



andrewhessler and **johnousterhout** missing word

703391c · yesterday



2266 lines (1700 loc) · 111 KB

Preview

Code

Blame



Raw



(This document is the result of a series of discussions, some online and some in person, held between Robert "Uncle Bob" Martin and John Ousterhout between September 2024 and February 2025. If you would like to comment on anything in this discussion, we recommend that you do so on the [Google group associated with APOSD](#))

Introductions

JOHN:

Hi (Uncle) Bob! You and I have each written books on software design. We agree on some things, but there are some pretty big differences of opinion between my recent book *A Philosophy of Software Design* (hereafter "APOSD") and your classic book *Clean Code*. Thanks for agreeing to discuss those differences here.

UB:

My pleasure John. Before we begin, let me say that I've carefully read through your book and I found it very enjoyable, and full of valuable insights. There are some things I disagree with you on, such as TDD, and Abstraction-First incrementalism, but overall I enjoyed it a lot.

JOHN:

I'd like to discuss three topics with you: method length, comments, and test-driven development. But before getting into these, let's start by comparing overall philosophies. When you hear about a new idea related to software design, how do you decide whether or not to endorse that idea?

I'll go first. For me, the fundamental goal of software design is to make it easy to understand and modify the system. I use the term "complexity" to refer to things that make it hard to understand and modify a system. The most important contributors to complexity relate to information:

- How much information must a developer have in their head in order to carry out a task?
- How accessible and obvious is the information that the developer needs?

The more information a developer needs to have, the harder it will be for them to work on the system. Things get even worse if the required information isn't obvious. The worst case is when there is a crucial piece of information hidden in some far-away piece of code that the developer has never heard of.

When I'm evaluating an idea related to software design, I ask whether it will reduce complexity. This usually means either reducing the amount of information a developer has to know, or making the required information more obvious.

Now over to you: are there general principles that you use when deciding which ideas to endorse?

UB:

I agree with your approach. A discipline or technique should make the job of programmers easier. I would add that the programmer we want to help most is not the author. The programmer whose job we want to make easier is the programmer who must read and understand the code written by others (or by themselves a week later). Programmers spend far more hours reading code than writing code, so the activity we want to ease is that of reading.

Method Length

JOHN:

Our first area of disagreement is method length. On page 34 of *Clean Code* you say "The first rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that*." Later on, you say "Functions should hardly ever be 20 lines long" and suggest that functions should be "just two, three, or four lines long". On page 35, you say "Blocks within `if` statements, `else` statements, `while` statements, and so on should be one line long. Probably that line should be a function call." I couldn't find anything in *Clean Code* to suggest that a function could ever be too short.

I agree that dividing up code into relatively small units ("modular design") is one of the most important ways to reduce the amount of information a programmer has to keep in their mind at once. The idea, of course, is to take a complex chunk of functionality and encapsulate it in a separate method with a simple interface. Developers can then harness the functionality of the method (or read code that invokes the method) without learning the details of how the method is implemented; they only need to learn its interface. The best methods are those that provide a lot of functionality but have a very simple interface: they replace a large cognitive load (reading the detailed implementation) with a much smaller cognitive load (learning the interface). I call these methods "deep".

However, like most ideas in software design, decomposition can be taken too far. As methods get smaller and smaller there is less and less benefit to further subdivision. The amount of functionality hidden behind each interface drops, while the interfaces often become more complex. I call these interfaces "shallow": they don't help much in terms of reducing what the programmer needs to know. Eventually, the point is reached where someone using the method needs to understand every aspect of its implementation. Such methods are usually pointless.

Another problem with decomposing too far is that it tends to result in *entanglement*. Two methods are entangled (or "conjoined" in APOSD terminology) if, in order to understand how one of them works internally, you also need to read the code of the other. If you've ever found yourself flipping back and forth between the implementations of two methods as you read code, that's a red flag that the methods might be entangled. Entangled methods are hard to read because the information you need to have in your head at once isn't all in the same place. Entangled methods can usually be improved by combining them so that all the code is in one place.

The advice in *Clean Code* on method length is so extreme that it encourages programmers to create teeny-tiny methods that suffer from both shallow interfaces and entanglement. Setting arbitrary numerical limits such as 2-4 lines in a method and a single line in the body of an `if` or `while` statement exacerbates this problem.

UB:

While I do strongly recommend very short functions, I don't think it's fair to say that the book sets arbitrary numerical limits. The 2-4 line functions that you referred to on page 34 were part of the *Sparkle* applet that Kent Beck and I wrote together in 1999 as an exercise for learning TDD. I thought it was remarkable that most of the functions in that applet were 2-4 lines long because it was a Swing program; and Swing programs tend to have very long methods.

As for setting limits, on page 13 I make clear that although the recommendations in the book have worked well for me and the other authors, they might not work for everyone. I claimed no final authority, nor even any absolute "rightness". They are offered for consideration.

JOHN:

I think these problems will be easiest to understand if we look at specific code examples. But before we do that, let me ask you, Bob: do you believe that it's possible for code to be over-decomposed, or is smaller always better? And, if you believe that over-decomposition is possible, how do you recognize when it has occurred?

UB:

It is certainly possible to over-decompose code. Here's an example:

```
void doSomething() {doTheThing()} // over-decomposed.
```



The strategy that I use for deciding how far to take decomposition is the old rule that a method should do "*One Thing*". If I can *meaningfully* extract one method from another, then the original method did more than one thing. "Meaningfully" means that the extracted functionality can be given a descriptive name; and that it does less than the original method.

JOHN:

Unfortunately the One Thing approach will lead to over-decomposition:

1. The term "one thing" is vague and easy to abuse. For example, if a method has two lines of code, isn't it doing two things?
2. You haven't provided any useful guardrails to prevent over-decomposition. The example you gave is too extreme to be useful, and the "can it be named" qualification doesn't help: anything can be named.
3. The One Thing approach is simply wrong in many cases. If two things are closely related, it might well make sense to implement them in a single method. For example, any thread-safe method will first have to acquire a lock, then carry out its function. These are two "things", but they belong in the same method.

UB:

Let me tackle the last thing first. You suggested that locking the thread, and performing a critical section should be together in the same method. However, I would be tempted to separate the locking from the critical section.

```
void concurrentOperation() {  
    lock()  
    criticalSection();  
    unlock()  
}
```



This decouples the critical section from the lock and allows it to be called at times when locking isn't necessary (e.g. in single thread mode) or when a lock has already been set by someone else.

Now, on to the "ease of abuse" argument. I don't consider that to be a significant concern. If statements are easy to abuse. Switch statements are easy to abuse. Assignment statements are easy to abuse. The fact that something is easy to abuse does not mean that it should be avoided or suppressed. It simply means people should take appropriate care. There will always be this thing called: *judgment*.

So when faced with this snippet of code in a larger method:

```
...  
amountOwed=0;  
totalPoints=0;  
...
```



It would be poor judgement to extract them as follows, because the extraction is not meaningful. The implementation is not more deeply detailed than the interface.

```
void clearAmountOwed() {  
    amountOwed=0;  
}  
  
void clearTotalPoints() {  
    totalPoints=0;  
}
```



However, it may be good judgement to extract them as follows because the interface is abstract, and the implementation has deeper detail.

```
void clearTotals() {  
    amountOwed=0;
```



```
totalPoints=0;  
}
```

The latter has a nice descriptive name that is abstract enough to be meaningful without being redundant. And the two lines together are strongly related so as to qualify for doing *one thing*: initialization.

JOHN:

Of course anything can be abused. But the best approaches to design encourage people to do things the right way and discourage abuse. Unfortunately, the One Thing Rule encourages abuse for the reasons I gave above.

And of course software designers will need to use judgment: it isn't possible to provide precise recipes for software design. But good judgment requires principles and guidance. The *Clean Code* arguments about decomposition, including the One Thing Rule, are one-sided. They give strong, concrete, quantitative advice about when to chop things up, with virtually no guidance for how to tell you've gone too far. All I could find is a 2-sentence example on page 36 about Listing 3-3 (which is pretty trivial), buried in the middle of exhortations to "chop, chop, chop".

One of the reasons I use the deep/shallow characterization is that it captures both sides of the tradeoff; it will tell you when a decomposition is good and also when decomposition makes things worse.

UB:

You make a good point that I don't talk much, in the book, about how to make the judgement call. Back in 2008 my concern was breaking the habit of the very large functions that were common in those early days of the web. I have been more balanced in the 2d ed.

Still, if I must err, I'd rather err on the side of decomposition. There is value in considering, and visualizing decompositions. They can always be inlined if we judge them to have gone too far.

JOHN:

Coming back to your `clearTotals` example:

- The `clearTotals` method seems to contradict the One Thing Rule: the variables `amountOwed` and `totalPoints` don't seem particularly related, so initializing them both is doing two things, no? You say that both statements are performing initialization, which makes it just one thing (initialization). Does that mean it would also be okay to have a single method that initializes two completely independent objects with nothing in common? I suspect not.

It feels like you are struggling to create a clean framework for applying the One Thing Rule; that makes me think it isn't a good rule.

- Without seeing more context I'm skeptical that the `clearTotals` method makes sense.

UB:

I hope you agree that between these two examples, the former is a bit better.

```
public String makeStatement() {  
    clearTotals();  
    return makeHeader() + makeRentalDetails() + makeFooter();  
}
```



```
public String makeStatement() {  
    amountOwed=0;  
    totalPoints=0;  
    return makeHeader() + makeRentalDetails() + makeFooter();  
}
```



JOHN:

Well, actually, no. The second example is completely clear and obvious: I don't see anything to be gained by splitting it up.

SPOCK (a.k.a UB):

Fascinating.

