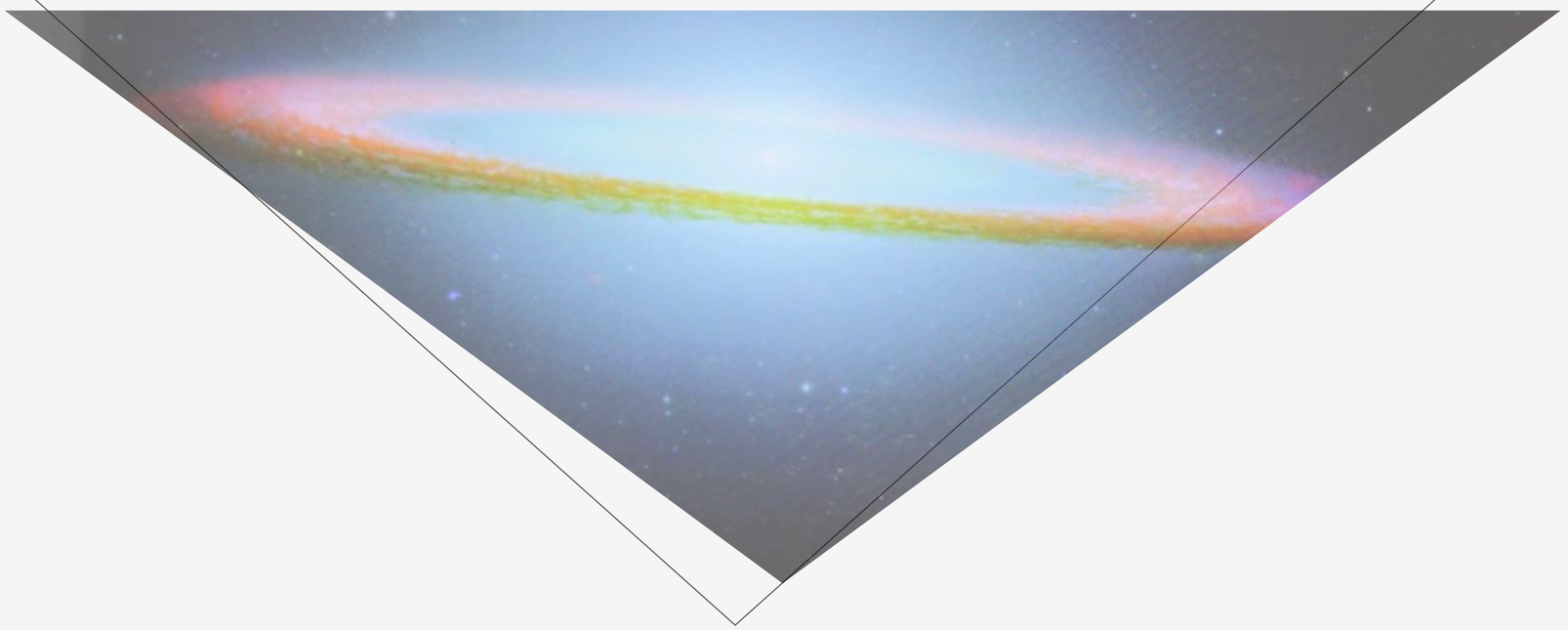


CLEAN CODE

A HANDBOOK OF AGILE SOFTWARE CRAFTSMANSHIP

ROBERT C. MARTIN

INDEX

Quien soy

aloaisa

1

CLEAN CODE

CRAFTSMANSHIP

There are two parts to learning craftsmanship: knowledge and work.

WANT
TO BE BETTER

"You are reading this book for two reasons. First, you are a programmer. Second, you want to be a better programmer. We need better programmers."

ROBERT C. MARTIN

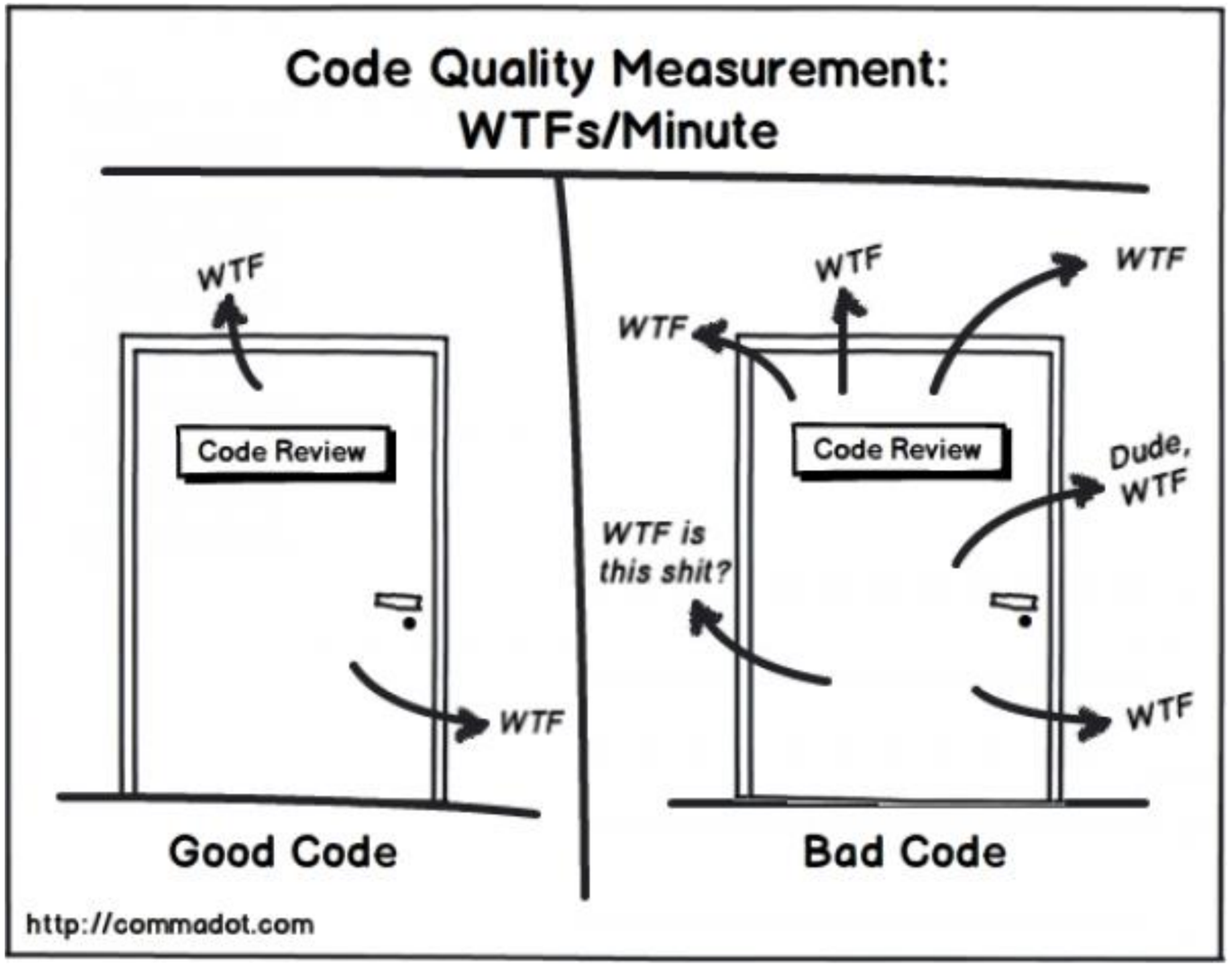
LEARN & PRACTICE

It requires much time and effort, but we think it's worth.

MEASUREMENT

CODE QUALITY

WTFs/minute



GOOD/BAD CODE

WHAT IS CLEAN CODE?

The main source of Bad Code are we, the programmers. We are not professionals.

