Clean Code style and concept

Wednesday, March 6, 2019 11:43 PM

Uncle Bob Martin: **Clean Code Book**

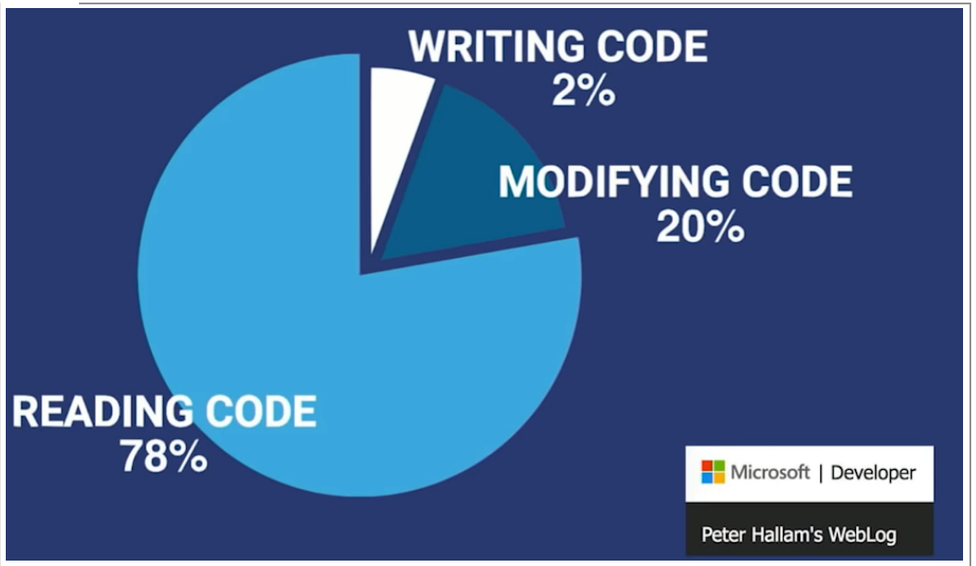
```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
```

How long it will take to you write down the instructions that a robot could tie your shoes in five seconds?

Ans: This is the problem we face as a **s/w developers**. We can visualize the problem and the problem is fundamentally a simple problem, but we have to break it down into steps that are impossibly small that is our job. And it takes a long time and it's very difficult. *The next time you are asked to estimate how long something is going to take keep that one in mind* ***" Don't look at task itself, the task itself might be simple, look at the cost of the writing down the instructions that's what we really have to do"***

Think about it how long it will take, then notice it will take 3-times longer, it is fundamental truth of software everything takes 3-times longer than you think of it will, even when you have taken this into account.
If you write Test Cases it will shorten your time, if you follow the practice of Testing that will make the whole process shorter, because you spent no time debugging. And how much time does debugging take? If you follow the **Testing** discipline there will be no bugs at least no obvious bugs, most of the bugs will be gone.

```
  if (setup != null) {
    WikiPagePath setupPath =
      wikiPage.getPageCrawler().getFullPath(setup);
    String setupPathName = PathParser.render(setupPath);
    buffer.append("!include -setup .")
          .append(setupPathName)
          .append("\n");
  }
}
buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
  WikiPage teardown =
    PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
  if (teardown != null) {
    WikiPagePath tearDownPath =
      wikiPage.getPageCrawler().getFullPath(teardown);
    String tearDownPathName = PathParser.render(tearDownPath);
    buffer.append("\n")
          .append("!include -teardown .")
          .append(tearDownPathName)
          .append("\n");
  }
  if (includeSuiteSetup) {
    WikiPage suiteTeardown =
      PageCrawlerImpl.getInheritedPage(
              SuiteResponder.SUITE_TEARDOWN_NAME,
              wikiPage
      );
```



```
    if (suiteTeardown != null) {
      WikiPagePath pagePath =
        suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
      String pagePathName = PathParser.render(pagePath);
      buffer.append("!include -teardown .")
            .append(pagePathName)
            .append("\n");
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

The Purpose of Formatting

First of all, let's be clear. Code formatting is important. It is too important to ignore and it is too important to treat religiously. Code formatting is about communication, and communication is the professional developer's first order of business. Perhaps you thought that "getting it working" was the first order of business for a professional developer. I hope by now, however, that this book has disabused you of that idea. The functionality that you create today has a good chance of changing in the next release, but the **readability** of your code will have a profound effect on all the changes that will ever be made. The **coding style and readability** set precedents that continue to affect **maintainability and extensibility** long after the original code has been changed beyond recognition. Your **style** and **discipline survives**, even though your **code** does not.

A good rule of thumb for a Function is that "every line in that function stays at
same level of abstraction (SLAP) & that level is one below the name of function" so
the name of function is highest level of concept in that function .

However, with just a few simple method extractions, some renaming, and a little restructuring, I was able to capture the intent of the function in the nine lines of Listing 3-2. See whether you can understand that in the next 3 minutes.

REFACTORED FUNCTION

