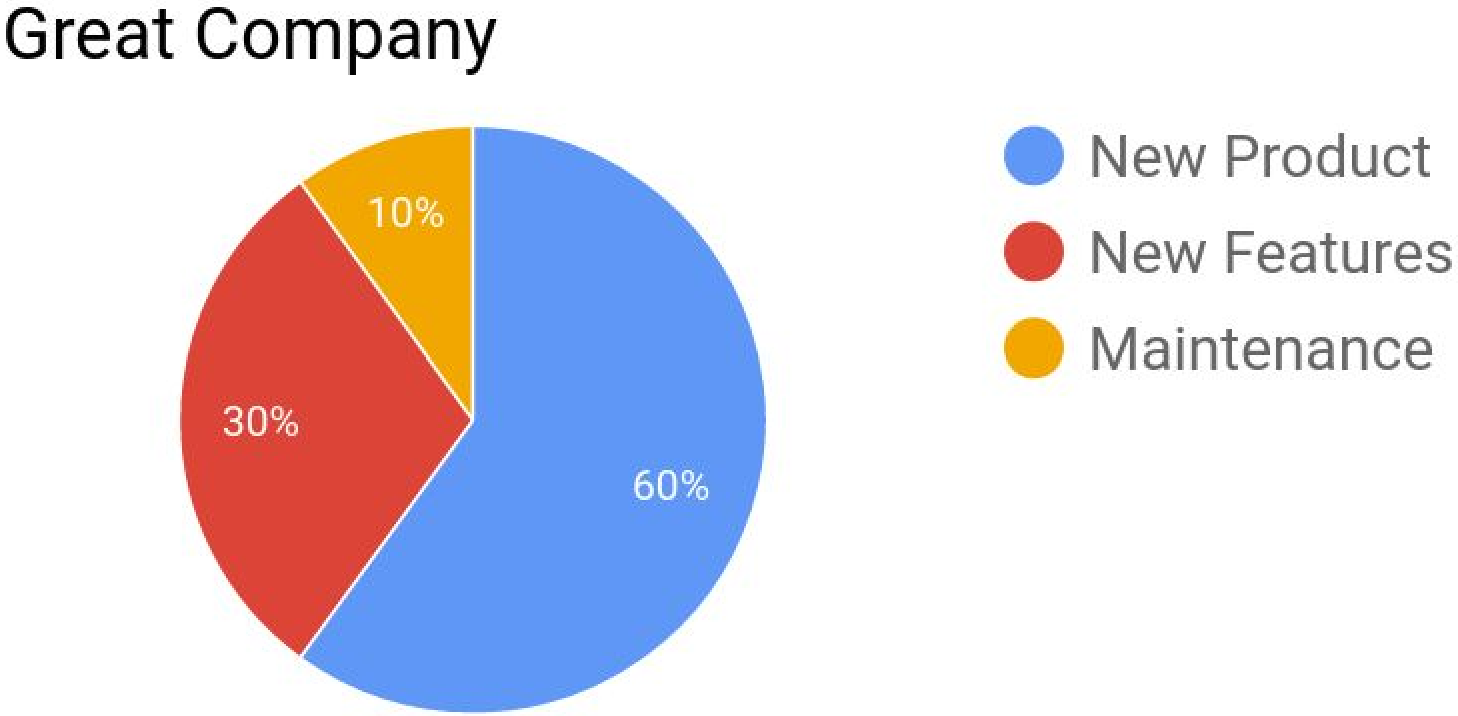
The downward, corkscrew-motion of a disabled aircraft which is unrecoverably headed for a crash