ELEGANT

BJARNE STROUSTRUP

- I like my code to be elegante and efficient

- Clean code does one thing well

SIMPLE, DIRECT, PROSE

GRADY BOOCH

- Clean code is simple and direct
- Clean code reads like well-written prose

LITERATE

DAVE THOMAS

- Clean code can be read

- Clean code shuold be literate

CARE

MICHAEL FEATHERS

- Clean code always looks like it was written by someone who cares

SMALL, EXPRESSIVE, SIMPLE

RON JEFFRIES

- Reduced duplication, high expressiveness, and early building of simple abstractions

WHAT YOU EXPECTED

WARD CUNNINGHAM

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

ROBERT C. MARTIN

THE BOY SCOUT RULE

"Always leave the campground cleaner than you found it."

If you find a mess on the ground, you clean it up regardless of who might have made the mess.

UNCLE BOB.

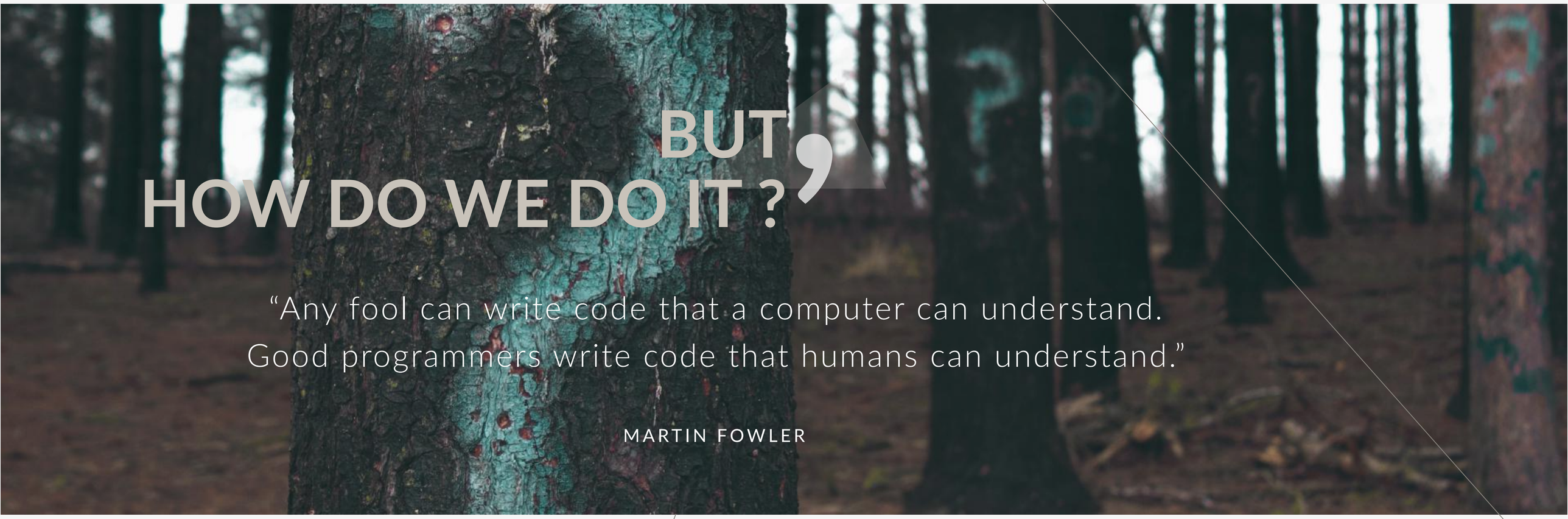
THE BROKEN WINDOW THEORY

Consider a building with a few broken windows. If the windows are not repaired, the tendency is for vandals to break a few more windows

MONEY

THE COST OF THE BAD CODE

Time, Productivity, Stress and Proyects.
All it’s about the Money $$$



BUT, HOW DO WE DO IT ?

“Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.”

MARTIN FOWLER

MEANINGFUL NAMES

In the code, the names are everywhere. Variables, Functions, Arguments, Classes and Packages, Directories and Files, Jars, Wars and Ears.

NAMING:

USE INTENTION-REVEALING NAMES

- **getLinearSearchPosition**: -- indicates how the method works
- **getSearchPosition**: -- is better
- **getPosition**: -- is even better

NAMING:

AVOID

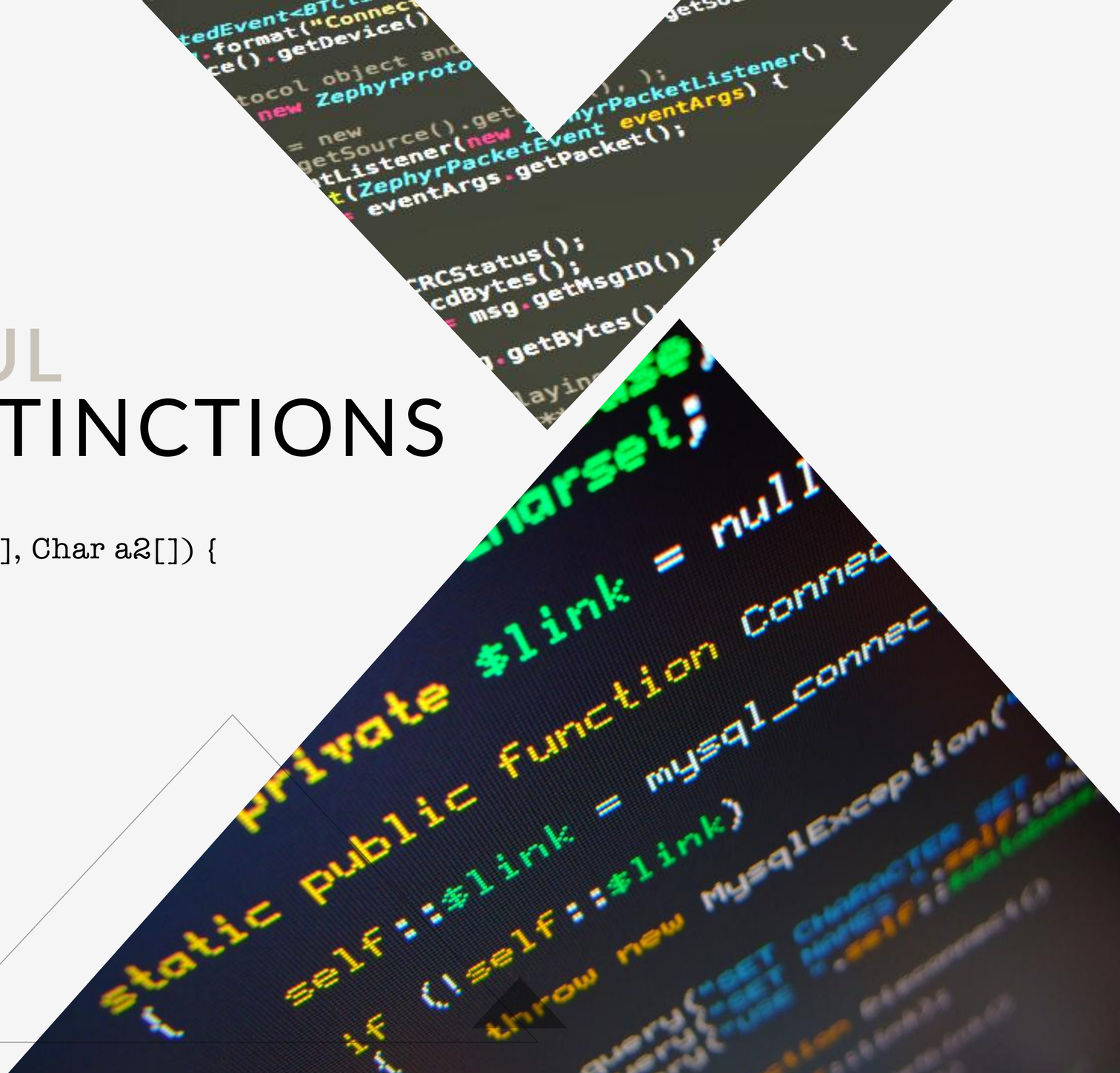
- DISINFORMATION

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

NAMING:

- MAKE MEANINGFUL DISTINCTIONS

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



NAMING:

```
class DtaCrc {
  private Date genymdhms;
  ...
```

```
class Customer {
  private Date generationTimestamp;
  ...
```

```
PhoneNumber phoneString;
// name not changed when type changed!
```

PRONOUNCEABLE NAMES

SEARCHEABLE

```
int realTaskWeeks =
    (realDays / WORK_DAYS_PER_WEEK)
```

HUNGARIAN NOTATION

MEMBER PREFIXES

```
public class Part {
    private String m_dsc;
    void setName(String name) {
        m_dsc = name;
    }
}
```

NAMING:

```
for (a = 0; a < 10; a++)
   for (b = 0; b < 10; b++)
  ...
```

```
for (i = 0; i < 10; i++)
   for (j = 0; j < 10; j++)
  ...
```

```
postpayment, deletePage, save, ...
// methods should have verb or verb phrase names

String name = employee.getName();
customer.setName("Alvaro");
if (paycheck.isPosted())...

Complex complex = Complex.fromRealNumber(23.0);
// is generally better than
Complex complex = new Complex(23.0);
```

AVOID MENTAL MAPPING

CLASS NAMES

Manager, Processor, Data, Info
// a class name should not be a verb

Customer, WikiPage, Account, AddressParser

METHOD NAMES

PICK ONE WORD PER CONCEPT

fetch, retrieve, get //As equivalen methods

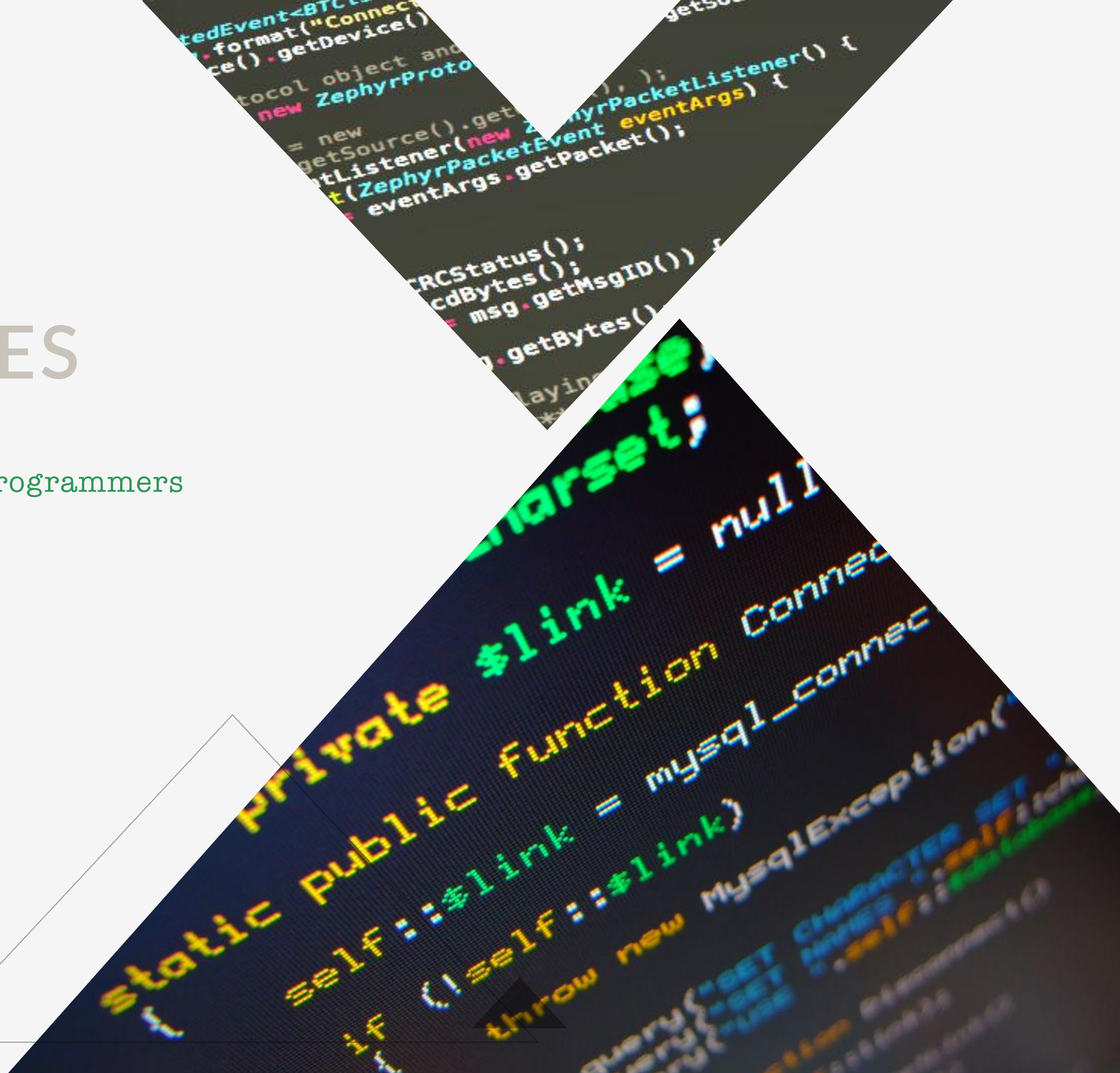
controller, manager, driver //Confusing

NAMING:

- USE DOMAIN NAMES

AccountVisitor, JobQueue

```
// people who read you code will be programmers

// Don't Add Gratuitous Context
class Address {
  private String nameAddress;
  ...
```



3

CLEAN CODE

FUNCTIONS

The first rule of functions is that they should be small.

The second rule of functions is that they should be smaller than that.

FUNCTIONS:

DO

ONE THING

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

```
// < 150 CARACTERES PER LINE
// < 20 LINES
```

FUNCTIONS:

```
// high level of abstraction
getHTML();

// intermediate level of abstraction
String pagePathName = PathParse.render(pagePath);

// remarkably low level
message.append("\n");
```

```
class Employee...
 int payAmount() {

  switch (getType()) {
    case EmployeeType.ENGENIEER:
      return salary;

    case EmployeeType.MANAGER:
      return salary + bonus;
    ...
```

```
class EmployeeType...
 abstract int payAmount() {

class Engenieer extends EmployeeType...
  int payAmount() {
   return salary;
  }

 class Manager extends EmployeeType...
  int payAmount() {
   return salar + bonus;
  }
```

ONE LEVEL OF ABSTRACTION

THE STEPDOWN RULE

```
// Reading Code from Top to Bottom

// we want the code to read like a
//     top-down narrative
```

SWITCH STATEMENTS

USE DESCRIPTIVE NAMES

```
// Don't be afraid to make a name long.
// A long descriptive name is better than a
// short enigmatic one.

   // Experiment with names to find the
   // ones  that clarify the design of your
     // module
```

FUNCTIONS:
ARGUMENTS

```
// if a función is going to transform its input argument
// the transformation shuold appear as the return value

InputStream openFile = fileOpen(“My File”);
```

COMMON MONADIC FORM

FLAG ARGUMENTS

```
render(true);

renderForSuite();
renderForSingleTest();
```


```
writeField(name);
// is easier to understand than
writeField(outputStream, name);

// perfectly reasonable
Point point = new Point(0, 0);

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

DYADIC/TRIAD FUNCTIONS

ARGUMENT OBJECTS

```
Circle makeCircle(double x, double y,
                  double radius);

Circle makeCircle(Point center,
                  double radius);
```

FUNCTIONS:

```
write(name);
writeField(name);

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

Functions should:
- do something (change the state of an object) — a **command**
- return something (return some information about that object) — a **query**

but not do both, as doing both often leads to confusion.