```
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();
  }
```

Method Names

Methods should have verb or verb phrase names like `postPayment`, `deletePage`, or `save`. Accessors, mutators, and predicates should be named for their value and prefixed with `get`, `set`, and `is` according to the javabean standard.[4]

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

mutator, value, accessor, value, predicate

Small!

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that. This is not an assertion that I can justify. I can't provide any references to research that shows that very small functions are better. What I can tell you is that for nearly four decades I have written functions of all different sizes. I've written several nasty 3,000-line abominations. I've written scads of functions in the 100 to 300 line range. And I've written functions that were 20 to 30 lines long. What this experience has taught me, through long trial and error, is that functions should be very small.

In the **eighties** we used to say that a function should be no bigger than a screen-full. Of course we said that at a time when VT100 screens were 24 lines by 80 columns, and our editors used 4 lines for administrative purposes. Nowadays with a cranked-down font and a nice big monitor, you can fit 150 characters on a line and a 100 lines or more on a screen. Lines should not be 150 characters long. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.

Blocks and Indenting

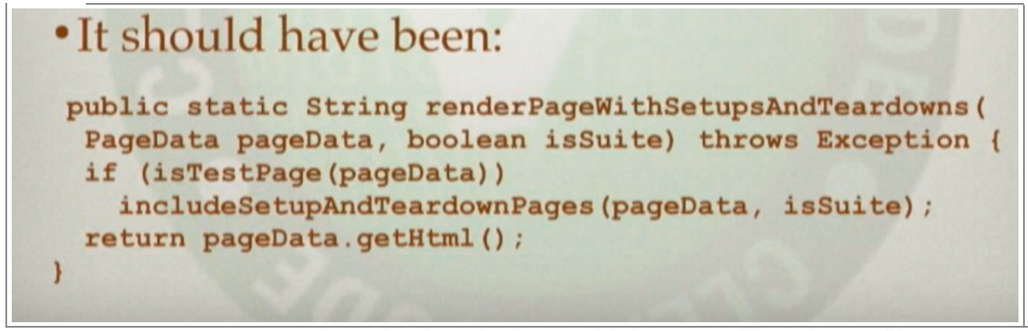
This implies that the blocks within if statements, else statements, while statements, and so on should be one line long. Probably that line should be a function call. Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name. This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This, of course, makes the functions easier to read and understand.

The following advice has appeared in one form or another for 30 years or more. **FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY.**
So, another way to know that a function is doing more than "one thing" is if you can extract another function from it with a name that is not merely a restatement of its implementation [G34].

Q: What should be length of a function?

Ans: Smaller than a Screenful, this screenful is old idea

- Functions should not be 100 lines long
- Functions should hardly ever be 20 lines long. This is also not good
- Indeed the refactored function was too long
- ***SLAP (Single Level of Abstraction) this is what desired in a function***
  - If a function contains code and you can extract another function from it then very clearly that original function did more than one thing, we should extract code from a function until I cannot extract anymore,



To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns. **To** include the setups, we include the suite setup if this is a suite, then we include the regular setup. **To** include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page. To search the parent. . .
It turns out to be very difficult for programmers to learn to follow this rule and write functions that stay at a single level of abstraction. But learning this trick is also very important. It is the key to keeping functions short and making sure they do "one thing." Making the code read like a top-down set of TO paragraphs is an effective technique for keeping the abstraction level consistent.

It takes no effort to understand this **function(renderPageWithSetupAndTeardowns())**, if you want to dig deep then navigate into the incoming functions and you are done. You might be thinking this is insane if I did that I will drown in sea of little tiny functions that have no clue, you will in fact know what these functions are. **You will not be hiding the high level structure of the system you will in fact be exposing the high level structure of the system because the high level function will have nothing in it but high level code.** Think about extracting functions you have to invent names for them, and those names tend to get longer and longer the more detailed those functions get. As you are extracting initially the functions you have extract fairly short names because they are still relatively general but then when you extract more and more those functions get really precise. They are so precise that they can only be called from one place and such functions have long sentence like names and those long sentence like names fit inside the parenthesis of your **If() statements** and **while() Loops**. And make your code read like a sentence like structure.

Reading code from top to bottom

- We want the code to read like a top-down narrative.
- We want every function to be followed by those at the next level of abstraction,
  - We can read the program, descending one level of abstraction at a time.
- We want to read the program as if it were a set of two paragraphs,
  - each of which describes the current level of abstraction
  - and references subsequent TO paragraphs at the next level down.

Names and Design

- Choosing descriptive names will clarify the design of the module in your mind,
  - and help you to improve it.
- Hunting for a good name often results in a favorable restructuring of the code.
- In software development we name all the time. We name our **variables, functions, classes, packages & components**. That's why it's necessary to do it well. Choosing a good name takes time but it helps in code readability & keeping logic evident & explicit.

In modern languages we have much richer type systems, and the compilers remember and enforce the types. What's more, there is a trend toward smaller classes and shorter functions so that people can usually see the point of declaration of each variable they're using.

- Use Searchable variable Names
  - Use solution domain names
  - Use problem domain names
- Avoid Encodings

How long name of a variable should it be?

Ans: "**The length of a variable name should be proportional to the Scope in which that variable is active" e.g.**

For(int i = 0; i < 10; i++){
Cout<< i ;
} the scope of **'i'** is inside the body of **for loop** so a single character variable **'i'** is perfectly fine. If this **for loop** was larger of 10 lines then you should give variable "i" slightly longer name,

1. If this variable is a **local variable of a function** which spans 20 lines then you need a **longer name**
    a. My personal preference is that **single-letter names** can ONLY be used as local variables inside **short methods**. The length of a name should correspond to the size of its scope.
2. If this variable is a **member variable** then you still need little **longer name**
3. If this variable is a **global variable** then you need a very **long name**

- Hungarian Notation
- Member Prefixes
    - You also don't need to prefix member variables with m_ anymore. Your classes and functions should be small enough that you don't need them. And you should be using an editing environment that highlights or colorizes members to make them distinct.
- **Interfaces** and **Implementations**
- Avoid Mental Mapping
    - One difference between a smart programmer and a professional programmer is that the professional understands that **clarity is king**. Professionals use their powers for good and write code that others can understand.