Great Company VS Bad Company

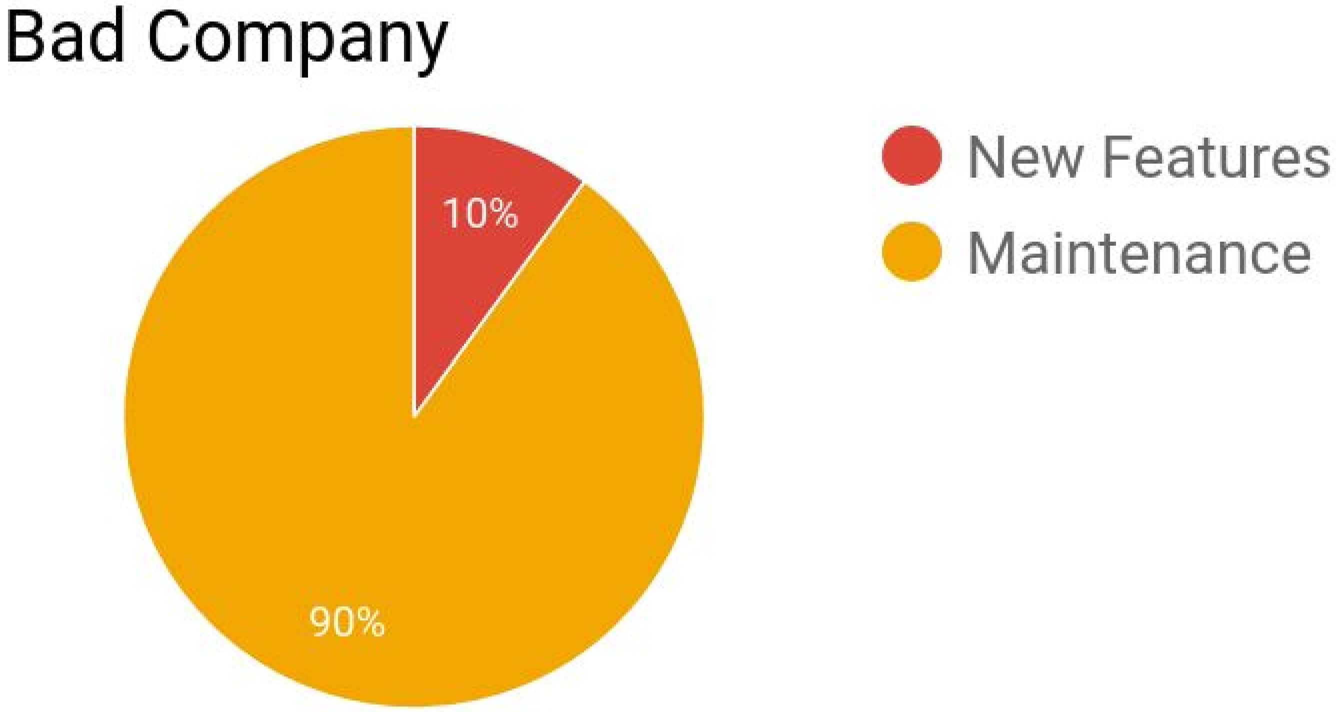
Beginning



Great Company VS Bad Company



Great Company VS Bad Company



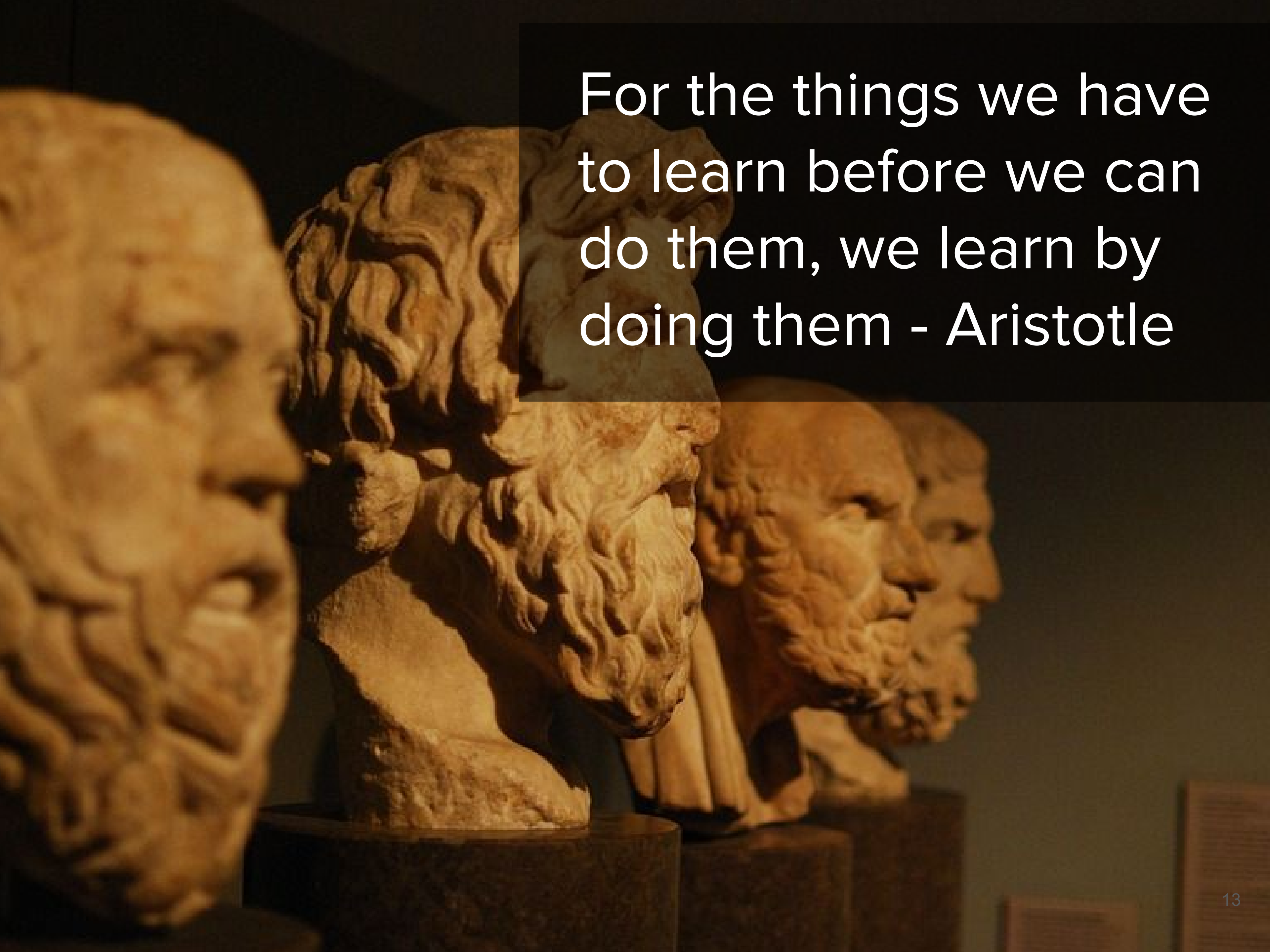
Who is Responsible for Bad Code?