JOHN:

I think it will be easier to clarify our differences if we consider a nontrivial code example. Let's look at the `PrimeGenerator` class from *Clean Code*, which is Listing 10-8 on pages 145-146. This Java class generates the first N prime numbers:

```
package literatePrimes;  
  
import java.util.ArrayList;  
  
public class PrimeGenerator {  
    private static int[] primes;  
    private static ArrayList<Integer> multiplesOfPrimeFactors;  
  
    protected static int[] generate(int n) {  
        primes = new int[n];
```



```
        multiplesOfPrimeFactors = new ArrayList<Integer>();
        set2AsFirstPrime();
        checkOddNumbersForSubsequentPrimes();
        return primes;
    }

    private static void set2AsFirstPrime() {
        primes[0] = 2;
        multiplesOfPrimeFactors.add(2);
    }

    private static void checkOddNumbersForSubsequentPrimes() {
        int primeIndex = 1;
        for (int candidate = 3; primeIndex < primes.length; candidate
            if (isPrime(candidate))
                primes[primeIndex++] = candidate;
        }
    }

    private static boolean isPrime(int candidate) {
        if (isLeastRelevantMultipleOfLargerPrimeFactor(candidate)) {
            multiplesOfPrimeFactors.add(candidate);
            return false;
        }
        return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
    }

    private static boolean isLeastRelevantMultipleOfLargerPrimeFactor
        int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.si
        int leastRelevantMultiple = nextLargerPrimeFactor * nextLarge
        return candidate == leastRelevantMultiple;
    }

    private static boolean isNotMultipleOfAnyPreviousPrimeFactor(int
        for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
            if (isMultipleOfNthPrimeFactor(candidate, n))
                return false;
        }
        return true;
    }

    private static boolean isMultipleOfNthPrimeFactor(int candidate,
        return candidate == smallestOddNthMultipleNotLessThanCandidat
    }

    private static int smallestOddNthMultipleNotLessThanCandidate(int
        int multiple = multiplesOfPrimeFactors.get(n);
        while (multiple < candidate)
            multiple += 2 * primes[n];
        multiplesOfPrimeFactors.set(n, multiple);
        return multiple;
    }
```



```
    }
}
```

Before we dive into this code, I'd encourage everyone reading this article to take time to read over the code and draw your own conclusions about it. Did you find the code easy to understand? If so, why? If not, what makes it complex?

Also, Bob, can you confirm that you stand by this code (i.e. the code properly exemplifies the design philosophy of *Clean Code* and this is the way you believe the code should appear if it were used in production)?

UB:

Ah, yes. The `PrimeGenerator`. This code comes from the 1982 paper on [Literate Programming](#) written by Donald Knuth. The program was originally written in Pascal, and was automatically generated by Knuth's WEB system into a single very large method which I translated into Java.

Of course this code was never meant for production. Both Knuth and I used it as a pedagogical example. In *Clean Code* it appears in a chapter named *Classes*. The lesson of the chapter is that a very large method will often contain many different sections of code that are better decomposed into independent classes.

In the chapter I extracted three classes from that function: `PrimePrinter`, `RowColumnPagePrinter` and `PrimeGenerator`.

One of those extracted classes was the `PrimeGenerator`. It had the following code (which I did not publish in the book.) The variable names and the overall structure are Knuth's.

```
public class PrimeGenerator {
    protected static int[] generate(int n) {
        int[] p = new int[n];
        ArrayList<Integer> mult = new ArrayList<Integer>();
        p[0] = 2;
        mult.add(2);
        int k = 1;
        for (int j = 3; k < p.length; j += 2) {
            boolean jprime = false;
            int ord = mult.size();
            int square = p[ord] * p[ord];
            if (j == square) {
                mult.add(j);
            } else {
                jprime=true;
                for (int mi = 1; mi < ord; mi++) {
                    int m = mult.get(mi);
```



```

        while (m < j)
            m += 2 * p[mi];
        mult.set(mi, m);
        if (j == m) {
            jprime = false;
            break;
        }
    }
    if (jprime)
        p[k++] = j;
}
return p;
}
}

```

Even though I was done with the lesson of the chapter, I didn't want to leave that method looking so outdated. So I cleaned it up a bit as an afterthought. My goal was not to describe how to generate prime numbers. I wanted my readers to see how large methods, that violate the Single Responsibility Principle, can be broken down into a few smaller well-named classes containing a few smaller well-named methods.

JOHN:

Thanks for the background. Even though the details of that code weren't the main point of the chapter, presumably the code represents what you think is the "right" and "cleanest" way to do things, given the algorithm at hand. And that's where I disagree.

There are many design problems with `PrimeGenerator`, but for now I'll focus on method length. The code is chopped up so much (8 teeny-tiny methods) that it's difficult to read. For starters, consider the

`isNotMultipleOfAnyPreviousPrimeFactor` method. This method invokes `isMultipleOfNthPrimeFactor`, which invokes `smallestOddNthMultipleNotLessThanCandidate`. These methods are shallow and entangled: in order to understand `isNot...` you have to read the other two methods and load all of that code into your mind at once. For example, `isNot...` has side effects (it modifies `multiplesOfPrimeFactors`) but you can't see that unless you read all three methods.

UB:

I think you have a point. Eighteen years ago, when I was in the throes of this refactoring, the names and structure made perfect sense to me. They make sense to me now, too -- but that's because I once again understand the algorithm. When I returned to the algorithm for the first time a few days ago, I struggled with the names and structure. Once I understood the algorithm the names and structure made perfect sense.

JOHN:

Those names are problematic even for someone who understands the algorithm; we'll talk about them a bit later, when discussing comments. And, if code no longer makes sense to the writer when the writer returns to the code later, that means the code is problematic. The fact that code can eventually be understood (with great pain and suffering) does not excuse its entanglement.

UB:

Would that we had such a crystal ball that we could help our future selves avoid such "*great pain and suffering*". ;-)

JOHN:

There is no need for a crystal ball. The problems with `PrimeGenerator` are pretty obvious, such as the entanglement and interface complexity; maybe you were surprised that it is hard to understand, but I am not. Said another way, if you are unable to predict whether your code will be easy to understand, there are problems with your design methodology.

UB:


Fair enough. I will say, however, that I had equal "*pain and suffering*" interpreting your rewrite (below). So, apparently, neither of our methodologies were sufficient to rescue our readers from such struggles.


JOHN:

Going back to my introductory remarks about complexity, splitting up `isNot...` into three methods doesn't reduce the amount of information you have to keep in your mind. It just spreads it out, so it isn't as obvious that you need to read all three methods together. And, it's harder to see the overall structure of the code because it's split up: readers have to flip back and forth between the methods, effectively reconstructing a monolithic version in their minds. Because the pieces are all related, this code will be easiest to understand if it's all together in one place.

UB:

I disagree. Here is `isNotMultipleOfAnyPreviousPrimeFactor` .

```
private static boolean isNotMultipleOfAnyPreviousPrimeFactor(int cand   
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {  
        if (isMultipleOfNthPrimeFactor(candidate, n))  
            return false;  
    }  
    return true;  
}
```



If you trust the `isMultipleOfNthPrimeFactor` method, then this method stands alone quite nicely. I mean we loop through all `n` previous primes and see if the candidate is a multiple. That's pretty straight forward.

Now it would be fair to ask the question how we determine whether the candidate is a multiple, and in that case you'd want to inspect the `isMultiple...` method.

JOHN:

This code does appear to be simple and obvious. Unfortunately, this appearance is deceiving. If a reader trusts the name `isMultipleOfNthPrimeFactor` (which suggests a predicate with no side effects) and doesn't bother to read its code, they will not realize that it has side effects, and that the side effects create a constraint on the `candidate` argument to `isNot...` (it must be monotonically non-decreasing from invocation to invocation). To understand these behaviors, you have to read both `isMultiple...` and `smallestOddNth...`. The current decomposition hides this important information from the reader.

If there is one thing more likely to result in bugs than not understanding code, it's thinking you understand it when you don't.

UB:

That's a valid concern. However, it is tempered by the fact that the functions are presented in the order they are called. Thus we can expect that the reader has already seen the main loop and understands that `candidate` increases by two each iteration.

The side effect buried down in `smallestOddNth...` is a bit more problematic. Now that you've pointed it out I don't like it much. Still, that side effect should not confound the basic understanding of `isNot...` .

In general, if you trust the names of the methods being called then understanding the caller does not require understanding the callee. For example:

```
for (Employee e : employees)
    if (e.shouldPayToday())
        e.pay();
```



This would not be made more understandable if we replaced those two method calls with their implementations. Such a replacement would simply obscure the intent.

JOHN:

This example works because the called methods are relatively independent of the parent. Unfortunately that is not the case for `isNot...`.

In fact, `isNot...` is not only entangled with the methods it calls, it's also entangled with its callers. `isNot...` only works if it is invoked in a loop where `candidate` increases monotonically. To convince yourself that it works, you have to find the code that invokes `isNot...` and make sure that `candidate` never decreases from one call to the next. Separating `isNot...` from the loop that invokes it makes it harder for readers to convince themselves that it works.

UB:

Which, as I said before, is why the methods are ordered the way they are. I expect that by the time you get to `isNot...` you've already read `checkOddNumbersForSubsequentPrimes` and know that `candidate` increases by twos.

JOHN:

Let's discuss this briefly, because it's another area where I disagree with *Clean Code*. If methods are entangled, there is no clever ordering of the method definitions that will fix the problem.

In this particular situation two other methods intervene between the loop in `checkOdd...` and `isNot...`, so readers will have forgotten the loop context before they get to `isNot...`. Furthermore, the actual code that creates a dependency on the loop isn't in `isNot...`: it's in `smallestOdd...`, which is even farther away from `checkOdd...`.

UB:

I sincerely doubt anyone is going to forget that `candidate` is being increased by twos. It's a pretty obvious way to avoid waste.

JOHN:

In my opening remarks I talked about how it's important to reduce the amount of information people have to keep in their minds at once. In this situation, readers have to remember that loop while they read four intervening methods that are mostly unrelated to the loop. You apparently think this will be easy and natural (I disagree). But it's even worse than that. There is no indication which parts of `checkOdd...` will be important later on, so the only safe approach is to remember *everything*, from *every* method, until you have encountered every other method that could possibly descend from it. And, to make the connection between the pieces, readers must also reconstruct the call graph to notice that, even through 4 layers of method call, the code in `smallestOdd...` places constraints on the loop in `checkOdd...`. This is an unreasonable cognitive burden to place on readers.

If two pieces of code are tightly related, the solution is to bring them together. Separating the pieces, even in physically adjacent methods, makes the code harder to understand.

To me, all of the methods in `PrimeGenerator` are entangled: in order to understand the class I had to load all of them into my mind at once. I was constantly flipping back and forth between the methods as I read the code. This is a red flag indicating that the code has been over-decomposed.

Bob, can you help me understand why you divided the code into such tiny methods? Is there some benefit to having so many methods that I have missed?

UB:

I think you and I are just going to disagree on this. In general, I believe in the principle of small well-named methods and the separation of concerns. Generally speaking if you can break a large method into several well-named smaller methods with different concerns, and by doing so expose their interfaces, and the high level functional decomposition, then that's a good thing.