- Class Names
- Method Names
- Pick One Word per Concept

What is the Rule for Function name, how long a function name should be ?

Ans: Here the rule is exactly the opposite
"**The larger the Scope of Function the shorter we want that name** " because very large scope must be very general and so that's why we don't want very long specific name we want short name

- We want functions at high level broad scope to have very short name but as we go down lower in the scope in hierarchy the functions name get longer and longer. So
    - Member functions name little bit longer
    - Private Functions names little bit longer, and as you keep extracting, those functions names keep getting longer and longer as their scope keep getting smaller and smaller

What about Class Names and Modules Names?

Ans : they follow the same rule as **Functions**

- High level modules short name
- Low level modules long name

How many Arguments should a function have?

Ans:

- The ideal number of arguments for a function is **zero** (niladic).
- Next comes **one** (**monadic**),
- Followed closely by **two** (dyadic).
- **Three** arguments (**triadic**) should be avoided where possible.
- More than **three** (**polyadic**) requires very special justification, & then shouldn't be used anyway.

Arguments are hard. They take a lot of conceptual power. Arguments are even harder from a testing point of view. Imagine the difficulty of writing all the test cases to ensure that all the various combinations of arguments work properly. If there are no arguments, this is trivial. If there's one argument, it's not too hard. With two arguments the problem gets a bit more challenging. With more than two arguments, testing every combination of appropriate values can be daunting.

Function Arguments

Output arguments are harder to understand than input arguments. When we read a function, we are used to the idea of information going into the function through arguments and out through the return value. We don't usually expect information to be going out through the arguments. So output arguments often cause us to do a double-take. One input argument is the next best thing to no arguments. SetupTeardownIncluder.render(pageData) is pretty easy to understand. Clearly we are going to render the data in the pageData object.

Common Monadic Forms

There are two very common reasons to pass a single argument into a function. You may be asking a question about that argument, as in boolean fileExists("MyFile"). Or you may be operating on that argument, transforming it into something else and returning it. For example, InputStream fileOpen("MyFile") transforms a file name String into an InputStream return value. These two uses are what readers expect when they see a function. You should choose names that make the distinction clear, and always use the two forms in a consistent context. (See Command Query Separation below.)

Structured Programming

Some programmers follow Edsger Dijkstra's rules of structured programming. Dijkstra said that every **function**, and every block within a **function**, should have one entry and one exit. Following these rules means that there should only be one **return** statement in a function, no **break** or **continue** statements in a **loop**, and never, ever, any **goto** statements.

Conclusion

While we are sympathetic to the goals and disciplines of structured programming, those rules serve little benefit when functions are very small. It is only in larger functions that such rules provide significant benefit. So if you keep your functions small, then the occasional multiple **return**, **break**, or **continue** statement does no harm and can sometimes even be more expressive than the single-entry, single-exit rule. On the other hand, **goto** only makes sense in large functions, so it should be avoided.

How Do You Write Functions Like This?

Writing software is like any other kind of writing. When you write a paper or an article, you get your thoughts down first, then you massage it until it reads well. The first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read. When I write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code. But I also have a suite of unit tests that cover every one of those clumsy lines of code. So then I massage and refine that code, splitting out functions, changing names, eliminating duplication. I shrink the methods and reorder them. Sometimes I break out whole classes, all the while keeping the tests passing. In the end, I wind up with functions that follow the rules I've laid down in this chapter. I don't write them that way to start. I don't think anyone could.

Conclusion

Every system is built from a domain-specific language designed by the programmers to describe that system. Functions are the verbs of that language, and classes are the nouns. This is not some throwback to the hideous old notion that the nouns and verbs in a requirements document are the first guess of the classes and functions of a system. Rather, this is a much older truth. The art of programming is, and has always been, the art of language design. Master programmers think of systems as stories to be told rather than programs to be written. They use the facilities of their chosen programming language to construct a much richer and more expressive language that can be used to tell that story. Part of that domain-specific language is the hierarchy of functions that describe all the actions that take place within that system. In an artful act of recursion those actions are written to use the very domain-specific language they define to tell their own small part of the story. This chapter has been about the mechanics of writing functions well. If you follow the rules herein, your functions will be short, well named, and nicely organized. But never forget that your real goal is to tell the story of the system, and that the functions you write need to fit cleanly together into a clear and precise language to help you with that telling

Comments

Javadocs in Public APIs

There is nothing quite so helpful and satisfying as a well-described public API. The javadocs for the standard Java library are a case in point. It would be difficult, at best, to write Java programs without them. If you are writing a public API, then you should certainly write good javadocs for it. But keep in mind the rest of the advice in this chapter. Javadocs can be just as misleading, nonlocal, and dishonest as any other kind of comment.

Mandated Comments

It is just plain silly to have a rule that says that every function must have a javadoc, or every variable must have a comment. Comments like this just clutter up the code, propagate lies, and lend to general confusion and disorganization. For example, required javadocs for every function lead to abominations such as Listing 4-3. This clutter adds nothing and serves only to obfuscate the code and create the potential for lies and misdirection.

Journal Comments

Sometimes people add a comment to the start of a module every time they edit it. These comments accumulate as a kind of journal, or log, of every change that has ever been made. I have seen some modules with dozens of pages of these run-on journal entries.