- BOD?
- Management?
- Program Manager?
- Product Manager?
- Unrealistic schedule?

Unprofessional Software Engineers.

It is your job to **defend** your code with **equal passion.**

It is a **crime** if a surgeon does not "**clean**" his hands



For the things we have to learn before we can do them, we learn by doing them - Aristotle

Solution

- Naming
- Function
- Comment
- Formatting
- Objects and Data Structures
- Error Handling
- Class
- Emergence

Naming

> “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.*”
>
> - Robert C. Martin

Naming

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

❌

Say that we’re working in a mine sweeper game.

Naming

Use intention-revealing names

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

✓

Naming

Use intention-revealing names

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

✓

Naming

Avoid misleading and unmeaningful distinctions

```
int a = l;
if (O == l)
  a = O1;
else
  l = 01;
```

MISLEADING ✖

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

UNMEANINGFUL DISTINCTIONS ✖

Naming

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

CAN YOU PRONOUNCE THEM? ❌

Naming

Use pronounceable names

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

✓

Naming

Classes and objects should be noun, methods should be verb

```
public class Point {
    private int x, y;
    public Point(int x, int y) {...}
    public int getX() {...}
    public void setX(int x) {...}
    public int getY() {...}
    public void setY(int y) {...}
}
```

✓

Naming

```
public class Circle {
   double radius;
   String color;

   public double getRadius() {...}
   public String fetchColor() {...}
   public double retrieveArea() {...}
}
```

GET OR FETCH OR RETRIEVE? ❌

Naming

Pick one word per concept

```
public class Circle {
   double radius;
   String color;

   public double getRadius() {...}
   public String getColor() {...}
   public double getArea() {...}
}
```

✓

Naming

```
for (int i=0; i<34; i++) {
    s += (t[i]*4)/5;
}
```

NOT EASY TO SEARCH ❌

if single-letter names and numeric constants are used across body of text, they are not easy to locate

Naming

Use searchable names

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int i=0; i < NUMBER_OF_TASKS; i++) {
   int realTaskDays = taskEstimate[i] * realDaysPerIdealDay;
   int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
   sum += realTaskWeeks;
}
```

✓

Function

> The first rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that.*

- Robert C. Martin

```java
public static String renderPageWithSetupsAndTeardowns(
     PageData pageData, boolean isSuite
   ) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
   WikiPage testPage = pageData.getWikiPage();
   StringBuffer newPageContent = new StringBuffer();
   includeSetupPages(testPage, newPageContent, isSuite);
   newPageContent.append(pageData.getContent());
   includeTeardownPages(testPage, newPageContent, isSuite);
   pageData.setContent(newPageContent.toString());
  }
  return pageData.getHtml();
}
```

TOO BIG ✗

Function

Do one thing

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
    if (isTestPage(pageData))
       includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}
```

✓

Function is doing more than “one thing” if you can extract another function from it with a name that is not merely a restatement

Function

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

if (set(”username”, ”unclebob”))
```

❌

Is it asking whether the “`username`” attribute was previously set to “`unclebob`”? Or is it asking whether the “`username`” attribute was successfully set to “`unclebob`”?

Function

Do something or answer something, but not both

```
if (attributeExists(”username”)) {
     setAttribute(”username”, ”unclebob”);
}
```

✓

Function

Transformational function should appear as the return value

```
void transform(StringBuffer out)
```

✗

These implementation simply returns the input argument

```
StringBuffer transform(StringBuffer in)
```

✓

Function

Avoid too many function parameters

```
Car createCar(float wheelDiameter, float
wheelColor, float wheelMaterial, float
wheelManufacturer, String engineType, String
engineColor, String engineManufacturer);
```

✗

Too many parameters will make function hard to be understood and maintained.