The **solution is to separate the command from the query**, by creating two functions from the one that did two things.

VERBS AND KEYWORDS

HAVE NO SIDE EFFECTS

COMMAND QUERY SEPARATION

OUTPUT ARGUMENTS

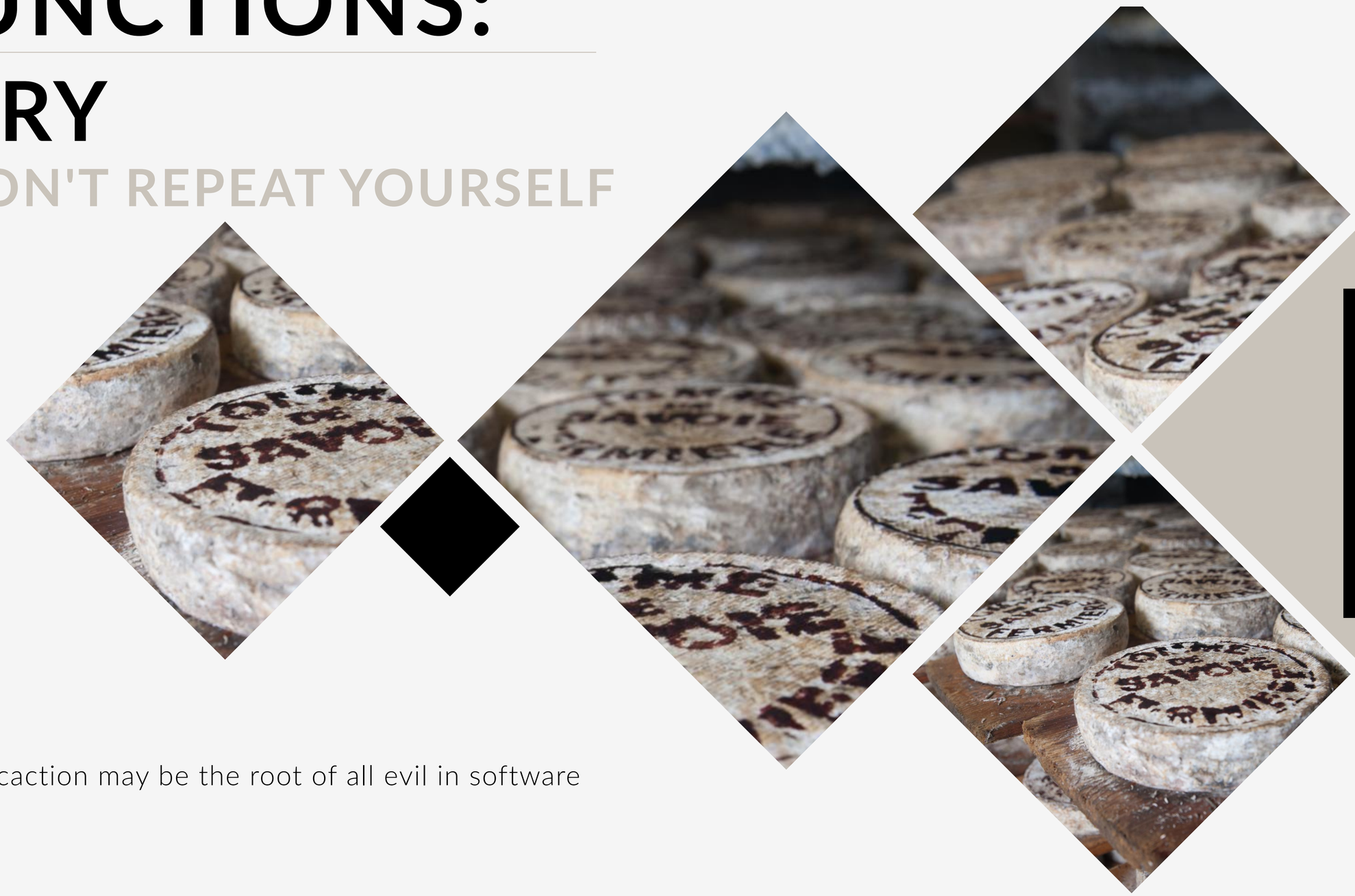
```
public boolean checkPassword(User user, String password) {
  String codedPhrase = user.getPhraseEncodedByPassword();
  if (cryptographer.decrypt(codedPhrase, password)) {
     Session.initialize();
     return true;
  }
  return false;
}
```


```
// In general output arguments should
// be avoided.
// If your function must change the state
//   of something, have it change the state
//   of its owning object.
```

FUNCTIONS:

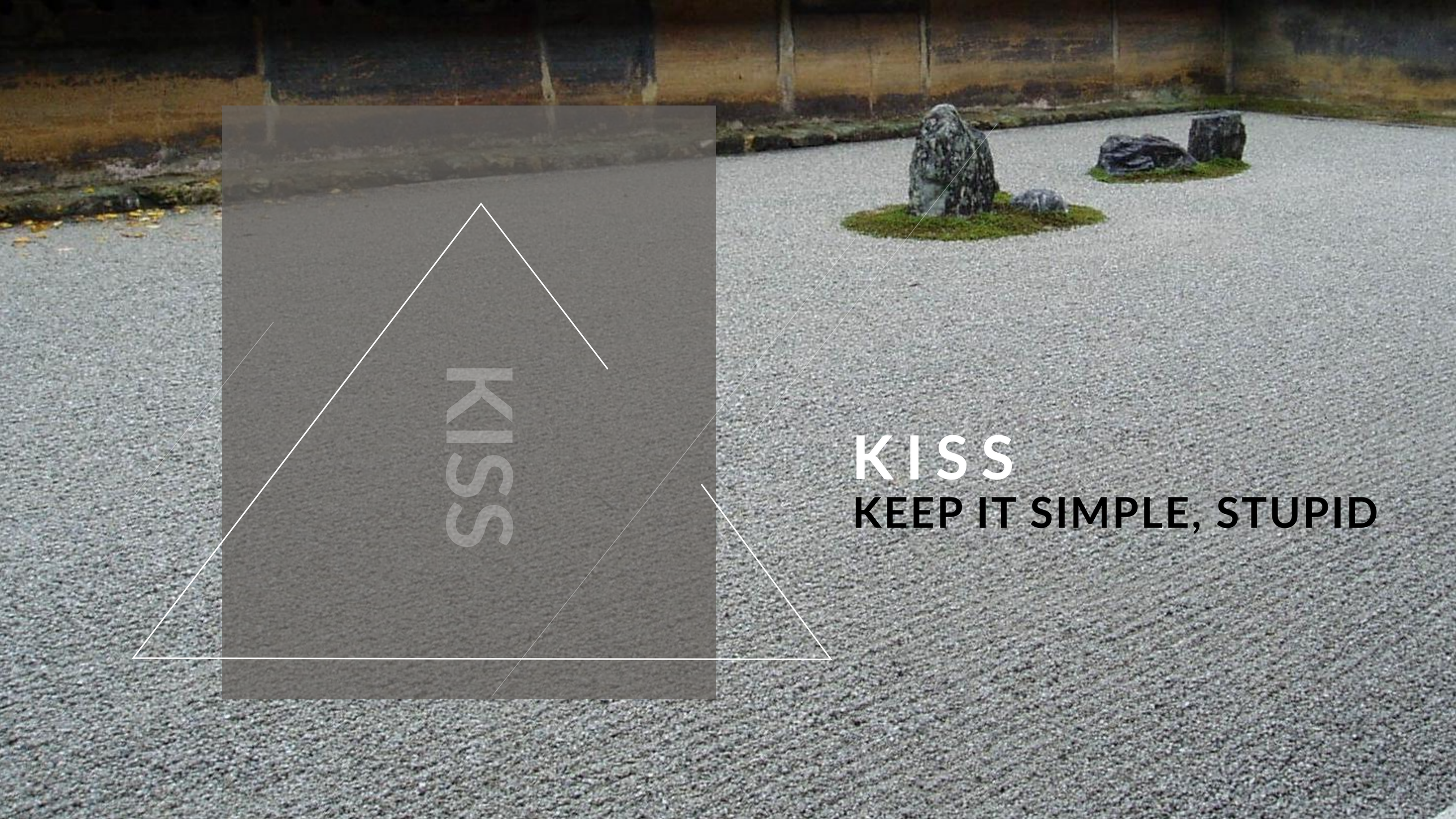
DRY

DON'T REPEAT YOURSELF



DRY

Duplicaction may be the root of all evil in software



KISS

KEEP IT SIMPLE, STUPID

4

CLEAN CODE

COMMENTS

- Comments Do Not Make Up for Bad Code
- Don't comment bad code, rewrite it!
- Explain Yourself in Code

COMMENTS:

GOOD

```
/*****************************************
 * Copyright (C) 2010-2011 {name} <{email}>
 *
 * This file is part of {project}.
 *
 * {project} can not be copied and/or distributed
 *  without the express permission of {name}
 *****************************************/
```

LEGAL COMMENTS

```
// This is our best attempt to get a race condition
// by creating large number of threads.
for (int i = 0; i < 25000; i++) {
    Thread....
```

EXPLANATION OF INTENT

INFORMATIVE COMMENTS

```
// return an instance of the Response being tested
Protected abstract Responder responderInstance();

// Format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern tomeMatcher = Pattern.compile(
“\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*”);
```

CLARIFICATION

```
assertTrue(a.compareTo(b == -1)); // a < b
assertTrue(b.compareTo(a == 1)); // b > a
```

COMMENTS:
GOOD

```
public static SimpleDateFormat makeStandardHttpDateFormat() {
   // SimpleDateFormat is not thread safe,
   // so we need to create each instance independently
   SimpleDateFormat df = new SimpleDateFormat('dd MM yyyy');
   df.setTimeZone(TimeZone.getTimeZone("GMT"));
   return df;
 }
```

WAITING OF CONSEQUENCES

```
String listItemContent = match.group(3).trim();
// the trim is real important. It removes the starting
// spaces that could cause the item to be recognized
// as another list.
...
```

AMPLIFICATION

TODO COMMENTS

```
// TODO We need to test this class
// and controll the Exceptions
```

JAVADOCS IN PUBLIC APIS

Only in publics APIs!

COMMENTS:
BAD

```
try {
   String propertiesPath = propertiesLocation + “\” + PROPERTIES;
   FileInputStream properties = new FileInputStream(propertiesPath);
   loadedProperties.load(properties);
 } catch () {
    // No properties files means all default are loaded
 }
```

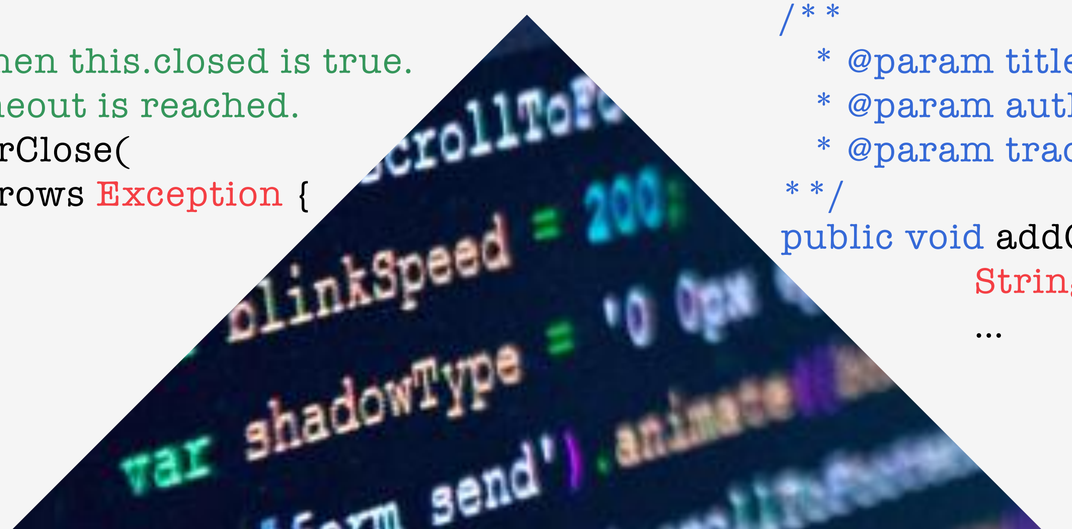
MUMBLING

```
// The Processor delay for this component
protected int backgroundProcessorDelay = -1;

// The lifecycle event support for this component
protected LifecycleSupport lifecycle = new ...

// The container event list for this Container
protected ArrayList eventList = new ArrayList();
```

MISLEADING COMMENTS

REDUNDANT COMMENTS

```
// Utility method that returns when this.closed is true.
// Throws an exception if the timeout is reached.
public synchronized void waitForClose(
        final long timeoutMillis) throws Exception {
        ....
```



MANDATED COMMENTS

```
/**
   * @param title The title of the CD
   * @param author The author of the CD
   * @param tracks The tracks of the CD
**/
public void addCD(String title,
                  String author, int tracks) {
                  ...
```

COMMENTS:

BAD

```
/**************************************************************
 * commit ca82a6dff817ec66f44342007202690a93763949
 * Author: Scott Chacon <schacon@gee-mail.com>
 * Date:   Mon Mar 17 21:52:11 2008 -0700
 *
 *     changed the version number
```

JOURNAL COMMENTS

```
/** The name */
private String name;

/** The version */
private String version;
```

SCARY NOISE

NOISE COMMENTS

```
/**
  *  Default constructor
**/
protected AnnualDateRule() {}

// The day of the month
private int dayOfMonth;
```

DON'T USE A COMMENT

```
// Don't Use a Comment When You Can
//    use a Function or a Variable
```

COMMENTS:
BAD

```
// Private Methods ////////////////////////////////
```

POSITION MAKERS

CLOSING BRACE COMMENTS

```
while (message.isPresent()) {
    ...
} // while
```

```
/**Added by Rick **/
```

ATTRIBUTIONS AND BYLINES

COMMENTED-OUT CODE

```
String name = person.getName();
// String name = “text”;
// loadedProperties.load(properties);
```

COMMENTS:
BAD

```
/**
* <p>Here are my notes and favourite from chapter 4,
* “Comments”, of one of my all-time favourite books:
* <em>
* <a href="..." target="_blank">Clean Code: A Handbook of
Agile Software Craftsmanship</a>
* </em>, by Robert C. Martin.</p>
```

HTML COMMENTS

```
/**
 * SIP is based on an HTTP-like request/response transaction model.
 * Each transaction consists of a request that invokes a particular
 * method, or function, on the server and at least one response.  In
 * this example, the transaction begins with Alice's softphone sending
 * an INVITE request addressed to Bob's SIP URI.  INVITE is an example
 * of a SIP method that specifies the action that the requestor (Alice)
 * wants the server (Bob) to take.  The INVITE request contains a
 * of header fields.  Header fields are named attributes that provide
 * additional information about a message.  The ones present in an
 * INVITE include a unique identifier for the call, the destination
```

TOO MUCH INFORMATION

NONLOCAL INFORMATION

```
/**
 * Port on with fitnesse would run.
 * Default to <b>8082</b>.
*/
public void setFitnessePort(int port)
```

INOBVIOUS CONNECTION

```
/**
 * start with an array that is big enough to hold
 * all the pixel (plus filtre bytes), and an extra
 * 200 bytes for header info
*/
    this.pngBytes = new byte[(
        (this.width + 1) * this.height * 3)
        + 200];
```

SHORT

FUNCTION HEADERS

SHORT FUNCTIONS DON’T NEED MUCH DESCRIPTION

5

CLEAN CODE

FORMATTING

THE PURPOSE OF FORMATTING IS COMMUNICATION

First of all, let's be clear. Code formatting is important.