```
    * Changes (from 11-Oct-2001)
    * --------------------------
    * 11-Oct-2001 : Re-organised the class and moved it to new package
    *               com.jrefinery.date (DG);
    * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
    *               class (DG);
    * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
    *               class is gone (DG);  Changed getPreviousDayOfWeek(),
    *               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
    *               bugs (DG);
    * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
    * 29-May-2002 : Moved the month constants into a separate interface
    *               (MonthConstants) (DG);
    * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
    * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
    * 13-Mar-2003 : Implemented Serializable (DG);
    * 29-May-2003 : Fixed bug in addMonths method (DG);
    * 04-Sep-2003 : Implemented Comparable.  Updated the isInRange javadocs (DG);
    * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

Long ago there was a good reason to create and maintain these log entries at the start of every module. We didn't have source code control systems that did it for us. Nowadays, however, these long journals are just more clutter to obfuscate the **module**. They should be completely removed.

Consider this from apache commons:

```
this.bytePos = writeBytes(pngIdBytes, 0);
```

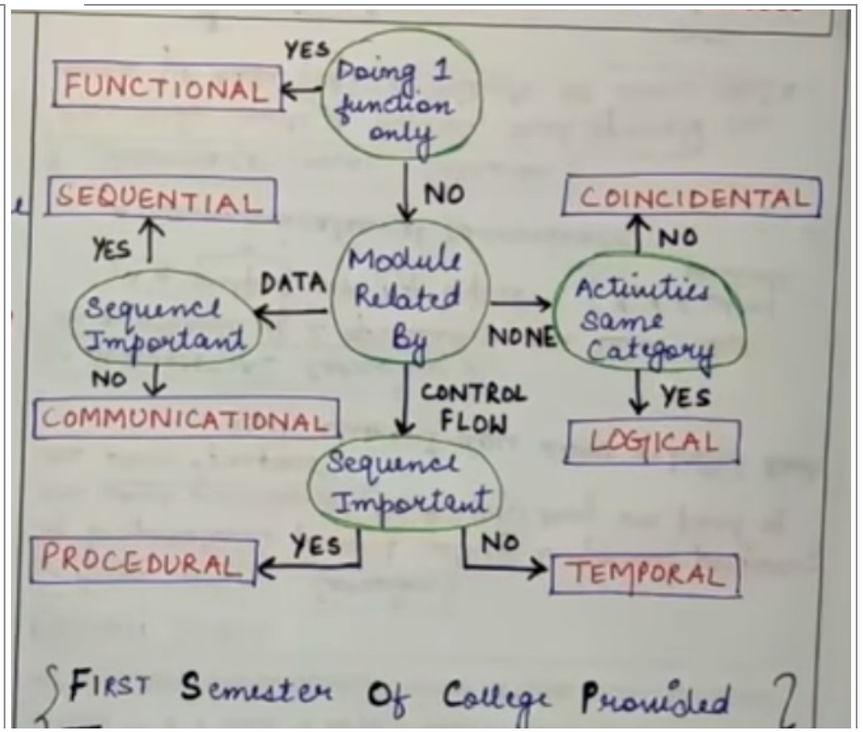
```
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
     writeEnd();
     this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}

  else {
       this.pngBytes = null;
  }
  return this.pngBytes;
```

Why are those two lines of code commented? Are they important? Were they left as reminders for some imminent change? Or are they just cruft that someone commented-out years ago and has simply not bothered to clean up. There was a time, back in the sixties, when commenting-out code might have been useful. But we've had good source code control systems for a very long time now. Those systems will remember the code for us. We don't have to comment it out any more. Just delete the code. We won't lose it. Promise.

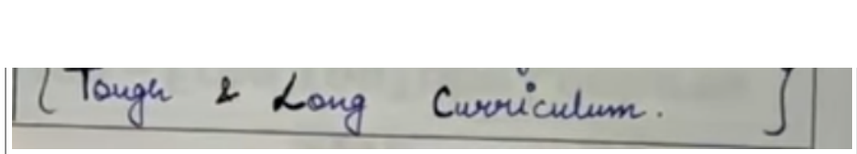
Dependent Functions. If one function calls another, they should be vertically close, and the caller should be above the callee, if at all possible. This gives the program a natural flow. If the convention is followed reliably, readers will be able to trust that function definitions will follow shortly after their use. Consider, for example, the snippet from FitNesse in Listing 5-5. Notice how the topmost function calls those below it and how they in turn call those below them. This makes it easy to find the called functions and greatly enhances the readability of the whole module.

WikiPageResponder.java

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
  loadPage(pageName, context);
  if (page == null)
    return notFoundResponse(context, request);
  else
    return makePageResponse(context);
}

private String getPageNameOrDefault(Request request, String defaultPageName)
{
  String pageName = request.getResource();
  if (StringUtil.isBlank(pageName))
    pageName = defaultPageName;

  return pageName;
}

protected void loadPage(String resource, FitNesseContext context)
  throws Exception {
  WikiPagePath path = PathParser.parse(resource);
  crawler = context.root.getPageCrawler();
  crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
  page = crawler.getPage(context.root, path);
  if (page != null)
    pageData = page.getData();
}

private Response notFoundResponse(FitNesseContext context, Request request)
  throws Exception {
  return new NotFoundResponder().makeResponse(context, request);
```



```
  }

  private SimpleResponse makePageResponse(FitNesseContext context)
    throws Exception {
    pageTitle = PathParser.render(crawler.getFullPath(page));
    String html = makeHtml(context);