- Looping over the odd numbers is one concern.
- Determining primality is another.
- Marking off the multiples of primes is yet another.

It seems to me that separating and naming those concerns helps to expose the way the algorithm works -- even at the expense of some entanglement.

In your solution, which we are soon to see below, you break the algorithm up in a similar way. However, instead of separating the concerns into functions, you separate them into sections with comments above them.

You mentioned that in my solution readers will have to keep the loop context in mind while reading the other functions. I suggest that in your solution, readers will have to keep the loop context in mind while reading your explanatory comments. They may have to "flip back and forth" between the sections in order to establish their understanding.

Now perhaps you are concerned that in my solution the "flipping" is a longer distance (in lines) than in yours. I'm not sure that's a significant point since they all fit on the same screen (at least they do on my screen) and the landmarks are pretty obvious.

Method Length Summary

JOHN:

It sounds like it's time to wrap up this section. Is this a reasonable summary of where we agree and disagree?

- We agree that modular design is a good thing.
- We agree that it is possible to over-decompose, and that *Clean Code 1st ed.* doesn't provide much guidance on how to recognize over-decomposition.
- We disagree on how far to decompose: you recommend decomposing code into much smaller units than I do. You believe that the additional decomposition you recommend makes code easier to understand; I believe that it goes too far and actually makes code more difficult to understand.
- You believe that the One Thing Rule, applied with judgment, will lead to appropriate decompositions. I believe it lacks guardrails and will lead to over-decomposition.
- We agree that the internal decomposition of `PrimeGenerator` into methods is problematic. You point out that your main goal in writing `PrimeGenerator` was to show how to decompose into classes, not so much how to decompose a class internally into methods.
- Entanglement between methods in a class doesn't bother you as much as it bothers me. You believe that the benefits of decomposing methods can compensate for problems caused by entanglement. I believe they can't: when decomposed methods are entangled, they are harder to read than if they were not decomposed, and this defeats the whole purpose of decomposition.
- You believe that ordering the methods in a class can help to compensate for entanglement between methods; I don't.

UB:

I think this is a fair assessment of our agreements and disagreements. We both value decomposition, and we both avoid entanglement; but we disagree on the relative weighting of those two values.

Comments

JOHN:

Let's move on to the second area of disagreement: comments. In my opinion, the *Clean Code* approach to commenting results in code with inadequate documentation, which increases the cost of software development. I'm sure you disagree, so let's discuss.

Here is what *Clean Code* says about comments (page 54):

The proper use of comments is to compensate for our failure to express ourselves in code. Note that I use the word failure. I meant it. Comments are always failures. We must have them because we cannot always figure out how to express ourselves without them, but their use is not a cause for celebration... Every time you write a comment, you should grimace and feel the failure of your ability of expression.

I have to be honest: I was horrified when I first read this text, and it still makes me cringe. This stigmatizes writing comments. Junior developers will think "if I write comments, people may think I've failed, so the safest thing is to write no comments."

UB:

That chapter begins with these words:

Nothing can be quite so helpful as a well placed comment.

It goes on to say that comments are a *necessary* evil.

The only way a reader could infer that they should write no comments is if they hadn't actually read the chapter. The chapter walks through a series of comments, some bad, some good.

JOHN:

Clean Code focuses a lot more on the "evil" aspects of comments than the "necessary" aspects. The sentence you quoted above is followed by two sentences criticizing comments. Chapter 4 spends 4 pages talking about good comments, followed by 15 pages talking about bad comments. There are snubs like "the only truly good comment is the comment you found a way not to write". And "Comments are always failures" is so catchy that it's the one thing readers are most likely to remember from the chapter.

UB:

The difference in page count is because there are just a few ways to write good comments, and so many more ways to write bad ones.

JOHN:

I disagree; this illustrates your bias against comments. If you look at Chapter 13 of APOSD, it finds a lot more constructive ways to use comments than *Clean Code*. And if you compare the tone of Chapter 13 of APOSD with Chapter 4 of *Clean Code*, the hostility of *Clean Code* towards comments becomes pretty clear.

UB:

I'll leave you to balance that last comment with the initial statement, and the final example, in the *Comments* chapter. They do not communicate "hostility".

I'm not hostile to comments in general. I *am* very hostile to gratuitous comments.

You and I likely both survived through a time when comments were absolutely necessary. In the '70s and '80s I was an assembly language programmer. I also wrote a bit of FORTRAN. Programs in those languages that had no comments were impenetrable.

As a result it became conventional wisdom to write comments by default. And, indeed, computer science students were taught to write comments uncritically. Comments became *pure good*.

In *Clean Code* I decided to fight that mindset. Comments can be *really bad* as well as good.

JOHN:

I don't agree that comments are less necessary today than they were 40 years ago.

Comments are crucially important and add enormous value to software. The problem is that there is a lot of important information that simply cannot be expressed in code. By adding comments to fill in this missing information, developers can make code dramatically easier to read. This is not a "failure of their ability to express themselves", as you put it.

UB:

It's very true that there is important information that is not, or cannot be, expressed in code. That's a failure. A failure of our languages, or of our ability to use them to express ourselves. In every case a comment is a failure of our ability to use our languages to express our intent.

And we fail at that very frequently, and so comments are a necessary evil -- or, if you prefer, *an unfortunate necessity*. If we had the perfect programming language (TM) we would never write another comment.

JOHN:

I don't agree that a perfect programming language would eliminate the need for comments. Comments and code serve very different purposes, so it's not obvious to me that we should use the same language for both. In my experience, English works quite well as a language for comments. Why do you feel that information about a program should be expressed entirely in code, rather than using a combination of code and English?

UB:

I bemoan the fact that we must sometimes use a human language instead of a programming language. Human languages are imprecise and full of ambiguities. Using a human language to describe something as precise as a program is very hard, and fraught with many opportunities for error and inadvertent misinformation.

JOHN:

I agree that English isn't always as precise as code, but it can still be used in precise ways and comments typically don't need the same degree of precision as code. Comments often contain qualitative information such as *why* something is being done, or the overall idea of something. English works better for these than code because it is a more expressive language.

UB:

I have no argument with that statement.

JOHN:

Are you concerned that comments will be incorrect or misleading and that this will slow down software development? I often hear people complain about stale comments (usually as an excuse for writing no comments at all) but I have not found them be a significant problem over my career. Incorrect comments do happen, but I don't encounter them very often and when I do, they rarely cost me much time. In contrast, I waste *enormous* amounts of time because of inadequate documentation; it's not unusual for me to spend 50-80% of my development time wading through code to figure out things that would be obvious if the code was properly commented.

UB:

You and I have had some very different experiences.

I have certainly been helped by well-placed comments. I have also, just as certainly, (and within this very document) been distracted and confused by a comment that was incorrect, misplaced, gratuitous, or otherwise just plain bad.

JOHN:

I invite everyone reading this article to ask yourself the following questions:

- How much does your software development speed suffer because of incorrect comments?
- How much does your software development speed suffer because of missing comments?

For me the cost of missing comments is easily 10-100x the cost of incorrect comments. That is why I cringe when I see things in *Clean Code* that discourage people from writing comments.

Let's consider the `PrimeGenerator` class. There is not a single comment in that code; does this seem appropriate to you?

UB:

I think it was appropriate for the purpose for which I wrote it. It was an adjunct to the lesson that very large methods can be broken down into smaller classes containing smaller methods. Adding lots of explanatory comments would have detracted from that point.

In general, however, the commenting style I used in Listing 4-8 is more appropriate. That listing, at the very end of the *Comments* chapter, describes yet another `PrimeGenerator` with a slightly different algorithm, and a better set of comments.

JOHN:

I disagree that adding comments would have distracted from your point, and I think Listing 4-8 is also woefully undercommented. But let's not argue about either of those issues. Instead, let's discuss what comments the `PrimeGenerator` code *should* have if it were used in production. I will make some suggestions, and you can agree or disagree.

For starters, let's discuss your use of megasyllabic names like

`isLeastRelevantMultipleOfLargerPrimeFactor`. My understanding is that you advocate using names like this instead of using shorter names augmented with descriptive comments: you're effectively moving the comments into code. To me, this approach is problematic:

- Long names are awkward. Developers effectively have to retype the documentation for a method every time they invoke it, and the long names waste horizontal space and trigger line wraps in the code. The names are also awkward to read: my mind wants to parse every syllable every time I read it, which slows me down. Notice that both you and I resorted to abbreviating names in this discussion: that's an indication that the long names are awkward and unhelpful.
- The names are hard to parse and don't convey information as effectively as a comment. When students read `PrimeGenerator` one of the first things they complain about is the long names (students can't make sense of them). For example, the name above is vague and cryptic: what does "least relevant" mean, and what is a "larger prime factor"? Even with a complete understanding of the code in the method, it's hard for me to make sense of the name. If this name is going to eliminate the need for a comment, it needs to be even longer.

In my opinion, the traditional approach of using shorter names with descriptive comments is more convenient and conveys the required information more effectively. What advantage is there in the approach you advocate?

UB:

"Megasyllabic": Great word!

I like my method names to be sentence fragments that fit nicely with keywords and assignment statements. It makes the code a bit more natural to read.

```
if (isTooHot)
    cooler.turnOn();
```



I also follow a simple rule about the length of names. The larger the scope of a method, the shorter its name should be and vice versa -- the shorter the scope the longer the name. The private methods I extracted in this case live in very small scopes, and so have longish names. Methods like this are typically called from only one place, so there is no burden on the programmer to remember a long name for another call.

JOHN:

Names like `isTooHot` are totally fine by me. My concern is about names like `isLeastRelevantMultipleOfLargerPrimeFactor`.

It's interesting that as methods get smaller and narrower, you recommend longer names. What this says to me is that the interfaces for those functions are more complex, so it takes more words to describe them. This provides supporting evidence for my assertion a while back that the more you split up a method, the shallower the resulting methods will be.

UB:

It's not the functions that get smaller, it's the scope that gets smaller. A private function has a smaller scope than the public function that calls it. A function called by that private function has an even smaller scope. As we descend in scope, we also descend in situational detail. Describing such detail often requires a long name, or a long comment. I prefer to use a name.

As for long names being hard to parse, that's a matter of practice. Code is full of things that take practice to get used to.

JOHN:

I don't accept this. Code may be full of things that take practice to get used to, but that doesn't excuse it. Approaches that require more practice are worse than those that require less. If it's going to take a lot of work to get comfortable with the long names then there had better be some compensating benefit; so far I'm not seeing any. And I don't see any reason to believe that practice will make those names easier to digest.