FORMATTING:
VERTICAL

```
// each blank line is a visual cue
// that identifies a new and separate concept

// another concept...
```

VERTICAL OPENNESS BETWEEN CONCEPTS

VERTICAL DENSITY

```
// Vertical density implies close association
/**
 * the class name of the reportar listener
*/
private String m_className;

/**
 * the properties of the reporter listener
*/
private m_properties = new ArrayList();
```

```
// Variables
// should be declared as  close to their usage as possible

// instance variables
// should be declared at the top of the class

// dependent functions
// if one function calls another, they should be vertically
// close, and the caller should be above the called

// conceptual affinity
// certain bits of code want to be near other bits
```

VERTICAL DISTANCE

THE NEWSPAPER METAPHOR

```
// high-level —> details
```

FORMATTING:
HORIZONTAL

```
public void PersistLogicResultToTheDataAccessObject() {
  Query query = new Query();
  Result result = new Result();
  when(businessLogic.handle(query)).thenReturn(result);
  Response response = controller.doSomethingWith(query);
  assertThat(response.status(), is(OK));
  verify(dataAccessObject.persist(result));
}
```

```
public class FitNesseServer implements SocketServer {
   private FitNesseContext context;
   public FitNesseServer(FitNesseContext context) {
   this.context = context; }
   public void serve(Socket s) { serve(s, 10000); }
   public void serve(Socket s, long requestTimeout) {
   try { FitNesseExpediter sender = new
   FitNesseExpediter(s, context);
   sender.setRequestParsingTimeLimit(requestTimeout);
   sender.start(); }
   catch(Exception e) { e.printStackTrace(); } }
}
```

HORIZONTAL OPENNESS AND DENSITY

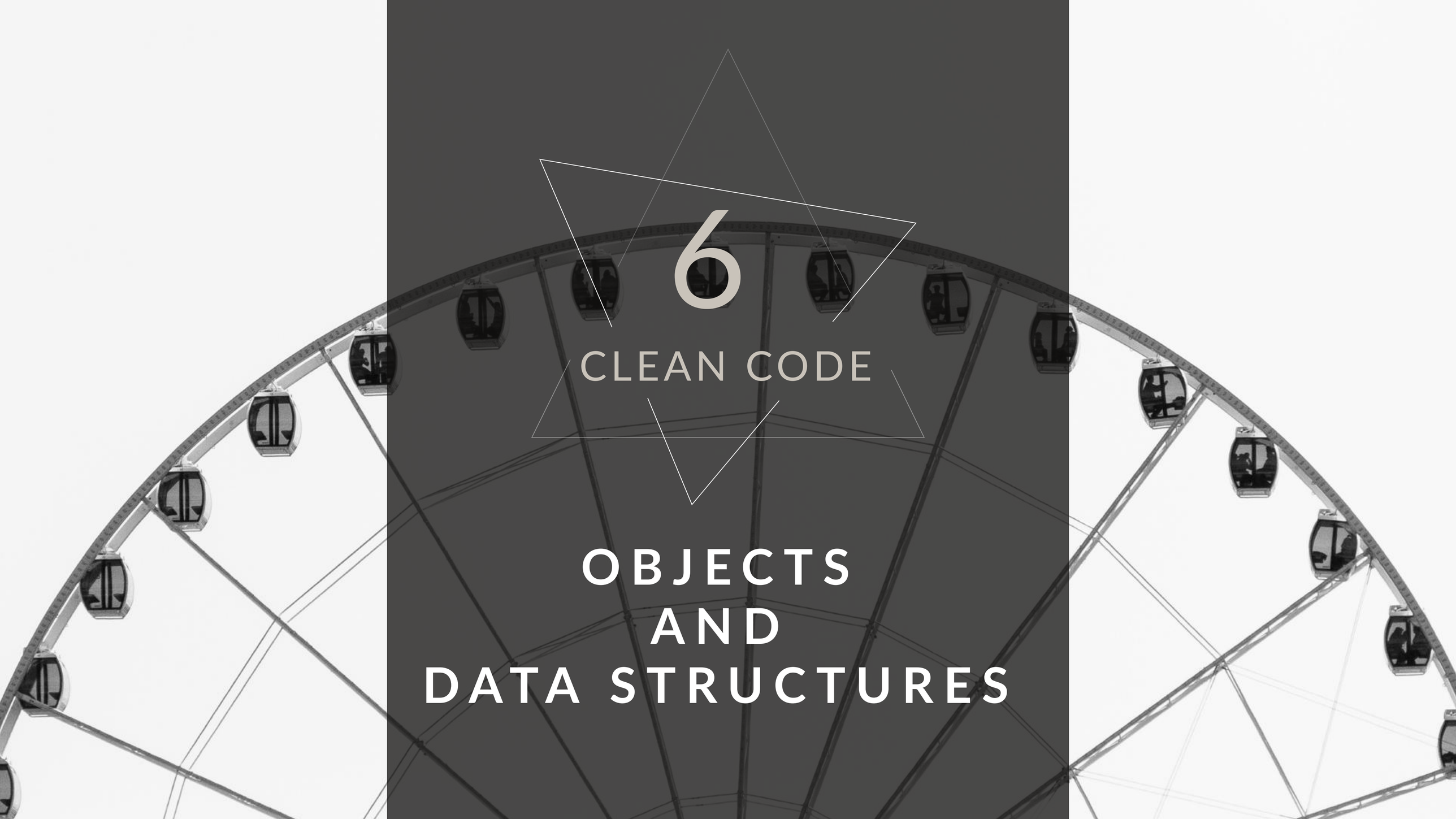
HORIZONTAL ALIGNMENT

BREAKING INDENTATION

TEAM RULES

```
public FitNesseExpediter(Socket              s,
                         FitNesseContext context)
        throws Exception {
   this.context                 = context;
   socket                       = s;
   input                        = s.getInputStream();
   output                       = s.getOutputStream();
   requestParsingTimeLimit = 10000;
}
```


```
// Every Programmer has his own
//    favourite formatting rules.

// But if he works in a team
// then the team rules
```



6

CLEAN CODE

OBJECTS AND DATA STRUCTURES

DATA STRUCTURE:
OBJECTS

```
public interface Vehicle {
    double getFuelTankCapacityInGallons();
    double getGallonsOfGasoline();
}
```

```
public interface Vehicle {
    double getPercentFuelRemaining();
}
```

```
// objects hide their data behind abstractions and
// expose functions that operate on that data

// data structure expose their data and
// have no meaningful functions
```