```
Car createCar(Wheel wheel, Engine engine);
```

✓

Comments

> Good code is its own best documentation. As you’re about to add a comment, ask yourself, ‘How can I improve the code so that this comment isn’t needed?'
>
> \- Steve McConnell

Comments

Redundant Comments

The comment is not more informative than the code

```
// Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(
    final long timeoutMillis) throws Exception {
    if(!closed)  {
       wait(timeoutMillis);
       if(!closed)
           throw new Exception(
               "MockResponseSender could not be closed");
    }
}
```

❌

Comments

Noise Comments

Only restate the obvious and provide no new information

```
/**
 * Returns the day of the month.
 * @return the day of the month.
 */
public int getDayOfMonth() {
    return dayOfMonth;
}
```

❌

Comments

Explain yourself in code

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

✗

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

✓

Comments

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile("\\d*:\\d*:\\d* \\w*,
    \\w* \\d*, \\d*");
```

INFORMATIVE ✓

```
public void writeJournal(Diary diary) {
    for (Thread thread : threadList) {
        thread.run(() -> {
            // Prevent original object to be modified
            Diary copyOfDiary = new Diary(diary);
            write(copyOfDiary);
        });
    }
}
```

EXPLANATION OF INTENT ✓

Comments

```
public static SimpleDateFormat makeStandardHttpDateFormat() {
    //SimpleDateFormat is not thread safe,
    //so we need to create each instance independently.
    SimpleDateFormat df = new SimpleDateFormat(
        ”EEE, dd MMM yyyy HH:mm:ss z”);
    df.setTimeZone(TimeZone.getTimeZone(”GMT”));
    return df;
}
```

WARNING OF CONSEQUENCES ✓

Formatting

Formatting

Variables

Should be declared as close to their usage as possible

```
public void paySalary(Employee employee) {
    float bonus;
    float totalSalary;
    bonus = calculateBonus(employee.getSalary());
    totalSalary = bonus + employee.getSalary();
    sendMoney(employee, totalSalary);
}
```

❌

Formatting

Variables

Should be declared as close to their usage as possible

```java
public void paySalary(Employee employee) {
    float bonus;
    bonus = calculateBonus(employee.getSalary());
    float totalSalary;
    totalSalary = bonus + employee.getSalary();
    sendMoney(employee, totalSalary);
}
```

✓

Formatting

Instance Variables

Should be declared at the top of the class

```
public class Employee {
    private String name;
    private String id;
    private float salary;

    public void getName() { … }
}
```

✓

Formatting

Dependent Functions

Should be vertically close, and the caller should be above the called

```
public void paySalary() {
    calculateBonus(salary);
}

private float calculateBonus(float salary) {
    return (salary / 10);
}
```

✓

Formatting

Conceptual *Affinity*

Certain bits of code want to be near other bits

```
public class Employee {
    public void payTax() {
    }
    public void payOverdueTax(Date date) {
    }
    public void increaseSalary() {
    }
    public void decreaseSalary() {
    }
}
```

✓

Formatting

Space

```
public float volume (float length,float width,float height) {
    // code
}
```

✗

```
public float volume(float length, float width, float height) {
    // code
}
```

✓

Formatting

Horizontal Alignment

```
public class WebService {
    private   Request         request;
    private   Response        response;
    private   FitnesseContext context;
    protected long            requestTimeLimit;
}
```

DIFFICULT TO MAINTAIN ✖

Formatting

Horizontal Alignment

```
public class WebService {
    private Request request;
    private Response response;
    private FitnesseContext context;
    protected long requestTimeLimit;
}
```

✓

Formatting

Indentation

```
public String functionName() {return "";}
```

✗

```
public String functionName() {
    return "";
}
```

✓

Objects and Data Structures

Objects and Data Structures

Data structures expose data and have no behavior

Data structures make it easy to add functions without the need to modify existing structures.

```
public class Point {
    public double x;
    public double y;
}
```

✓

Objects and Data Structures

Object expose behavior and hide data

Objects make it easy to add classes without the need to modify existing functions.

```
public class Vehicle {
    public getFuelCapacity() {...}
    public getPercentageFuelRemaining() {...}
}
```

✓

Objects and Data Structures

Law of Demeter

Given method *f* of class C, *f* should only call methods of:

- C
- An Object created by *f*
- An Object passed as an argument to *f*
- An instance variable of C

Objects and Data Structures

Law of Demeter

Given method *f* of class C, *f* should only call methods of:

- C

```
public class Vehicle {
    public getFuelCapacity() {...}

    public getPercentageFuelRemaining() {
        return (fuel / getFuelCapacity() * 100);
    }
}
```

✓

Objects and Data Structures

Law of Demeter

Given method *f* of class C, *f* should only call methods of:

- An Object created by *f*

```java
public void registryEmployee() {
    Employee newEmployee = new Employee();
    registry.createEmplyeeId(newEmployee);
    employeeIdList.add(newEmployee.getId());
}
```

✓

Objects and Data Structures

Law of Demeter

Given method *f* of class C, *f* should only call methods of:

- An Object passed as an argument to *f*

```
public void calculateEmployeeBonus(Employee employee) {
    return (employee.getSalary() / 10);
}
```

✓

Objects and Data Structures

Law of Demeter

Given method *f* of class C, *f* should only call methods of:

- An instance variable of C

```
public class Car {
    private Engine engine;

     public String getCarFuelType() {
          return engine.getFuelType();
    }
}
```

✓

Error Handling

Error Handling

Prefer exceptions to return error code

```
if (deletePage(page) == E_OK) {
    if (registry.delete(page.reference) == E_OK) {
        logger.log(“page deleted”);
    } else {
        logger.log(“delete registry failed”);
    }
} else {
    logger.log(“delete failed”);
}
```

❌

Error Handling

Prefer exceptions to return error code

```
try {
    deletePage(page);
    registry.delete(page.reference);
} catch (Exception e) {
    logger.log(e.getMessage());
}
```

✓

Error Handling

- **Functions should do one thing and error handling is one thing**
  - Implement the normal flow of the function
- **Don’t return NULL**
  - So other function doesn’t need to implement error handling
- **Don’t pass NULL**
  - So other function doesn’t need to implement error handling

Class

Class

Class Organization

Declare the constants, variables, and methods in this order:

```java
public class TimeCalculator {
    public static final int TIME = 25; // public static constant
    public static int DURATION = 25; // private static variables
    private int now;               // private instance variables
    public int addTime(int time) {           // public functions
        ...
        configTime = getConfigTime()
        ...
    }
    private int getConfigTime() { … }       // private utilities
}
```

✓

Class

Classes should be small!

- The first rule is that they should be small
- The second rule is that they should be smaller than that

Single Responsibility Principle (SRP)

- A class or module should have one, and only one, reason to change
- SRP is one of the more important concept in OO design

Class

Refactored:

```
public class TimeCalculator {
    ...
    public int addTime(int time) { … }
    ...
}
```

✓

```
public class UserConfiguration {
    ...
    public int getConfigTime() { … }
    ...
}
```

✓

Class

Cohesion

- Classes should have a small number of instance variables
- The more variables (or class modules) a method manipulates the more cohesive that method is to its class.
- A class in which each variable is used by each method is maximally cohesive.
- Maintaining cohesion results in many small classes

```
public class CustomStack {
    private int topOfStack = 0;
    private int duration = 100;
    List<Integer> elements = new LinkedList<Integer>();
    public int size() { … }
    public void push(int element) { … }
    public int pop() throws PoppedWhenEmpty { … }

    public void sleep() {
        TimeUnit.SECONDS.sleep(duration);
    }
    public void log() {
        ...
        logger.log(Level.WARNING, ”This is a warning!”);
        ...
    }
    ...
}
```

LOW COHESION ❌

```java
public class Stack {
    private int topOfStack = 0;
    List<Integer> elements = new LinkedList<Integer>();
    public int size() {
        return topOfStack;
    }
    public void push(int element) {
        topOfStack++;
        elements.add(element);
    }
    public int pop() throws PoppedWhenEmpty {
        if (topOfStack == 0)
            throw new PoppedWhenEmpty();
        int element = elements.get(--topOfStack);
        elements.remove(topOfStack);
        return element;
    }
}
```

HIGH COHESION ✓

Emergence

Emergence

- Runs all the tests. To make it easy, make sure:
    - Low coupling
    - SRP
- Contains no duplication (*Refactoring*)
- Expresses the intent of the programmer (*Refactoring*)
    - Easy to read and understand
- Minimizes the number of classes and methods (*Refactoring*)
    - But don’t take it too far

```java
public void scaleToOneDimension(…) {
    …
    RenderedOp newImage = ImageUtilities.getScaledImage(
        image, scalingFactor, scalingFactor);
    image.dispose();
    System.gc();
    image = newImage;
}

public synchronized void rotate(int degrees) {
    RenderedOp newImage = ImageUtilities.getRotatedImage(
        image, degrees);
    image.dispose();
    System.gc();
    image = newImage;
}
```