In addition, your comment above violates one of my fundamental rules, which is "complexity is in the eye of the reader". If you write code that someone else thinks is complicated, then you must accept that the code is probably complicated (unless you think the reader is completely incompetent). It is not OK to make excuses or suggest that it is really the reader's problem ("you just don't have enough practice"). I'm going to have to live by this same rule a bit later in our discussion.

UB:

Fair enough. As for the meaning of "leastRelevant", that's a much larger problem that you and I will encounter shortly. It has to do with the intimacy that the author has with the solution, and the reader's lack of that intimacy.

JOHN:

You still haven't answered my question: why is it better to use super-long names rather than shorter names augmented with descriptive comments?

UB:

It's a matter of preference for me. I prefer long names to comments. I don't trust comments to be maintained, nor do I trust that they will be read. Have you ever noticed that many IDEs paint comments in light grey so that they can be easily ignored? It's harder to ignore a name than a comment.

(BTW, I have my IDE paint comments in bright fire-engine red)

JOHN:

I don't see why a monster name is more likely to be "maintained" than a comment, and I don't agree that IDEs encourage people to ignore comments (this is your bias coming out again). My current IDE (VSCode) doesn't use a lighter color for comments. My previous one (NetBeans) did, but the color scheme didn't hide the comments; it distinguished them from the code in a way that made both code and comments easier to read.

Now that we've discussed the specific issue of comments vs. long method names, let's talk about comments in general. I think there are two major reasons why comments are needed. The first reason for comments is abstraction. Simply put, without comments there is no way to have abstraction or modularity.

Abstraction is one of the most important components of good software design. I define an abstraction as "a simplified way of thinking about something that omits unimportant details." The most obvious example of an abstraction is a method. It should be possible to use a method without reading its code. The way we achieve this is by writing a header comment that describes the method's *interface* (all the information someone needs in order to invoke the method). If the method is well-designed, the interface will be much simpler than the code of the method (it omits implementation details), so the comments reduce the amount of information people must have in their heads.

UB:

Long ago, in a 1995 book, I defined abstraction as:

The amplification of the essential and the elimination of the irrelevant.

I certainly agree that abstraction is of importance to good software design. I also agree that well-placed comments can enhance the ability of readers to understand the abstractions we are attempting to employ. I disagree that comments are the *only*, or even the *best*, way to understand those abstractions. But sometimes they are the only option.

But consider:

```
addSongToLibrary(String title, String[] authors, int durationInSeconds)
```



This seems like a very nice abstraction to me, and I cannot imagine how a comment might improve it.

JOHN:

Our definitions of abstraction are very similar; that's good to see. However, the `addSongToLibrary` declaration is not (yet) a good abstraction because it omits information that is essential. In order to use `addSongToLibrary`, developers need answers to the following questions:

- Is there any expected format for an author string, such as "LastName, FirstName"?
- Are the authors expected to be in alphabetical order? If not, is the order significant in some other way?
- What happens if there is already a song in the library with the given title but different authors? Is it replaced with the new one, or will the library keep multiple songs with the same title?
- How is the library stored (e.g. is it entirely in memory? saved on disk?)? If this information is documented somewhere else, such as the overall class documentation, then it need not be repeated here.

Thus `addSongToLibrary` needs quite a few comments. Sometimes the signature of a method (names and types of the method, its arguments, and its return value) contains all the information needed to use it, but this is pretty rare. Just skim through the documentation for your favorite library package: in how many cases could you understand how to use a method with only its signature?

UB:

Yes, there are times when the signature of a method is an incomplete abstraction and a comment is required. This is especially true when the interface is part of a public API, or an API intended for use by a separate team of developers. Within a single development team, however, long descriptive comments on interfaces are often more of an impediment than a help. The team has intimate knowledge of the internals of the system, and will generally be able to understand an interface simply from its signature.

JOHN:

In one of our in-person discussions you argued that interface comments are unnecessary because when a group of developers is working on a body of code they can collectively keep the entire code "loaded" in their minds, so comments are unnecessary: if you have a question, just ask the person who is familiar with that code. This creates a huge cognitive load to keep all that code mentally loaded, and it's hard for me to imagine that it would actually work. Maybe your memory is better than mine, but I find that I quickly forget code that I wrote just a few weeks ago. In a project of any size, I think your approach would result in developers spending large amounts of time reading code to re-derive the interfaces, and probably making mistakes along the way. Spending a few minutes to document the interfaces would save time, reduce cognitive load, and reduce bugs.

UB:

I think that certain interfaces need comments, even if they are private to the team. But I think it is more often the case that the team is familiar enough with the system that well named methods and arguments are sufficient.

JOHN:

Let's consider a specific example from `PrimeGenerator`: the `isMultipleOfNthPrimeFactor` method. When someone reading the code encounters the call to `isMultiple...` in `isNot...` they need to understand enough about how `isMultiple...` works in order to see how it fits into the code of `isNot...`. The method name does not fully document the interface, so if there is no header comment then readers will have to read the code of `isMultiple`. This will force readers to load more information into their heads, which makes it harder to work in the code.

Here is my first attempt at a header comment for `isMultiple`:

```
/**
 * Returns true if candidate is a multiple of primes[n], false otherwise
 * May modify multiplesOfPrimeFactors[n].
 * @param candidate
 *     Number being tested for primality; must be at least as
```




```
*      large as any value passed to this method in the past.  
* @param n  
*      Selects a prime number to test against; must be  
*      <= multiplesOfPrimeFactors.size().  
*/
```

What do you think of this?

UB:

I think it's accurate. I wouldn't delete it if I encountered it. I don't think it should be a javadoc.

The first sentence is redundant with the name `isMultipleOfNthPrimeFactor` and so could be deleted. The warning of the side effect is useful.

JOHN:

I agree that the first sentence is largely redundant with the name, and I debated with myself about whether to keep it. I decided to keep it because I think it is a bit more precise than the name; it's also easier to read. You propose to eliminate the redundancy between the comment and the method name by dropping the comment; I would eliminate the redundancy by shortening the method name.

By the way, you complained earlier about comments being less precise than code, but in this case the comment is *more* precise (the method name can't include text like `primes[n]`).

UB:

Fair enough. There are times when precision is better expressed in a comment.

Continuing with my critique of your comment above: The name `candidate` is synonymous with "Number being tested for primality".

In the end, however, all the words in a comment are just going to have to sit in my brain until I understand why they are there. I'm also going to have to worry if they are accurate. So I'm going to have to read the code to understand and validate the comment.

JOHN:

Whoah. That loud sound you just heard was my jaw hitting the floor. Help me understand this a bit better: approximately what fraction of comments that you encounter in practice are you willing to trust without reading the code to verify them?

UB:

I look at every comment as potential misinformation. At best they are a way to crosscheck the author's intent against the code. The amount of credence I give to a comment depends a lot on how easy they make that crosscheck. When I read a comment that does not cause me to crosscheck, then I consider it to be of no value. When I see a comment that causes me to crosscheck, and when that crosscheck turns out to be valuable, then that's a really good comment.

Another way to say this is that the best comments tell me something surprising and verifiable about the code. The worst are those that waste my time telling me something obvious, or incorrect.

JOHN:

It sounds like your answer is 0%: you don't trust any comment unless it has been verified against the code. This makes no sense to me. As I said above, the vast majority of comments are correct. It's not hard to write comments; the students in my software design class are doing this pretty well within a few weeks. It's also not hard to keep comments up to date as code evolves. Your refusal to trust comments is another sign of your irrational bias against comments.

Refusing to trust comments incurs a very high cost. In order to understand how to invoke a method, you will have to read all of the code of that method; if the method invokes other methods, you will also have to read them, and the methods they invoke, recursively. This is an enormous amount of work in comparison to reading (and trusting) a simple interface comment like the one I wrote above.

If you choose not to write an interface comment for methods, then you leave the interface of that method undefined. Even if someone reads the code of the method, they won't be able to tell which parts of the implementation are expected to remain the same and which parts may change (there is no way to specify this "contract" in code). This will result in misunderstanding and more bugs.

UB:

Well, I guess I've just been burned more than you have. I've gone down too many false comment induced rabbit holes, and wasted too much time on worthless word salads.

Of course my trust in comments is not a binary thing. I read them if they are there; but I don't implicitly trust them. The more gratuitous I feel the author was, or the less adept at english the author is, the less I trust the comments.

As I said above, our IDEs tend to paint comments in an ignorable color. I have my IDE paint comments in bright fire engine red because when I write a comment I intend for it to be read.

By the same token I use long names as a substitute for comments because I intend for those long names to be read; and it is very hard for a programmer to ignore names.

JOHN:

I mentioned earlier that there are two general reasons why comments are needed. So far we've been discussing the first reason (abstraction). The second general reason for comments is for important information that is not obvious from the code. The algorithm in `PrimeGenerator` is very non-obvious, so quite a few comments are needed to help readers understand what is going on and why. Most of the algorithm's complexity arises because it is designed to compute primes efficiently:

- The algorithm goes out of its way to avoid divisions, which were quite expensive when Knuth wrote his original version (they aren't that expensive nowadays).
- The first multiple for each new prime number is computed by squaring the prime, rather than multiplying it by 3. This is mysterious: why is it safe to skip the intervening odd multiples? Furthermore, it might seem that this optimization only has a small impact on performance, but in fact it makes an *enormous* difference (orders of magnitude). Using the square has the side effect that when testing a candidate, only primes up to the square root of the candidate are tested. If 3x were used as the initial multiple, primes within a factor of 3 of the candidate would be tested; that's a *lot* more tests. This implication of using the square is so non-obvious that I only realized it while preparing material for this discussion; it never occurred to me in the many times I have discussed the code with students.

Neither of these issues is obvious from the code; without comments, readers are left to figure them out on their own. The students in my class are generally unable to figure out either of them in the 30 minutes I give them, but I think that comments would have allowed them to understand in a few minutes. Going back to my introductory remarks, this is an example where information is important, so it needs to be made available.

Do you agree that there should be comments to explain each of these two issues?

UB:

I agree that the algorithm is subtle. Setting the first prime multiple as the square of the prime was deeply mysterious at first. I had to go on an hour-long bike ride to understand it.