    SimpleResponse response = new SimpleResponse();
    response.setMaxAge(0);
    response.setContent(html);
    return response;
  }
...
```



As an aside, this snippet provides a nice example of keeping constants at the appropriate level [G35]. The "FrontPage" constant could have been buried in the getPageNameOrDefault function, but that would have hidden a well-known and expected constant in an inappropriately low-level function. It was better to pass that constant down from the place where it makes sense to know it to the place that actually uses it.

Vertical Ordering

In general we want function call dependencies to point in the downward direction. That is, a function that is called should be below a function that does the calling.2 This creates a nice flow down the source code module from high level to low level. As in newspaper articles, we expect the most important concepts to come first, and we expect them to be expressed with the least amount of polluting detail. We expect the low-level details to come last. This allows us to skim source files, getting the gist from the first few functions, without having to immerse ourselves in the details.

Objects and Data Structures

There is a reason that we keep our variables private. We don't want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?

Data Abstraction

Consider the difference between Listing 6-1 and Listing 6-2. Both represent the data of a point on the Cartesian plane. And yet one exposes its implementation and the other completely hides it.

Listing 6-1
Concrete Point

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

Listing 6-2
Abstract Point

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

The beautiful thing about Listing 6-2 is that there is no way you can tell whether the implementation is in rectangular or polar coordinates. It might be neither! And yet the interface still unmistakably represents a data structure. But it represents more than just a data structure. The methods enforce an access policy. You can read the individual coordinates independently, but you must set the coordinates together as an atomic operation. Listing 6-1, on the other hand, is very clearly implemented in rectangular coordinates, and it forces us to manipulate those coordinates independently. This exposes implementation. Indeed, it would expose implementation even if the variables were private and we were using single variable getters and setters. Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions! A class does not simply push its variables out through getters and setters. Rather it exposes abstract interfaces that allow its users to manipulate the essence of the data, without having to know its implementation. Consider Listing 6-3 and Listing 6-4. The first uses concrete terms to communicate the fuel level of a vehicle, whereas the second does so with the abstraction of percentage. In the concrete case you can be pretty sure that these are just accessors of variables. In the abstract case you have no clue at all about the form of the data.

Listing 6-3
Concrete Vehicle

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

Listing 6-4
Abstract Vehicle

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

In both of the above cases the second option is preferable. We do not want to expose the details of our data. Rather we want to express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters. Serious thought needs to be put into the best way to represent the data that an object contains. **The worst option is to blithely add getters and setters.**

Data/Object Anti-Symmetry

These two examples show the difference **between objects and data structures**. **Objects** hide their **data** behind **abstractions** and **expose functions** that operate on that **data**. **Data structure** expose their **data** and **have no meaningful functions**. Go back and read that again. Notice the complimentary nature of the two definitions. They are virtual opposites. This difference may seem trivial, but it has far-reaching implications. Consider, for example, the procedural shape example in Listing 6-5. The Geometry class operates on the three shape classes. The shape classes are simple data structures without any behavior. All the behavior is in the Geometry class.

Listing 6-5

Procedural Shape

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Object-oriented programmers might wrinkle their noses at this and complain that it is procedural—and they'd be right. But the sneer may not be warranted. Consider what would happen if a perimeter() function were added to Geometry. The shape classes (square, rectangle, circle) would be unaffected! Any other classes that depended upon the shapes would also be unaffected! On the other hand, if I add a new shape, I must change all the functions in Geometry to deal with it. Again, read that over. Notice that the two conditions are diametrically opposed. Now consider the **object-oriented solution in Listing 6-6.** Here the area() method is **polymorphic**. No Geometry class is necessary. So if I add a new shape, none of the existing functions are affected, but if I add a new function all of the shapes must be changed!

Listing 6-6
Polymorphic Shapes

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}
```

There are ways around this that are well known to **experienced object-oriented designers**: **VISITOR**, or **dual-dispatch**, for example. But these techniques carry costs of their own and generally return the structure to that of a procedural program.

Listing 6-6 (continued)
Polymorphic Shapes

```
public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

Again, we see the complimentary nature of these two definitions; they are virtual opposites! This exposes the fundamental dichotomy between objects and data structures:

Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.

The complement is also true:

***Procedural code** makes it **hard** to add new **data structures** because all the **functions** must change.*
***OO code** makes it **hard** to add new **functions** because all the **classes** must change.*

So, the things that are hard for OO are easy for procedures, and the things that are hard for procedures are easy for OO! In any complex system there are going to be times when we want to add new data types rather than new functions. For these cases objects and OO are most appropriate. On the other hand, there will also be times when we'll want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate. ***Mature programmers know that the idea that everything is an object is a myth. Sometimes you really do want simple data structures with procedures operating on them.***

The Law of Demeter

There is a well-known heuristic called the Law of Demeter2 that says a module should not know about the innards of the objects it manipulates. As we saw in the last section, objects hide their data and expose operations. This means that an object should not expose its internal structure through **accessors** (getters) because to do so is to expose,

- The ideal number of arguments for a function is **zero** (niladic).
- Next comes **one** (**monadic**),

Doing this will enhance the **SRP**

rather than to hide, its internal structure. More precisely, the Law of Demeter says that a method f of a class C should only call the methods of these:

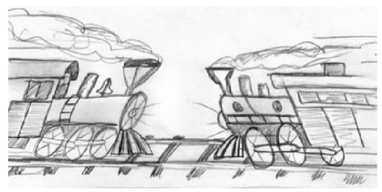
1. C
2. An object created by f
3. An object passed as an argument to f
4. An object held in an instance variable of C

The method should not invoke methods on objects that are returned by any of the allowed functions. In other words, talk to friends, not to strangers.

The following code3 appears to violate the Law of Demeter (among other things) because it calls the getScratchDir() function on the return value of getOptions() and then calls getAbsolutePath() on the return value of getScratchDir().

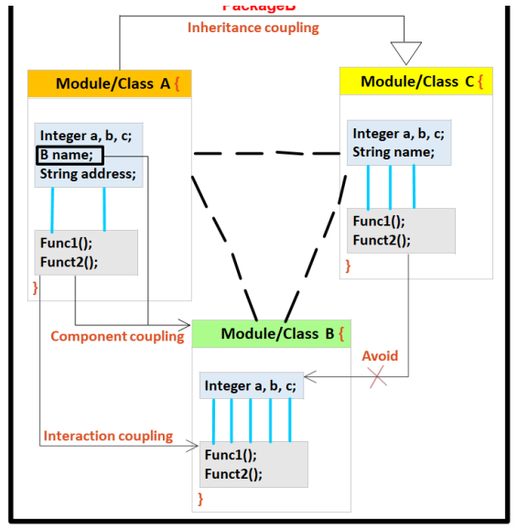
```
final String outputDir =
ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Train Wrecks