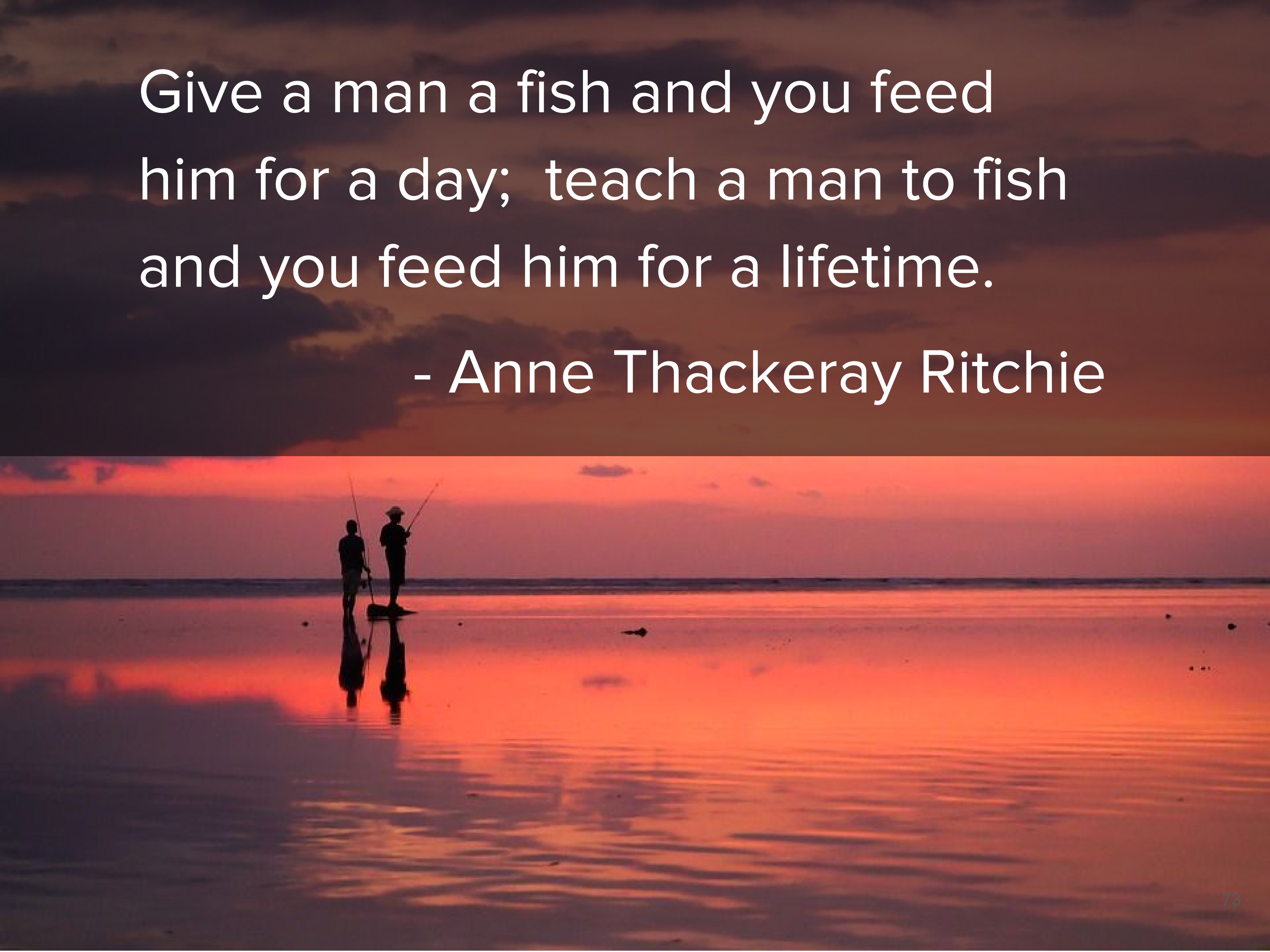
DUPLICATION ❌

```java
public void scaleToOneDimension(…) {
    …
    replaceImage(ImageUtilities.getScaledImage(
        image, scalingFactor, scalingFactor));
}

public synchronized void rotate(int degrees) {
    replaceImage(ImageUtilities.getRotatedImage(
        image, degrees));
}

private void replaceImage(RenderedOp newImage) {
    image.dispose();
    System.gc();
    image = newImage;
}
```

DUPLICATION REFACTORED ✓



Give a man a fish and you feed him for a day; teach a man to fish and you feed him for a lifetime.