Maybe we have a fundamental difference of philosophy here. I get the sense that you are happy to give readers a few clues and leave it to them to put the clues together. Perhaps you don't mind if people have to stare at something for a while to figure it out? I don't agree with this approach: it results in wasted time, misunderstandings, and bugs. I think software should be totally *obvious*, where readers don't need to be clever or "stare at this for some time" to figure things out. Suffering followed by catharsis is great for Greek tragedies, but not for reading code. Every question a reader might have should be naturally answered, either in the code or in comments. Key ideas and important conclusions should be stated explicitly, not left for the reader to deduce. Ideally, even if a reader is in a hurry and doesn't read the code very carefully, their first guesses about how things work (and why) should be correct. To me, that's clean code.

UB:

I don't disagree with your sentiment. Good clean code should be as easy as possible to understand. I want to give my readers as many clues as possible so that the code is intuitive to read.

That's the goal. As we are about to see, that can be a tough goal to achieve.

JOHN:

In that case, do you still stand by the "picture" you painted above? It doesn't seem consistent with what you just said. And if you really wanted to give your readers as many clues as possible, you'd include a lot more comments.

UB:

I stand by the picture as far as it's accuracy is concerned. And I think it makes a good crosscheck. I have no illusions that it is easy to understand.

This algorithm is challenging and will require work to comprehend. I finally understood it when I drew this picture in my mind while on that bike ride. When I got home I drew it for real and presented it in hopes that it might help someone willing to do the work to understand it.

Comments Summary

JOHN:

Let's wrap up this section of the discussion. Here is my summary of where we agree and disagree.

- Our overall views of comments are fundamentally different. I see more value in comments than you do, and I believe that they play a fundamental and irreplaceable role in system design. You agree that there are places where comments are necessary, but that comments don't always make it easier to understand code, so you see far fewer places where comments are needed.
- I would probably write 5-10x more lines of comments for a given piece of code than you would.
- I believe that missing comments are a much greater cause of lost productivity than erroneous or unhelpful comments; you believe that comments are a net negative, as generally practiced: bad comments cost more time than good comments save.
- You view it as problematic that comments are written in English rather than a programming language. I don't see this as particularly problematic and think that in many cases English works better.
- You recommend that developers should take information that I would represent as comments and recast it into code if at all possible. One example of this is super-long method names. I believe that super-long names are awkward and hard to understand, and that it would be better to use shorter names supplemented with comments.
- I believe that it is not possible to define interfaces and create abstractions without a lot of comments. You agree for public APIs, but see little need to comment interfaces that are internal to the team.
- You are unwilling to trust comments until you have read code to verify them. I generally trust comments; by doing so, I don't need to read as much code as you do. You think this exposes me to too much risk.
- We agree that implementation code only needs comments when the code is nonobvious. Although neither of us argues for a large number of implementation comments, I'm more likely to see value in them than you do.

Overall, we struggled to find areas of agreement on this topic.

UB:

This is a fair assessment of our individual positions; which I assume are based on our different individual experiences. Over the years I have found the vast majority of comments, as generally practiced in the industry, to be unhelpful. You seem to have found more help in the comments you have encountered.

John's Rewrite of PrimeGenerator

JOHN:

I mentioned that I ask the students in my software design class to rewrite `PrimeGenerator` to fix all of its design problems. Here is my rewrite (note: this was written before we began our discussion; given what I have learned during the discussion, I would now change several of the comments, but I have left this in its original form):

```
package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator2 {

    /**
     * Computes the first prime numbers; the return value contains th
     * computed primes, in increasing order of size.
     * @param n
     *     How many prime numbers to compute.
     */
    public static int[] generate(int n) {
        int[] primes = new int[n];

        // Used to test efficiently (without division) whether a cand
        // is a multiple of a previously-encountered prime number. Ea
        // here contains an odd multiple of the corresponding entry i
        // primes. Entries increase monotonically.
        int[] multiples = new int[n];

        // Index of the last value in multiples that we need to consi
        // when testing candidates (all elements after this are great
        // than our current candidate, so they don't need to be consi
        int lastMultiple = 0;

        // Number of valid entries in primes.
        int primesFound = 1;

        primes[0] = 2;
        multiples[0] = 4;

        // Each iteration through this loop considers one candidate;
        // the even numbers, since they can't be prime.
        candidates: for (int candidate = 3; primesFound < n; candidat
            if (candidate >= multiples[lastMultiple]) {
                lastMultiple++;
            }

        // Each iteration of this loop tests the candidate agains
```



```

// potential prime factor. Skip the first factor (2) sinc
// only consider odd candidates.
for (int i = 1; i <= lastMultiple; i++) {
    while (multiples[i] < candidate) {
        multiples[i] += 2*primes[i];
    }
    if (multiples[i] == candidate) {
        continue candidates;
    }
}
primes[primesFound] = candidate;

// Start with the prime's square here, rather than 3x the
// This saves time and is safe because all of the interve
// multiples will be detected by smaller prime numbers. A
// example, consider the prime 7: the value in multiples
// start at 49; 21 will be ruled out as a multiple of 3,
// 35 will be ruled out as a multiple of 5, so 49 is the
// multiple that won't be ruled out by a smaller prime.
multiples[primesFound] = candidate*candidate;
primesFound++;
}
return primes;
}
}

```

Everyone can read this and decide for themselves whether they think it is easier to understand than the original. I'd like to mention a couple of overall things:

- There is only one method. I didn't subdivide it because I felt the method already divides naturally into pieces that are distinct and understandable. It didn't seem to me that pulling out methods would improve readability significantly. When students rewrite the code, they typically have 2 or 3 methods, and those are usually OK too.
- There are a *lot* of comments. It's extremely rare for me to write code with this density of comments. Most methods I write have no comments in the body, just a header comment describing the interface. But this code is subtle and tricky, so it needs a lot of comments to make the subtleties clear to readers. The long length of some of the comments is a red flag indicating that I struggled to find a clear and simple explanation for the code. Even with all the additional explanatory material this version is a bit shorter than the original (65 lines vs. 70).

UB:

I presume this is a complete rewrite. My guess is that you worked to understand the algorithm from *Clean Code* and then wrote this from scratch. If that's so, then fair enough.

In *Clean Code* I *refactored* Knuth's algorithm in order to give it a little structure. That's not the same as a complete rewrite.

Having said that, your version is much better than either Knuth's or mine.

I wrote that chapter 18 years ago, so it's been a long time since I saw and understood this algorithm. When I first saw your challenge I thought: "Oh, I can figure out my own code!" But, no. I could see all the moving parts, but I could not figure out why those moving parts generated a list of prime numbers.

So then I looked at your code. I had the same problem. I could see all the moving parts, all with comments, but I still could not figure out why those moving parts generated a list of prime numbers.

Figuring that out required a lot of staring at the ceiling, closing my eyes, visualizing, and riding my bike.

Among the problems I had were the comments you wrote. Let's take them one at a time.

```
/**  
 * Computes the first prime numbers; the return value contains the  
 * computed primes, in increasing order of size.  
 * @param n  
 *      How many prime numbers to compute.  
 */  
public static int[] generate(int n) {
```



It seems to me that this would be better as:

```
public static int[] generateNPrimeNumbers(int n) {
```



or if you must:

```
//Return the first n prime numbers  
public static int[] generate(int n) {
```



I'm not opposed to Javadocs as a rule; but I write them only when absolutely necessary. I also have an aversion for descriptions and `@param` statements that are perfectly obvious from the method signature.

The next comment cost me a good 20 minutes of puzzling things out.

```
// Used to test efficiently (without division) whether a candidate  
// is a multiple of a previously-encountered prime number. Each entry
```



```
// here contains an odd multiple of the corresponding entry in  
// primes. Entries increase monotonically.
```

First of all I'm not sure why the "division" statement is necessary. I'm old school, so I expect that everyone knows to avoid division in inner loops if it can be avoided. But maybe I'm wrong about that...

Also, the *Sieve of Eratosthenes* does not do division, and is a lot easier to understand *and explain* than this algorithm. So why this particular algorithm? I think Knuth was trying to save *memory* -- and in 1982 saving memory was important. This algorithm uses a lot less memory than the sieve.

Then came the phrase: Each entry here contains an odd multiple... I looked at that, and then at the code, and I saw: `multiples[0] = 4; .`

"That's not odd" I said to myself. "So maybe he meant even."

So then I looked down and saw: `multiples[i] += 2*primes[i];`

"That's adding an even number!" I said to myself. "I'm pretty sure he meant to say 'even' instead of 'odd'."

I hadn't yet worked out what the `multiples` array was. So I thought it was perfectly reasonable that it would have even numbers in it, and that your comment was simply an understandable word transposition. After all, there's no compiler for comments so they suffer from the kinds of mistakes that humans often make with words.

It was only when I got to `multiples[primesFound] = candidate*candidate;` that I started to question things. If the `candidate` is prime, shouldn't `prime*prime` be odd in every case beyond 2? I had to do the math in my head to prove that. $(2n+1)(2n+1) = 4n^2 + 4n + 1$... Yeah, that's odd.

OK, so the `multiples` array is full of odd multiples, except for the first element, since it will be multiples of 2.

So perhaps that comment should be:

```
// multiples of corresponding prime.
```



Or perhaps we should change the name of the array to something like `primeMultiples` and drop the comment altogether.

Moving on to the next comment:

```
// Each iteration of this loop tests the candidate against one
// potential prime factor. Skip the first factor (2) since we
// only consider odd candidates.
```



That doesn't make a lot of sense. The code it's talking about is:

```
for (int i = 1; i <= lastMultiple; i++) {
    while (multiples[i] < candidate) {
```



The `multiples` array, as we have now learned, is an array of *multiples* of prime numbers. This loop is not testing the candidate against prime *factors*, it's testing it against the current prime *multiples*.

Fortunately for me the third or fourth time I read this comment I realized that you really meant to use the word "multiples". But the only way for me to know that was to understand the algorithm. And when I understand the algorithm, why do I need the comment?

That left me with one final question. What the deuce was the reason behind:

```
multiples[primesFound] = candidate*candidate;
```



Why the square? That makes no sense. So I changed it to:

```
multiples[primesFound] = candidate;
```



And it worked just fine. So this must be an optimization of some kind.

Your comment to explain this is:

```
// Start with the prime's square here, rather than 3x the prime.
// This saves time and is safe because all of the intervening
// multiples will be detected by smaller prime numbers. As an
// example, consider the prime 7: the value in multiples will
// start at 49; 21 will be ruled out as a multiple of 3, and
// 35 will be ruled out as a multiple of 5, so 49 is the first
// multiple that won't be ruled out by a smaller prime.
```



The first few times I read this it made no sense to me at all. It was just a jumble of numbers.