This kind of code is often called a train wreck because it look like a bunch of coupled train cars. Chains of calls like this are generally considered to be sloppy style and should be avoided [G36]. It is usually best to split them up as follows:

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



Are these (above) two snippets of code violations of the Law of Demeter? Certainly the containing module knows that the ctxt object contains options, which contain a scratch directory, which has an absolute path. That's a lot of knowledge for one function to know. The calling function knows how to navigate through a lot of different objects.

Whether this is a violation of Demeter depends on whether or not ctxt, Options, and ScratchDir are objects or data structures. If they are objects, then their internal structure should be hidden rather than exposed, and so knowledge of their innards is a clear violation of the Law of Demeter. On the other hand, if ctxt, Options, and ScratchDir are just data structures with no behavior, then they naturally expose their internal structure, and so Demeter does not apply.

The use of accessor functions confuses the issue. If the code had been written as follows, then we probably wouldn't be asking about Demeter violations.

final String outputDir = ctxt.options.scratchDir.absolutePath;

This issue would be a lot less confusing if data structures simply had public variables and no functions, whereas objects had private variables and public functions. However, there are frameworks and standards (e.g., "beans") that demand that even simple data structures have accessors and mutators.

Hybrids

This confusion sometimes leads to unfortunate hybrid structures that are half object and half data structure. They have functions that do significant things, and they also have either public variables or public accessors and mutators that, for all intents and purposes, make the private variables public, tempting other external functions to use those variables the way a procedural program would use a data structure.

Such hybrids make it hard to add new functions but also make it hard to add new data structures. They are the worst of both worlds. Avoid creating them. They are indicative of a muddled design whose authors are unsure of—or worse, ignorant of—whether they need protection from **functions** or **types(int, float, double, char, class, etc.)**.

Hiding Structure

What if ctxt, options, and scratchDir are objects with real behavior? Then, because objects are supposed to hide their internal structure, we should not be able to navigate through them. How then would we get the absolute path of the scratch directory?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

or

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

The first option could lead to an explosion of methods in the ctxt object. The second presumes that getScratchDirectoryOption() returns a data structure, not an object. Neither option feels good.

If ctxt is an object, we should be telling it to do something; we should not be asking it about its internals. So why did we want the absolute path of the scratch directory? What were we going to do with it? Consider this code from (many lines farther down in) the same module:

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

The admixture of different levels of detail [G34][G6] is a bit troubling. Dots, slashes, file extensions, and File objects should not be so carelessly mixed together, and mixed with the enclosing code. Ignoring that, however, we see that the intent of getting the absolute path of the scratch directory was to create a scratch file of a given name.

So, what if we told the ctxt object to do this?

BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);

That seems like a reasonable thing for an object to do! This allows ctxt to hide its internals and prevents the current function from having to violate the Law of Demeter by navigating through objects it shouldn't know about.

Data Transfer Objects

The quintessential (ideal, typical, exemplary, standard) form of a data structure is a class with **public** variables and no functions. This is sometimes called a data transfer object, or DTO. DTOs are very useful structures, especially when communicating with databases or parsing messages from sockets, and so on. They often become the first in a series of translation stages that convert raw data in a database into objects in the application code. Somewhat more common is the "bean" form shown in Listing 6-7. Beans have private variables manipulated by getters and setters. The quasi-encapsulation of beans seems to make some OO purists feel better but usually provides no other benefit.

Listing 6-7
address.java

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }
```

Listing 6-7 (continued)
address.java

```
  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

Active Record

Active Records are special forms of DTOs. They are data structures with public (or bean accessed) variables; but they typically have navigational methods like save and find. Typically these Active Records are direct translations from database tables, or other data sources.

Unfortunately we often find that developers try to treat these data structures as though they were objects by putting business rule methods in them. This is awkward because it creates a hybrid between a data structure and an object.
The solution, of course, is to treat the Active Record as a data structure and to create separate objects that contain the business rules and that hide their internal data (which are probably just instances of the Active Record).

Conclusion

Objects expose behavior and hide data. This makes it easy to add new kinds of objects without changing existing behaviors. It also makes it hard to add new behaviors to existing objects. Data structures expose data and have no significant behavior. This makes it easy to add new behaviors to existing data structures but makes it hard to add new data structures to existing functions.
In any given system we will sometimes want the flexibility to add new data types, and so we prefer objects for that part of the system. Other times we will want the flexibility to add new behaviors, and so in that part of the system we prefer data types and procedures. Good software developers understand these issues without prejudice and choose the approach that is best for the job at hand.