DATA ABSTRACTION

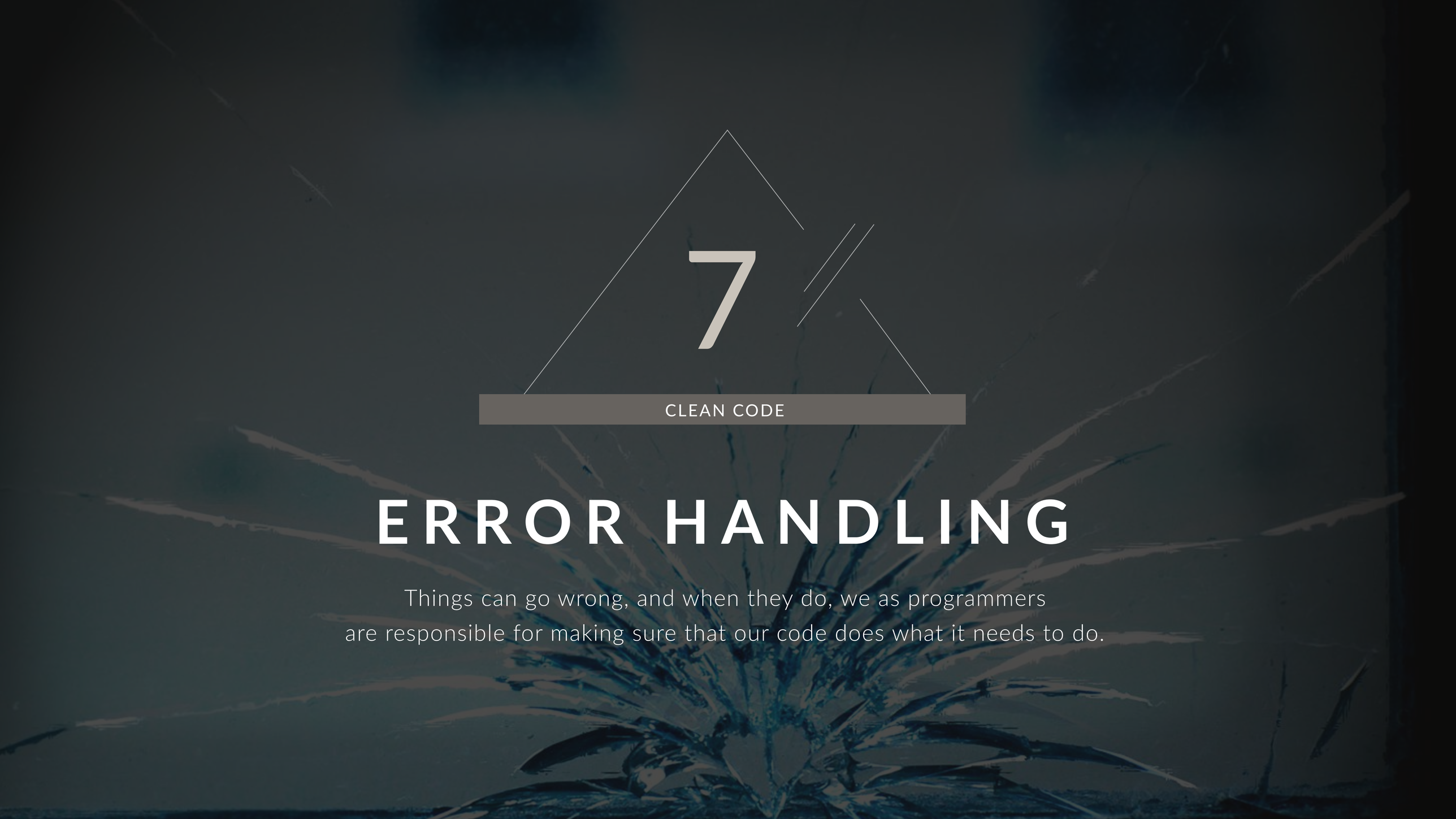
DATA/OBJECT ANTI-SYMMETRY

TRAIN WRECKS

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

THE LAW OF DEMETER

```
// a module should not know about the
//  innards of the objects it manipulates.

final String outputDir = ctxt.getOptions()
                           .getScratchDir()
                           .getAbsolutePath();
```

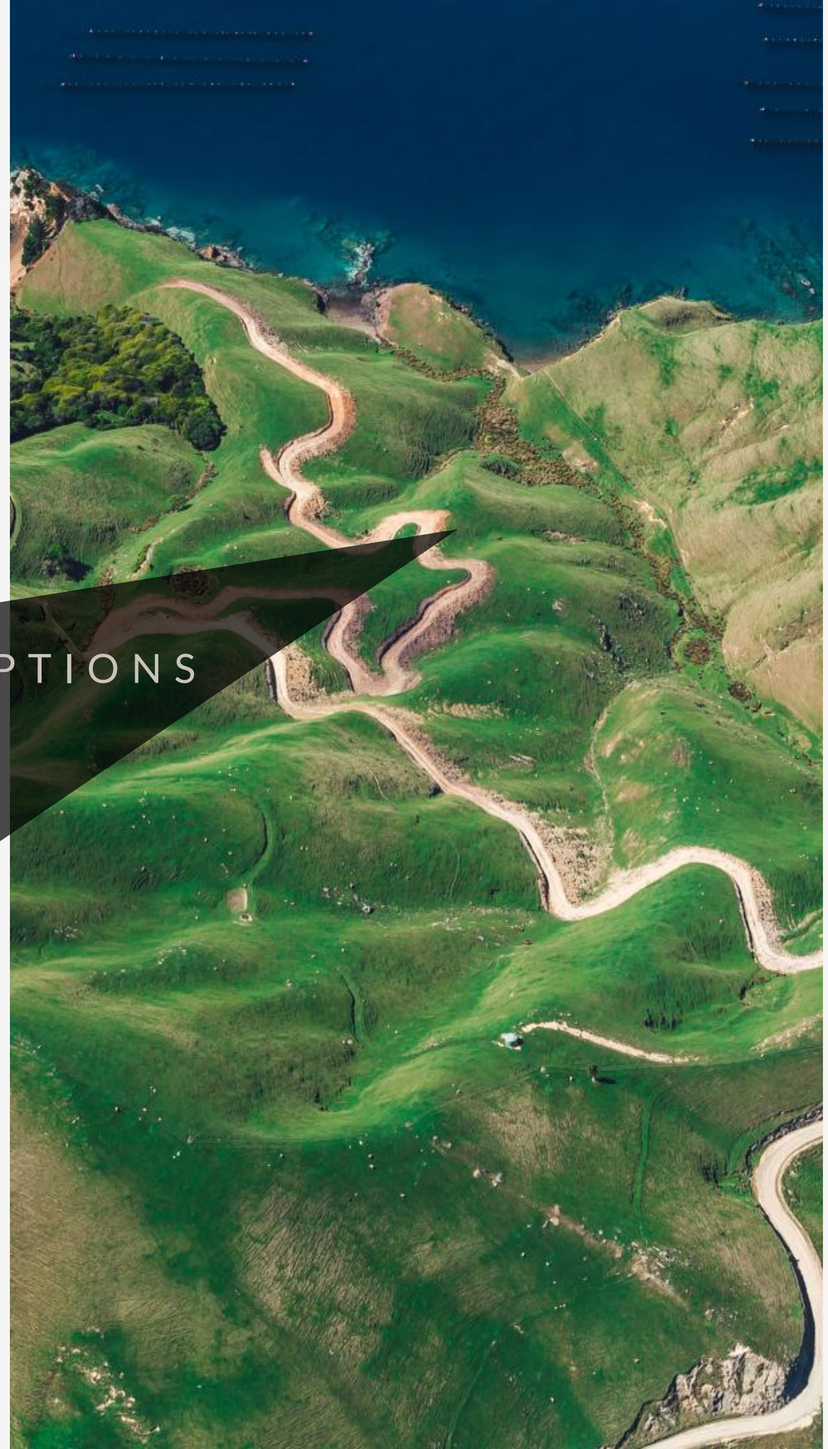


7

CLEAN CODE

ERROR HANDLING

Things can go wrong, and when they do, we as programmers
are responsible for making sure that our code does what it needs to do.