I stared at the ceiling, and closed my eyes to visualize. I couldn't see it. So I went on a long contemplative bike ride during which I realized that the prime multiples of 2 will at one point contain $2*3$ and then $2*5$. So the `multiples` array will at some point contain multiples of primes *larger* than the prime they represent. *And it clicked!*

Suddenly it all made sense. I realized that the `multiples` array was the equivalent of the array of booleans we use in the *Sieve of Eratosthenes* -- but with a really interesting twist. If you were to do the sieve on a whiteboard, you *could* erase every number less than the candidate, and only cross out the numbers that were the next multiples of all the previous primes.

That explanation makes perfect sense to me -- now, but I'd be willing to bet that those who are reading it are puzzling over it. The idea is just hard to explain.

Finally, I went back to your comment and could see what you were saying.

A Tale of Two Programmers

The bottom line here is that you and I both fell into the same trap. I refactored that old algorithm 18 years ago, and I thought all those method and variable names would make my intent clear -- *because I understood that algorithm*.

You wrote that code awhile back and decorated it with comments that you thought would explain your intent -- *because you understood that algorithm*.

But my names didn't help me 18 years later. They didn't help you, or your students either. And your comments didn't help me.

We were inside the box trying to communicate to those who stood outside and could not see what we saw.

The bottom line is that it is very difficult to explain something to someone who is not intimate with the details you are trying to explain. Often our explanations make sense only after the reader has worked out the details for themselves.

JOHN:

There's a lot of stuff in your discussion above, but I think it all boils down to one thing: you don't like the comments that I wrote. As I mentioned earlier, complexity is in the eye of the reader: if you say that my comments were confusing or didn't help you to understand the code, then I have to take that seriously.

At the same time, you have made it clear that you don't see much value in comments in general. Your preference is to have essentially no comments for this code (or any code). You argue above that there is simply nothing that comments can do to make the code easier to understand; the only way to understand the code is to read the code. That is a cop-out.

UB:

Sorry to interrupt you; but I think you are overstating my position. I certainly never said that comments can never be helpful. Sometimes, of course, they are. What I said was that I only trust them if the code validates them. Sometimes a comment will make that validation a lot easier.

JOHN:

You keep saying that you sometimes find use for comments, but the reality is that "sometimes" almost never occurs in your code. We'll see this when we look at your revision of my code.

Now back to my point. In order to write our various versions of the code, you and I had to accumulate a lot of knowledge about the algorithm, such as why it's OK for the first multiple of a prime to be its square. Unfortunately, not all of that knowledge can be represented in the code. It is our professional responsibility to do the best we can to convey that knowledge in comments, so that readers do not have to reconstruct it over and over. Even if the resulting comments are imperfect, they will make the code easier to understand.

If a situation like this occurred in real life I would work with you and others to improve my comments. For example, I would ask you questions to get a better sense of why the "squared prime" comment didn't seem to help you:

- Are there things in the comment that are misleading or confusing?
- Is there some important piece of information you acquired on your bike ride that suddenly made things clear?

I would also show the comment to a few other people to get their takes on it. Then I would rework the comment to improve it.

Given your fundamental disbelief in comments, I think it's likely that you would still see no value in the comment, even after my reworking. In this case I would show the comment to other people, particularly those who have a more positive view of comments in general, and get their input. As long as the comment is not misleading and at least a few people found it helpful, I would retain it.

Now let me discuss two specific comments that you objected to. The first comment was the one for the `multiples` variable:

```
// Used to test efficiently (without division) whether a candidate  
// is a multiple of a previously-encountered prime number. Each entry  
// here contains an odd multiple of the corresponding entry in  
// primes. Entries increase monotonically.
```



There is a bug in this comment that you exposed (the first entry is not odd); good catch! You then argued that most of the information in the comment is unnecessary and proposed this as an alternative:

```
// multiples of corresponding prime.
```



You have left out too much useful information here. For example, I don't think it is safe to assume that readers will figure out that the motivation is avoiding divisions. It's always better to state these assumptions and motivations clearly so that there will be no confusion. And I think it's helpful for readers to know that these entries never decrease. I would simply fix the bug, leaving all of the information intact:

```
// Used to test efficiently (without division) whether a candidate  
// is a multiple of a previously-encountered prime number. Each entry  
// (except the first, which is never used) contains an odd multiple of  
// the corresponding entry in primes. Entries increase monotonically.
```



The second comment was this one, for the `for` loop:

```
// Each iteration of this loop tests the candidate against one  
// potential prime factor. Skip the first factor (2) since we  
// only consider odd candidates.
```



You objected to this comment because the code of the loop doesn't actually test the candidate against the prime factor; it tests it against a multiple. When I write implementation comments like this, my goal is not to restate the code; comments like that don't usually provide much value. The goal here was to say *what* the code is doing in a logical sense, not *how* it does it. In that sense, the comment is correct.

However, if a comment causes confusion in the reader, then it is not a good comment. Thus, I would rewrite this comment to make it clear that it describes the abstract function of the code, not its precise behavior:

```
// Each iteration of this loop considers one existing prime, ruling  
// out the candidate if it is a multiple of that prime. Skip the
```



```
// first prime (2) since we only consider odd candidates.
```

To conclude, I agree with your assertion "it is very difficult to explain something to someone who is not intimate with the details you are trying to explain." And yet, it is our responsibility as programmers to do exactly that.

UB:

I'm glad we agree. We also agree about getting others to review the code and make recommendations on the code *and* the comments.

Bob's Rewrite of PrimeGenerator2

UB:

When I saw your solution, and after I gained a good understanding of it. I refactored it just a bit. I loaded it into my IDE, wrote some simple tests, and extracted a few simple methods.

I also got rid of that *awful* labeled `continue` statement. And I added 3 to the primes list so that I could mark the first element as *irrelevant* and give it a value of -1. (I think I was still reeling from the even/odd confusion.)

I like this because the implementation of the `generateFirstNPrimes` method describes the moving parts in a way that hints at what is going on. It's easy to read that implementation and get a glimpse of the mechanism. I'm not at all sure that the comment helps.

I think it is just the reality of this algorithm that the effort required to properly explain it, and the effort required for anyone else to read and understand that explanation is roughly equivalent to the effort needed to read the code and go on a bike ride.

```
package literatePrimes;
```



```
public class PrimeGenerator3 {  
    private static int[] primes;  
    private static int[] primeMultiples;  
    private static int lastRelevantMultiple;  
    private static int primesFound;  
    private static int candidate;
```

```
    // Lovely little algorithm that finds primes by predicting  
    // the next composite number and skipping over it. That predictio  
    // consists of a set of prime multiples that are continuously  
    // increased to keep pace with the candidate.
```

```
public static int[] generateFirstNPrimes(int n) {
    initializeTheGenerator(n);

    for (candidate = 5; primesFound < n; candidate += 2) {
        increaseEachPrimeMultipleToOrBeyondCandidate();
        if (candidateIsNotOneOfThePrimeMultiples()) {
            registerTheCandidateAsPrime();
        }
    }
    return primes;
}

private static void initializeTheGenerator(int n) {
    primes = new int[n];
    primeMultiples = new int[n];
    lastRelevantMultiple = 1;

    // prime the pump. (Sorry, couldn't resist.)
    primesFound = 2;
    primes[0] = 2;
    primes[1] = 3;

    primeMultiples[0] = -1; // irrelevant
    primeMultiples[1] = 9;
}

private static void increaseEachPrimeMultipleToOrBeyondCandidate(
    if (candidate >= primeMultiples[lastRelevantMultiple])
        lastRelevantMultiple++;

    for (int i = 1; i <= lastRelevantMultiple; i++)
        while (primeMultiples[i] < candidate)
            primeMultiples[i] += 2 * primes[i];
}

private static boolean candidateIsNotOneOfThePrimeMultiples() {
    for (int i = 1; i <= lastRelevantMultiple; i++)
        if (primeMultiples[i] == candidate)
            return false;
    return true;
}

private static void registerTheCandidateAsPrime() {
    primes[primesFound] = candidate;
    primeMultiples[primesFound] = candidate * candidate;
    primesFound++;
}
}
```

JOHN:

This version is a considerable improvement over the version in *Clean Code*. Reducing the number of methods made the code easier to read and resulted in cleaner interfaces. If it were properly commented, I think this version would be about as easy to read as my version (the additional methods you created didn't particularly help, but they didn't hurt either). I suspect that if we polled readers, some would like your version better and some would prefer mine.

Unfortunately, this revision of the code creates a serious performance regression: I measured a factor of 3-4x slowdown compared to either of the earlier revisions. The problem is that you changed the processing of a particular candidate from a single loop to two loops (the `increaseEach...` and `candidateIsNot...` methods). In the loop from earlier revisions, and in the `candidateIsNot` method, the loop aborts once the candidate is disqualified (and most candidates are quickly eliminated). However, `increaseEach...` must examine every entry in `primeMultiples`. This results in 5-10x as many loop iterations and a 3-4x overall slowdown.

Given that the whole reason for the current algorithm (and its complexity) is to maximize performance, this slowdown is unacceptable. The two methods must be combined.

I think what happened here is that you were so focused on something that isn't actually all that important (creating the tiniest possible methods) that you dropped the ball on other issues that really are important. We have now seen this twice. In the original version of `PrimeGenerator` you were so determined to make tiny methods that you didn't notice that the code was becoming incomprehensible. In this version you were so eager to chop up my single method that you didn't notice that you were blowing up the performance.

I don't think this was just an unfortunate combination of oversights. One of the most important things in software design is to identify what is important and focus on that; if you focus on things that are unimportant, you're likely to mess up the things that are important.

The code in your revision is still under-commented. You believe that there is no meaningful way for comments to assist the reader in understanding the code. I think this stems from your general disbelief in the value of comments; you are quick to throw in the towel. This algorithm is unusually difficult to explain, but I still believe that comments can help. For example, I believe you must make some attempt to help readers understand why the first multiple for a prime is the square of the prime. You have taken a lot of time to develop your understanding of this; surely there must be some way to convey that understanding to others? If you had included that information in your original version of the code you could have saved yourself that long bike ride. Giving up on this is an abdication of professional responsibility.

The few comments that you included in your revision are of little value. The first comment is too cryptic to provide much help: I can't make any sense of the phrase "predicting the next composite number and skipping over it" even though I completely understand the code it purports to explain. One of the comments is just a joke; I was surprised to see this, given your opposition to extraneous comments.

Clearly you and I live in different universes when it comes to comments.