Error Handling:

It might seem odd to have a section about error handling in a book about clean code. Error handling is just one of those things that we all have to do when we program. Input can be abnormal and devices can fail. In short, 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.
The connection to clean code, however, should be clear. Many code bases are completely dominated by error handling. When I say dominated, I don't mean that error handling is all that they do. I mean that it is nearly impossible to see what the code does because of all of the scattered error handling. Error handling is important, but if it **obscures logic**, it's wrong.
In this chapter I'll outline a number of techniques and considerations that you can use to write code that is both clean and robust—code that handles errors with grace and style.

Use Exceptions Rather Than Return Codes

Back in the distant past there were many languages that didn't have exceptions. In those languages the techniques for handling and reporting errors were limited. You either set an error flag or returned an error code that the caller could check. The code in **Listing 7-1** illustrates these approaches. **-------->>------->>---------->>---------->>---------->>**

Listing 7-1
DeviceController.java

```
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
    // Check the state of the device
    if (handle != DeviceHandle.INVALID) {
```

The problem with these approaches is that they clutter the caller. The caller must check for errors immediately after the call. Unfortunately, it's easy to forget. For this reason it is better to throw an exception when you encounter an error. The calling code is cleaner. Its logic is not obscured by error handling. **Listing 7-2** shows the code after we've chosen to throw exceptions in methods that can detect errors

Listing 7-2
DeviceController.java (with exceptions)

```
public class DeviceController {
  ...

  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }
```

Listing 7-2 (continued)
DeviceController.java (with exceptions)

```
  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
  }

  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
  }

  ...
}
```

```
    // Save the device status to the record field
    retrieveDeviceRecord(handle);
    // If not suspended, shut down
    if (record.getStatus() != DEVICE_SUSPENDED) {
      pauseDevice(handle);
      clearDeviceWorkQueue(handle);
      closeDevice(handle);
    } else {
      logger.log("Device suspended.  Unable to shut down");
    }
  } else {
    logger.log("Invalid handle for: " + DEV1.toString());
  }
}
...
}
```

Notice how much cleaner it is. This isn't just a matter of aesthetics. The code is better because two concerns that were tangled, the algorithm for device shutdown and error handling, are now separated. You can look at each of those concerns and understand them independently.

Clean Boundaries

Interesting things happen at boundaries. Change is one of those things. Good software designs accommodate change without huge investments and rework. When we use code that is out of our control, special care must be taken to protect our investment and make sure future change is not too costly. Code at the boundaries needs clear separation and tests that define expectations. We should avoid letting too much of our code know about the third-party particulars. It's better to depend on something you control than on something you don't control, lest it end up controlling you. We manage third-party boundaries by having very few places in the code that refer to them. We may wrap them as we did with Map, or we may use an **ADAPTER** to convert from our perfect interface to the provided interface. Either way our code speaks to us better, promotes internally consistent usage across the boundary, and has fewer maintenance points when the third-party code changes.

The Three Laws of TDD

By now everyone knows that TDD asks us to write unit tests first, before we write production code. But that rule is just the tip of the iceberg. Consider the following three laws:1

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

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

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

F.I.R.S.T

Clean tests follow five other rules that form the above acronym:

Fast Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently. If you don't run them frequently, you won't find problems early enough to fix them easily. You won't feel as free to clean up the code. Eventually the code will begin to rot.

Independent Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. When tests depend on each other, then the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.

Repeatable Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. If your tests aren't repeatable in any environment, then you'll always have an excuse for why they fail. You'll also find yourself unable to run the tests when the environment isn't available.

Self-Validating The tests should have a Boolean output. Either they pass or fail. You should not have to read through a log file to tell whether the tests pass. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren't self-validating, then failure can become subjective and running the tests can require a long manual evaluation.

Timely The tests need to be written in a timely fashion. Unit tests should be written just before the production code that makes them pass. If you write tests after the production code, then you may find the production code to be hard to test. You may decide that some production code is too hard to test. You may not design the production code to be testable.

Conclusion

We have barely scratched the surface of this topic. Indeed, I think an entire book could be written about clean tests. Tests are as important to the health of a project as the production code is. Perhaps they are even more important, because tests preserve and enhance the flexibility, maintainability, and reusability of the production code. So keep your tests constantly clean. Work to make them expressive and succinct. Invent testing APIs that act as domain-specific language that helps you write the tests. If you let the tests rot, then your code will rot too. Keep your tests clean.

What to Test: The Right-BICEP

It can be hard to look at a method or a class and anticipate all the bugs that might be lurking in there. With experience, you develop a feel for what's likely to break and learn to concentrate on testing those areas first. Until then, uncovering possible failure modes can be frustrating. End users are quite adept at finding our bugs, but that's embarrassing and damaging to our careers! What we need are guidelines to help us understand what's important to test.
Your Right-BICEP provides you with the strength needed to ask the right questions about what to test:

Right Are the results right?

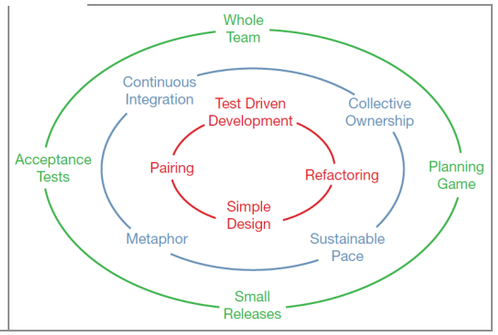
B Are all the **boundary conditions** correct?