ERROR HANDLING

USE EXCEPTIONS RATHER THAN RETURN CODES

```
if (deletePage(page) == E_OK) {
   if (registry.deleteReference(page.name) == E_OK) {
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
            logger.log("page deleted");
        } else {
           logger.log("configKey not deleted");
        }
    } else {
        logger.log("deleteReference from registry failed");
    }
} else {
   logger.log("delete failed");
   return E_ERROR;
}
```

```
try {
   deletePage(page);
   registry.deleteReference(page.name);
   configKeys.deleteKey(page.name.makeKey());
} catch (Exception e) {
   logger.log(e.getMessage());
}
```

EXTRACT
TRY/CATCH BLOCKS

```
public void delete(Page page) {
   try {
      deletePageAndAllReferences(page);
   }
   catch (Exception e) {
      logError(e);
   }
}

private void deletePageAndAllReferences(Page page)
                     throws Exception {
   deletePage(page);
   registry.deleteReference(page.name);
   configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
   logger.log(e.getMessage());
}
```

ERROR:
HANDLING

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
    }
}
```

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
public List<Employee> getEmployees() {
if ( .. there are no employees .. )
    return Collections.emptyList();
  }
```

```
// Functions should do one thing.
// Error handing is one thing.
```

ERROR HANDLING IS ONE THING

DEFINE THE NORMAL FLOW

```
try {
 MealExpenses expenses = expenseReportDAO
                                .getMeals(employee.getID());
   m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
   m_total += getMealPerDiem();
}
```

DON'T RETURN NULL

DON'T PASS NULL

```
calculator.xProjection(null, new Point(12, 13));
```

8

CLEAN CODE

UNIT TESTS

ONCE I GOT A SUITE OF TESTS TO PASS, I WOULD MAKE SURE THAT THOSE TESTS WERE CONVENIENT TO RUN FOR ANYONE ELSE WHO NEEDED TO WORK WITH THE CODE.

THE THREE 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

UNIT TESTS:

// test code is just as important as production code

KEEPING TEST CLEAN

// what makes a clean test? three things
// readability, readability, and readability

CLEAN TESTS

// the best rule is that you should
// minimize the number of asserts per concept and
// test just one concept per test function

SINGLE CONCEPT PER TEST

// tests come to a single conclusión
// that is quick and easy to understand

ONE ASSERT
PER TEST

F I R S T

FAST

Tests should be fast. They should run quickly

INDEPENDENT

Tests should not depend on each other. One test should not set up the conditions

REPEATABLE

Tests should be repeatable in any environment.

SELF-VALIDATING

The tests should have a boolean output. Either they pass or fail.

TIMELY

The tests need to be written in a timely fashion.

9

CLEAN CODE

CLASSES

CLASS:

// public static constants
// private static variables
// private instance variables
// public functions
// private utilities called by a public function right after

CLASS ORGANIZATION

// a class or module should have one, and only one,
// reason to change and only one responsibility

// SRP is one of the more important concept in OO design

THE SINGLE RESPONSIBILITY PRINCIPLE

SHOULD BE SMALL!

// the first rule is that they should be small
// the second rule is that they should be smaller
// than that

COHESION

// maintaining cohesion results in
// many small classes

10

CLEAN CODE

EMERGENCE!

KENT BECK'S FOUR RULES OF

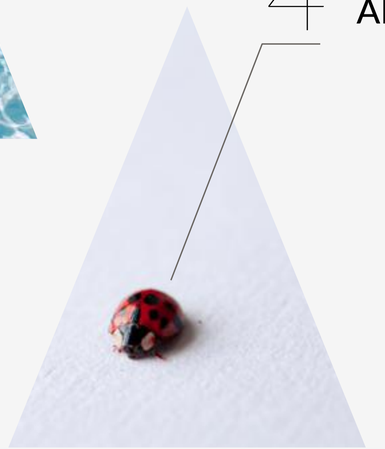
SIMPLE DESIGN

1 RUNS ALL THE TESTS

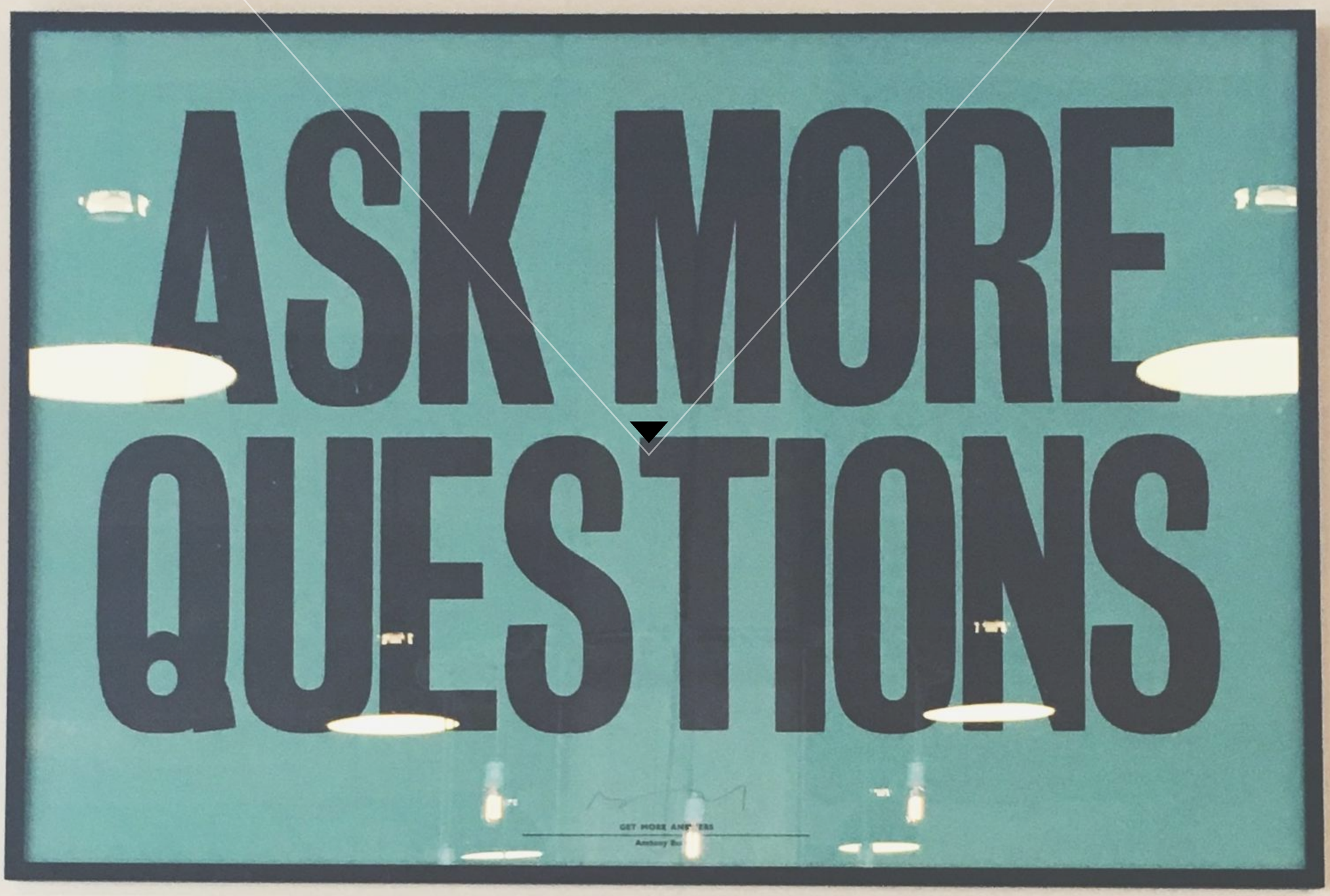
2 NO DUPLICATION

3 EXPRESSIVE

4 MINIMAL CLASSES AND METHODS



END


ASK MORE
QUESTIONS


Thank You