Finally, I don't understand why you are offended by the labeled `continue` statement in my code. This is a clean and elegant solution to the problem of escaping from nested loops. I wish more languages had this feature; the alternative is awkward code where you set a variable, then exit one level of loop, then check the variable and exit the next level.

UB:

Good catch! I would have caught that too had I thought to profile the solution. You are right that separating the two loops added some unnecessary iteration. I found a nice way to solve that problem without using the horrible `continue`. My updated version is now faster than yours! A million primes in 440ms as opposed to yours which takes 561ms. ;-) Below are just the changes.

```
public static int[] generateFirstNPrimes(int n) {  
    initializeTheGenerator(n);  
  
    for (candidate = 5; primesFound < n; candidate += 2)  
        if (candidateIsPrime())  
            registerTheCandidateAsPrime();  
  
    return primes;  
}  
  
private static boolean candidateIsPrime() {  
    if (candidate >= primeMultiples[lastRelevantMultiple])  
        lastRelevantMultiple++;  
  
    for (int i = 1; i <= lastRelevantMultiple; i++) {  
        while (primeMultiples[i] < candidate)  
            primeMultiples[i] += 2 * primes[i];  
        if (primeMultiples[i] == candidate)  
            return false;  
    }  
    return true;  
}
```



JOHN:

Yep, that fixes the problem. I note that you are now down to 4 methods, from 8 in the *Clean Code* version.

Test-Driven Development

JOHN:

Let's move on to our third area of disagreement, which is Test-Driven Development. I am a huge fan of unit testing. I believe that unit tests are an indispensable part of the software development process and pay for themselves over and over. I think we agree on this.

However, I am not fan of Test-Driven Development (TDD), which dictates that tests must be written before code and that code must be written and tested in tiny increments. This approach has serious problems without any compensating advantages that I have been able to identify.

UB:

As I said at the start I have carefully read *A Philosophy of Software Design*. I found it to be full of worthwhile insights, and I strongly agree with most of the points you make.

So I was surprised to find, on page 157, that you wrote a very short, dismissive, pejorative, and inaccurate section on *Test Driven Development*. Sorry for all the adjectives, but I think that's a fair characterization. So my goal, here, is to correct the misconceptions that led you to write the following:

"Test-driven development is an approach to software development where programmers write unit tests before they write code. When creating a new class, the developer first writes unit tests for the class, based on its expected behavior. None of these tests pass, since there is no code for the class. Then the developer works through the tests one at a time, writing enough code for that test to pass. When all of the tests pass, the class is finished."

This is just wrong. TDD is quite considerably different from what you describe. I describe it using three laws.

1. You are not allowed to write any production code until you have first written a unit test that fails because that code does not exist.
2. You are not allowed to write more of a unit test than is sufficient to fail, and failing to compile is failing.
3. You are not allowed to write more production code than is sufficient to make the currently failing test pass.

A little thought will convince you that these three laws will lock you into a cycle that is just a few seconds long. You'll write a line or two of a test that will fail, you'll write a line or two of production code that will pass, around and around every few seconds.

A second layer of TDD is the Red-Green-Refactor loop. This loop is several minutes long. It is comprised of a few cycles of the three laws, followed by a period of reflection and refactoring. During that reflection we pull back from the intimacy of the quick cycle and look at the design of the code we've just written. Is it clean? Is it well-structured? Is there a better approach? Does it match the design we are pursuing? If not, should it?

JOHN:

Oops! I plead "guilty as charged" to inaccurately describing TDD. I will fix this in the next revision of APOSD. That said, your definition of TDD does not change my concerns.

Let's discuss the potential advantages and disadvantages of TDD; then readers can decide for themselves whether they think TDD is a good idea overall.

Before we start that discussion, let me clarify the approach I prefer as an alternative to TDD. In your online videos you describe the alternative to TDD as one where a developer writes the code, gets it fully working (presumably with manual tests), then goes back and writes the unit tests. You argue that this approach would be terrible: developers lose interest once they think code is working, so they wouldn't actually write the tests. I agree with you completely. However, this isn't the only alternative to TDD.

The approach I prefer is one where the developer works in somewhat larger units than in TDD, perhaps a few methods or a class. The developer first writes some code (anywhere from a few tens of lines to a few hundred lines), then writes unit tests for that code. As with TDD, the code isn't considered to be "working" until it has comprehensive unit tests.

UB:

How about if we call this technique "bundling" for purposes of this document? This is the term I use in *Clean Code 2d ed.*

JOHN:

Fine by me.

The reason for working in larger units is to encourage design thinking, so that a developer can think about a collection of related tasks and do a bit of planning to come up with a good overall design where the pieces fit together well. Of course the initial design ideas will have flaws and refactoring will still be necessary, but the goal is to center the development process around design, not tests.

To start our discussion, can you make a list of the advantages you think that TDD provides over the approach I just described?

UB: The advantages I usually attribute to TDD are:

- Very little need for debugging. After all, if you just saw everything working a minute or two ago, there's not much to debug.
- A stream of reliable low level documentation, in the form of very small and isolated unit tests. Those tests describe the low level structure and operation of every facet of the system. If you want to know how to do something in the system, there are tests that will show you how.
- A less coupled design which results from the fact that every small part of the system must be designed to be testable, and testability requires decoupling.
- A suite of tests that you trust with your life, and therefore supports fearless refactoring.

However, you asked me which of these advantages TDD might have over *your* preferred method. That depends on how big you make those larger units you described. The important thing to me is to keep the cycle time short, and to prevent entanglements that block testability.

It seems to me that working in small units, and then immediately writing after the fact tests, can give you all the above advantages, so long as you are very careful to test every aspect of the code you just wrote. I think a disciplined programmer could effectively work that way. Indeed, I think such a programmer would produce code that I could not distinguish from code written by another programmer following TDD.

Above you suggested that bundling is to encourage design. I think encouraging design is a very good thing. My question for you is: Why do you think that TDD does not encourage design? My own experience is that design comes from strategic thought, which is independent of the tactical behavior of either TDD or Bundling. Design is taking one step back from the code and envisioning structures that address a larger set of constraints and needs.

Once you have that vision in your head it seems to me bundling and TDD will yield similar results.

JOHN:

First, let me address the four advantages you listed for TDD:

- Very little need for debugging? I think any form of unit testing can reduce debugging work, but not for the reason you suggested. The benefit comes because unit tests expose bugs earlier and in an environment where they are easier to track down. A relatively simple bug to fix in development can be very painful to track down in production. I'm not convinced by your argument that there's less debugging because "you just saw everything working a minute ago": it's easy to make a tiny change that exposes a really gnarly bug that has existed for a long time but hasn't yet been triggered. Hard-to-debug problems arise from the accumulated complexity of the system, not from the size of the code increments.

UB: True. However, when the cycles are very short then the cause of even the gnarliest of bugs have the best chance of being tracked down. The shorter the cycles, the better the chances.

JOHN: This is only true up to a point. I think you believe that making units smaller and smaller continues to provide benefits, with almost no limit to how small they can get. I think that there is a point of diminishing returns, where making things even smaller no longer helps and actually starts to hurt. We saw this disagreement over method length, and I think we're seeing it again here.

- Low level documentation? I disagree: unit tests are a poor form of documentation. Comments are a much more effective form of documentation, and you can put them right next to the relevant code. Trying to learn a method's interface by reading a bunch of unit tests seems much more difficult than just reading a couple of sentences of English text.

UB: Nowadays it's very easy to find the tests for a function by using the "where-used" feature of the IDE. As for comments being better, if that were true then no one would publish example code.

- A less coupled design? Possibly, but I haven't experienced this myself. It's not clear to me that designing for testability will produce the best design.

UB: Generally the decoupling arises because the test requires a mock of some kind. Mocks tend to force abstractions that might otherwise not exist.

JOHN: In my experience, mocking virtually never changes interfaces; it just provides replacements for existing (typically immovable) interfaces.

UB: Our experiences differ.

- Enabling fearless refactoring? BINGO! This is the where almost all of the benefits from unit testing come from, and it is a really, really big deal.

UB: Agreed.

I agree with your conclusion that TDD and bundling are about the same in terms of providing these benefits.

Now let me explain why I think TDD is likely to result in bad designs. The fundamental problem with TDD is that it forces developers to work too tactically, in units of development that are too small; it discourages design thinking. With TDD the basic unit of development is one test: first the test is written, then the code to make that test pass. However, the natural units for design are larger than this: a class or method, for example. These units correspond to multiple test cases. If a developer thinks only about the next test, they are only considering part of a design problem at any given time. It's hard to design something well if you don't think about the whole design problem at once. TDD explicitly prohibits developers from writing more code than is needed to pass the current test; this discourages the kind of strategic thinking needed for good design.

TDD does not provide adequate guidance to encourage design. You mentioned the Red-Green-Refactor loop, which recommends refactoring after each step, but there's almost no guidance for refactoring. How should developers decide when and what to refactor? This seems to be left purely to their own judgment. For example, if I am writing a method that requires multiple iterations of the TDD loop, should I refactor after every iteration (which sounds pretty tedious) or wait until after several iterations so that I can look at a bigger chunk of code when refactoring and hence be more strategic? Without guidance, it will be tempting for developers to keep putting off refactoring.

TDD is similar to the One Thing Rule we discussed earlier in that it is biased: it provides very strong and clear instructions pushing developers in one direction (in this case, acting tactically) with only vague guidance in the other direction (designing more strategically). As a result, developers are likely to err on the side of being too tactical.

TDD guarantees that developers will initially write bad code. If you start writing code without thinking about the whole design problem, the first code you write will almost certainly be wrong. Design only happens after a bunch of bad code has accumulated. I watched your video on TDD, and you repeatedly wrote the wrong code, then fixed it later. If the developer refactors conscientiously (as you did) they can still end up with good code, but this works against human nature. With TDD, that bad code will actually work (there are tests to prove it!) and it's human nature not to want to change something that works. If the code I'm developing is nontrivial, I will probably have to accumulate a lot of bad code with TDD before I have enough code in front of me to understand what the design should have been. It will be very difficult for me to force myself to throw away all that work.

It's easy for a developer to believe they are doing TDD correctly while working entirely tactically, layering on hack after hack with an occasional minor refactor, without ever thinking about the overall design.

I believe that the bundling approach is superior to TDD because it focuses the development process around design: design first, then code, then write unit tests. Of course, refactoring will still be required: it's almost never possible to get the design right the first time. But starting with design will reduce the amount of bad code you write and get you to a good design sooner. It is possible to produce equally good designs with TDD; it's just harder and requires a lot more discipline.