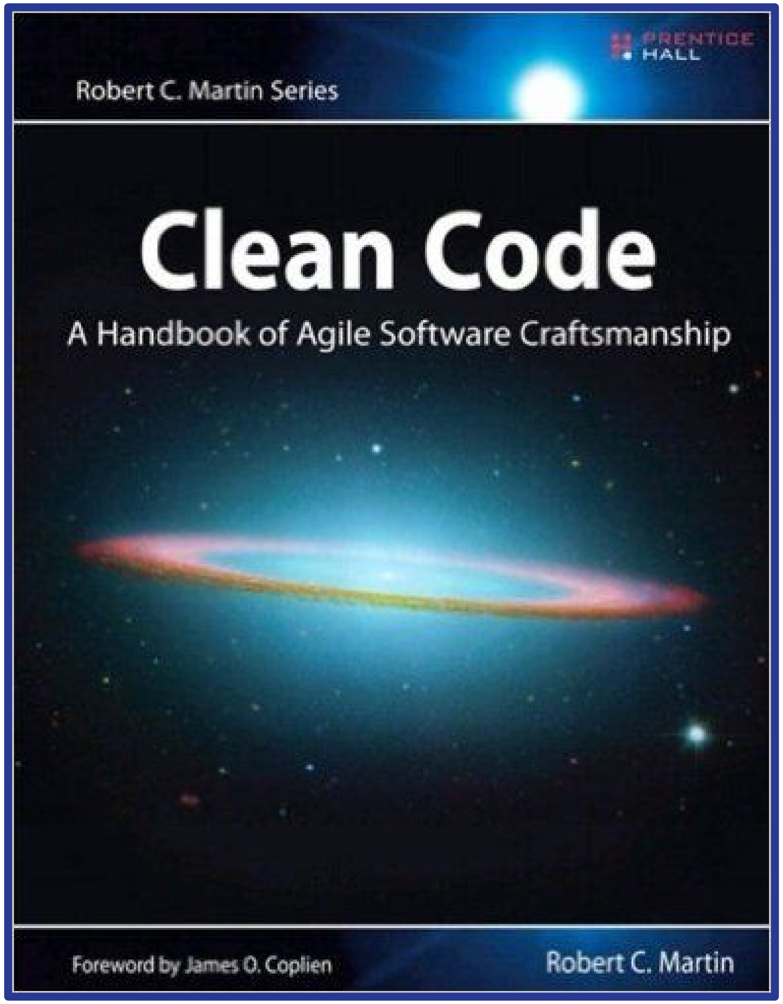
- Anne Thackeray Ritchie

Conclusion

1. Practice

   Is there a set of simple practices that can replace experience? Clearly not

2. Beware of *Bad Code* or *Code Smell*
3. Refactor

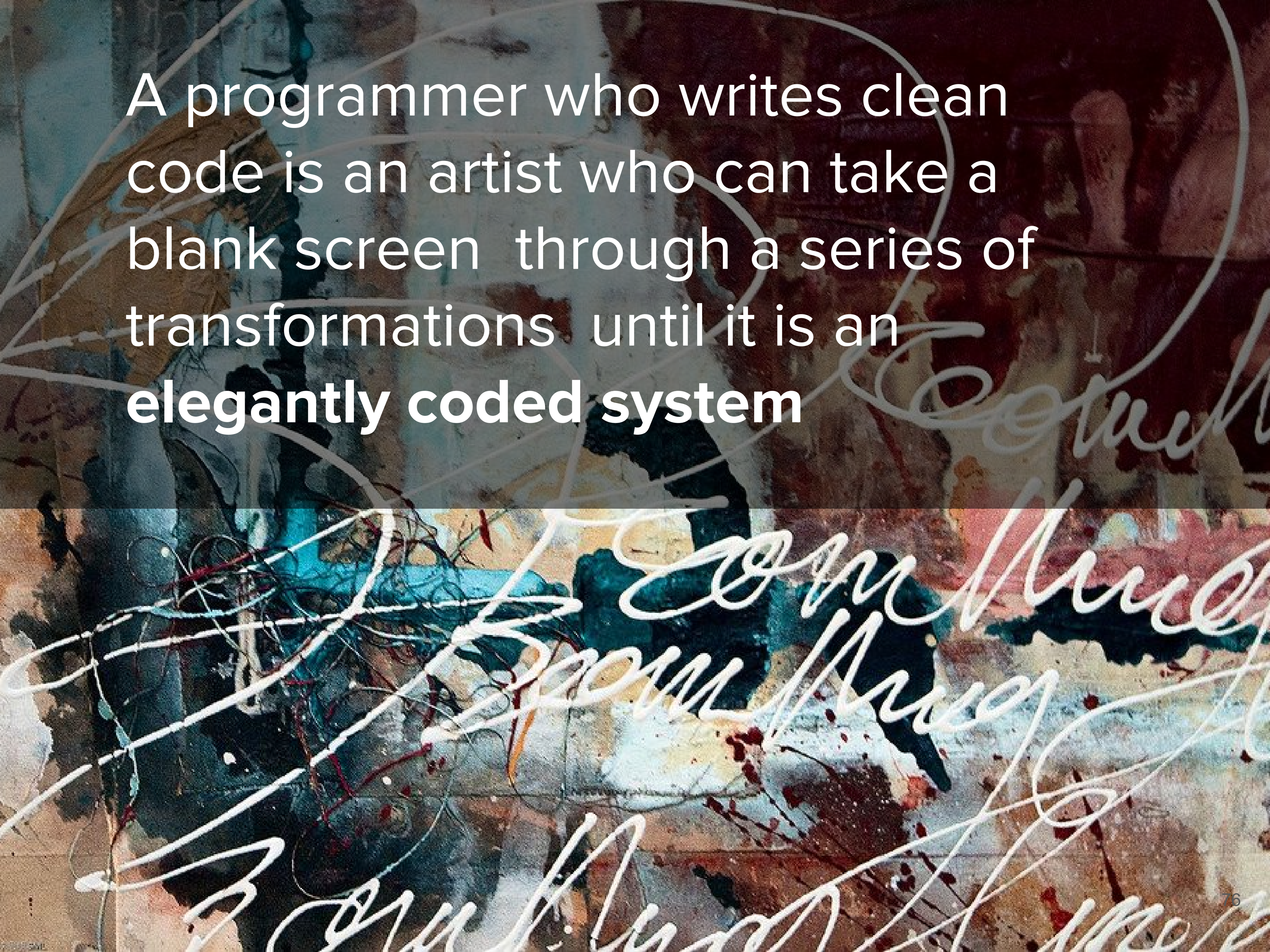

PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

Clean Code:

A Handbook of Agile Software Craftsmanship

Robert C. Martin

©2009 | Prentice Hall



A programmer who writes clean code is an artist who can take a blank screen through a series of transformations until it is an **elegantly coded system**

Thank You

Q & A

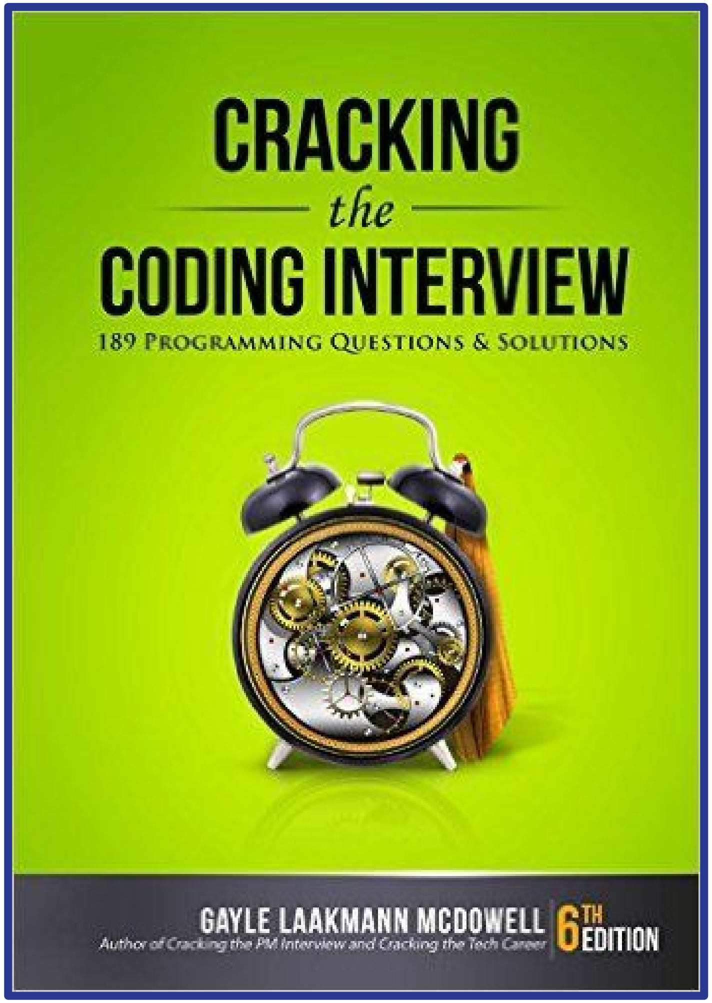
speakerdeck.com/gdplabs

gdp
LABS

We Are Hiring!

✉ jobs@gdplabs.id

gdp LABS


CRACKING
the
CODING INTERVIEW
189 PROGRAMMING QUESTIONS & SOLUTIONS
GAYLE LAAKMANN MCDOWELL
Author of Cracking the PM Interview and Cracking the Tech Career
6TH EDITION

Cracking the Coding Interview:

189 Programming Questions and Solutions

Gayle Laakmann McDowell

©2015 | CareerCup

Extras ⊕

- Code Smells
- OO Design Principles

Code Smells

- Large class
- Feature envy
- Inappropriate intimacy
- Refused bequest
- Lazy class
- Excessive use of literals
- Cyclomatic complexity
- Data clump
- Orphan variable or constant class
- Duplicated code
- Too many parameters
- Long method
- Excessive return of data
- Excessively long identifiers
- Excessively short identifiers

OO Design Principles: S.O.L.I.D

- **S** Single-responsibility principle
- **O** Open-closed principle
- **L** Liskov substitution principle
- **I** Interface segregation principle
- **D** Dependency Inversion Principle