An obvious happy path through the code might not hit any boundary conditions in the code—scenarios that involve the edges of the input domain. Many of the defects you'll code in your career will involve these corner cases, so you'll want to cover them with tests.

Remembering Boundary Conditions with CORRECT

The **CORRECT** acronym gives you a way to remember potential boundary conditions. For each of these items, consider whether or not similar conditions can exist in the method that you want to test, and what might happen if these conditions are violated:

- **Conformance**—Does the value conform to an expected format?
- **Ordering**—Is the set of values ordered or unordered as appropriate?
- **Range**—Is the value within reasonable minimum and maximum values?
- **Reference**—Does the code reference anything external that isn't under direct control of the code itself?
- **Existence**—Does the value exist (is it non-null, nonzero, present in a set, and so on)?
- Cardinality—Are there exactly enough values?
- **Time** (absolute and relative)—Is everything happening in order? At the right time? In time?



I Can you check inverse relationships?

C Can you cross-check results using other means?

E Can you force error conditions to happen?

P Are performance characteristics within bounds?

Unit test maintenance costs increase as your system's design/code quality decreases.

The Bigger Unit-Testing Picture

You can take your unit-testing skills to the next level by learning about the practice of **test-driven development (TDD).** We'll rewrite some familiar code using **TDD** so you can experience it firsthand. You'll then learn how to face some of the tougher challenges in unit testing. **Finally, you'll learn about unit-testing standards, pair programming, continuous integration (CI), and code coverage to understand how unit testing fits into the larger scope of a project team.**

Convergence with Continuous Integration

"It works on my machine!" cries Pat. "Must be something wrong on your machine," he says to Dale.
Unit tests aren't going to fix all such problems, but they are a standard of sorts: any changes to the code can't break the collective set of tests; otherwise the standards—the tests—have been violated.
To be able to view the **unit tests** as a **team-wide standard** requires a **shared repository**, of course. Developers check code out from the **repository** (or create local branches, depending on your worldview), make changes, test locally, then check the code back into the **shared repository** (also known as **integrating the code**).

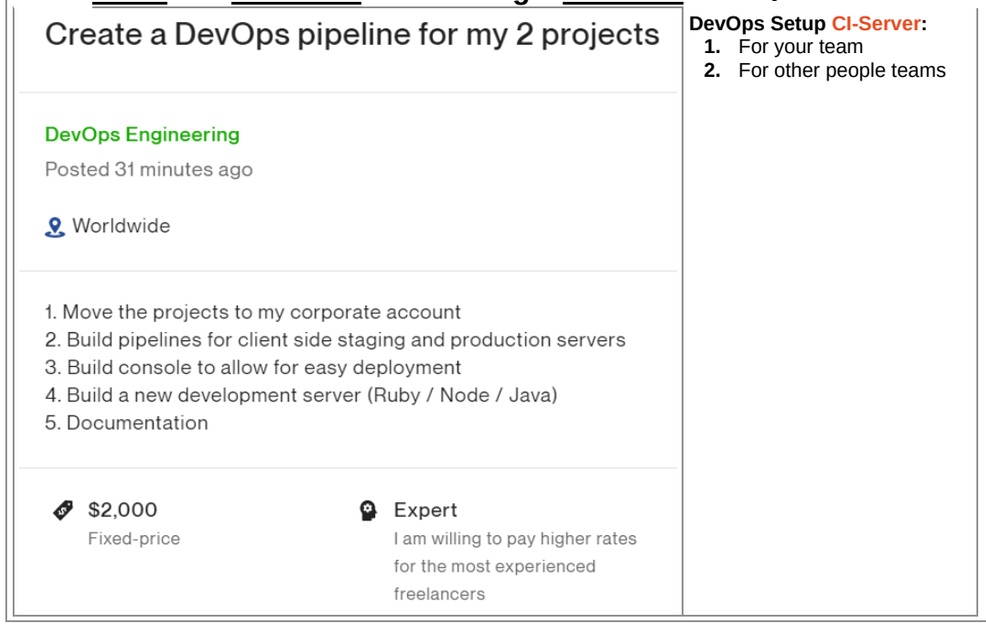
For the **CI server (continuous Integration server)** to provide any value, your **build** must now include the running of your unit tests. Because the CI server build process works on the code of record in your source repository, it demonstrates the overall health of your system. Not "my changes work on my machine," or "your changes work on your machine," but "our code works on one of our **golden servers**."

The CI server helps support healthy peer pressure against allowing bad code. Developers begin to habituate themselves to running their unit tests before check-in. No one wants to be recognized as the person wasting their teammates' time by causing the CI build process to fail.
Installing and configuring a typical CI server requires perhaps a day or two. This is time well spent. We consider use of a CI server to be foundational.

A CI server is a minimum for building a modern development team.

You'll find numerous **CI tools** that work well with **Java**. Some CI servers are free, some are open source, some are hosted, and some are licensed. Some of the more widely used CI servers include **Hudson, Jenkins (a fork of Hudson), TeamCity, AntHill, CruiseControl, Buildbot, and Bamboo.**

A CI server is a minimum for building a modern development team.



DevOps Setup CI-Server:

1. For your team
2. For other people teams

Classes

Your classes design & structure must follow SOLID engineering principles.

I was thinking we should not touch code which is not well structured and standardized , why, because it is a mess But then I realize it is an opportunity for us to earn more money in such cases you can ask for more money and if you can find Documentation or up to date Use cases and flow chart of algorithms(code) then it is worth.