UB:

I'll address your points one at a time.

- I haven't found that the scale of TDD is so tactical that it discourages thinking. Every programmer, regardless of their testing discipline, writes code one line at a time. That's immensely tactical and yet does not discourage design. So why would one test at a time discourage it?
- The literature on TDD strongly discourages delaying refactoring. While thinking about design is strongly encouraged. Both are integral parts of the discipline.
- We all write bad code at the start. The discipline of TDD gives us the opportunity, and the safety, to continuously clean it. Design insights arise from those kinds of cleaning activities. The discipline of refactoring allows bad designs to be transformed, one step at a time, into better designs.
- It's not clear to me why the act of writing tests late is a better design choice. There's nothing in TDD that prevents me from thinking through a design long before I write the very first tested code.

JOHN:

You say there is nothing about TDD that stops developers from thinking ahead about design. This is only partly true. Under TDD I can think ahead, but I can't actually write my ideas down in the form of code, since that would violate TDD Rule 1. This is a significant discouragement.

You claim that "thinking about design is strongly encouraged" in TDD, but I haven't seen this in your discussions of TDD. I watched your video example of using TDD for computing bowling scores, and design is never even mentioned after the first minute or two (ironically, one of the conclusions of this example is that the brief initial design turned out to be useless). There is no suggestion of thinking ahead in the video; it's all about cleaning up messes after the fact. In all of the TDD materials you have shown me, I have not seen any warnings about the dangers of becoming so tactical with TDD that design never occurs (perhaps you don't even view this as a serious risk?).

UB:

I usually use an abbreviated form of UML to capture my early design decisions. I have no objection to capturing them in pseudocode, or even real code. However, I would not commit any such pre-written code. I would likely hold it in a text file, and consult it while following the TDD cycle. I might feel safe enough to copy and paste from the text file into my IDE in order to make a failing test pass.

The Bowling game is an example of how wildly our initial design decisions can deviate from our eventual solutions. It's true that introductory videos often do not expose the depth of a discipline.

JOHN:

As I was watching your TDD video for the second time, you said something that jumped out at me:

Humans consider things that come first to be important and things that come at the end to be less important and somehow optional; that's why they are at the end, so we can leave them out if we have to.

This captures perfectly my concern about TDD. TDD insists that tests must come first, and design, if it happens at all, comes at the end, after code is working. I believe that good design is the most important thing, so it must be the top priority. I don't consider tests optional, but delaying them is safer than delaying design. Writing tests isn't particularly difficult; the most important thing is having the discipline to do it. Getting a good design is really hard, even if you are very disciplined; that's why it needs to be the center of attention.

UB:

TDD is a coding discipline. Of course design comes before coding -- I don't know anyone who thinks otherwise. Even the Bowling Game video made that point. But, as we saw in the Bowling Game video, sometimes the code will take you in a very different direction.

That difference doesn't imply that the design shouldn't have been done. It just implies that designs are speculative and may not always survive reality.

As Eisenhower once said:

“In preparing for battle I have always found that plans are useless, but planning is indispensable.”

JOHN:

You ask why writing tests later is a better design choice. It isn't. The benefit of the bundled approach doesn't come from writing tests later; it comes from doing design sooner. Writing tests (a bit) later is a consequence of this choice. The tests are still written pretty early-on with the bundled approach, so I don't think the delay causes significant problems.

UB:

I think we simply disagree that TDD discourages design. The practice of TDD does not discourage me from design; because I value design. I would suggest that those who do not value design will not design, no matter what discipline they practice.

JOHN:

You claim that the problems I worry about with TDD simply don't happen in practice. Unfortunately I have heard contrary claims from senior developers that I trust. They complain about horrible code produced by TDD-based teams, and they believe that the problems were caused by TDD. Of course horrible code can be produced with any design approach. And maybe those teams didn't implement TDD properly, or maybe those cases were outliers. But the problems reported to me line up exactly with what I would expect to happen, given the tactical nature of TDD.

UB:

My experience differs. I've worked on many projects where TDD has been used effectively and profitably. I'm sure the senior developers that you trust are telling you the truth about their experience. Having never seen TDD lead to such bad outcomes myself, I sincerely doubt that the blame can be traced to TDD.

JOHN:

You ask me to trust your extensive experience with TDD, and I admit that I have no personal experience with TDD. On the other hand, I have a lot of experience with tactical programming, and I know that it rarely ends well. TDD is one of the most extreme forms of tactical programming I've encountered. In general, if "making it work" is the #1 priority, instead of "develop a clean design", code turns to spaghetti. I don't see enough safeguards in your approach to TDD to prevent the disaster scenarios; I don't even see a clear recognition of the risk.

Overall, TDD is in a bad place on the risk-reward spectrum. In comparison to the bundling approach, the downside risks for poor code quality in TDD are huge, and I don't see enough upside reward (if any) to compensate.

UB:

All I can say to that is that your opinion is based on a number of false impressions and speculations, and not upon direct experience.

JOHN:

Now let me ask you a couple of questions.

First, at a microscopic level, why on earth does TDD prohibit developers from writing more code than needed to pass the current test? How does enforcing myopia make systems better?

UB: The goal of the discipline is to make sure that everything is tested. One good way to do that is to refuse to write any code unless it is to make a failing test pass. Also, working in such short cycles provides insights into the way the code is working. Those insights often lead to better design decisions.

JOHN: I agree that seeing code (partially) working can provide insights. But surely that benefit can be had without such a severe restriction on how developers think?

Second, at a broader level, do you think TDD is likely to produce better designs than approaches that are more design-centric, such as the bundling approach I described? If so, can you explain why?

UB: My guess is that someone adept at bundling, and someone adept at TDD would produce very similar designs, with very similar test coverage. I would also venture to guess that the TDDer would be somewhat more productive than the bundler if for no reason other than that the TDDer finds and fixes problems earlier than the bundler.

JOHN: I think that the bundling approach will result in a better design because it actually focuses on design, rather than focusing on tests and hoping that a good design will magically emerge. I think it's really hard to argue that the best way to achieve one thing is to focus your attention on something else. And the bundling approach will make progress faster because the early thinking about design will reduce the amount of bad code you end up having to throw away under TDD. Overall, I'd argue that the best-case outcomes for the two approaches will be about the same, but average and (especially) worst-case outcomes will be far worse for TDD.

JOHN:

I don't think we're going to resolve our disagreements on TDD. To do that, we'd need empirical data about the frequency of good and bad outcomes from TDD. Unfortunately I'm not aware of any such data. Thus, readers will have to decide for themselves whether the potential benefits of TDD outweigh the risks.

For anyone who chooses to use TDD, I urge you to do so with extreme caution. Your primary goal must not be just working code, but rather a clean design that will allow you to develop quickly in the future. TDD will not lead you naturally to the best design, so you will need to do significant and continuous refactoring to avoid spaghetti code. Ask yourself repeatedly "suppose that I knew everything I know now when I first started on this project; would I have chosen the current structure for the code?" When the answer is no (which will happen frequently) stop and refactor. Recognize that TDD will cause you to write more bad code than you may be used to, so you must be prepared to throw out and rewrite more than you are used to. Take time to plan ahead and think about the overall design, rather than just making the next test work. If you do all of these things diligently, I think it is possible to mitigate the risks of TDD and produce well-designed code.

UB:

Let's just say that I agree with all that advice, but disagree with your assertion that TDD might be the cause of bad code.

TDD Summary

JOHN:

Here is my attempt to summarize our thoughts on Test-Driven Development:

- We agree that unit tests are an essential element in software development. They allow developers to make significant changes to a system without fear of breaking something.
- We agree that it is possible to use TDD to produce systems with good designs.

- I believe that TDD discourages good design and can easily lead to very bad code. You do not believe that TDD discourages good design and don't see much of a risk of bad code.
- I believe that there are better approaches than TDD for producing good unit test suites, such as the "bundling" approach discussed above. You agree that bundling can produce outcomes just as good as TDD but think it may lead to somewhat less test coverage.
- I believe that TDD and bundling have similar best-case outcomes, but that the average and worst-case outcomes will be much worse for TDD. You disagree and believe that, if anything, TDD may produce marginally better outcomes than bundling. You also think that preference and personality are larger factors in making the choice between the two.

UB:

This is a fair summary of our discussion. We seem to disagree over the best application of discipline. I prefer a disciplined approach to keep the code covered by tests written first in very short cycles. You prefer a disciplined approach of writing relatively longer bundles of code and then writing tests for those bundles. We disagree on the risks and rewards of these two disciplines.

Closing Remarks

JOHN:

First, I'd like to thank you for tolerating (and responding to) the arguments I have made about some of the key ideas in *Clean Code*. I hope this discussion will provide food for thought for readers.

We have covered a lot of topics and subtopics in this discussion, but I think that most of my concerns result from two general errors made by *Clean Code*: failure to focus on what is important, and failure to balance design tradeoffs.

In software design (and probably in any design environment) it is essential to identify the things that really matter and focus on those. If you focus your attention on things that are unimportant you are unlikely to achieve the things that really are important. Unfortunately, *Clean Code* repeatedly focuses on things that don't really matter, such as:

- Dividing ten-line methods into five-line methods and dividing five-line methods into two- or three-line methods.
- Eliminating the use of comments written in English.

- Writing tests before code and making the basic unit of development a test rather than an abstraction.

None of these provides significant value, and we have seen how they distract from producing the best possible designs.

Conversely, *Clean Code* fundamentally undervalues comments, which are essential and irreplaceable. This comes at a huge cost. Without interface comments the specifications for interfaces are incomplete. This is guaranteed to result in confusion and bugs. Without implementation comments, readers are forced to rederive knowledge and intentions that were in the mind of the original developer. This wastes time and leads to more bugs.

In my opening remarks I said that systems become complex when important information is not accessible and obvious to developers. By refusing to write comments, you are hiding important information that you have and that others need.

The second general error in *Clean Code* has to do with balance. Design represents a balance between competing concerns. Almost any design idea becomes a bad thing if taken to the extreme. However, *Clean Code* repeatedly gives very strong advice in one direction without correspondingly strong advice in the other direction or any meaningful guidance about how to recognize when you have gone too far. For example, making methods shorter is often a good thing, but the *Clean Code* position is so one-sided and extreme that readers are likely to chop things up too much. We saw in the `PrimeGenerator` example how this resulted in code that was nearly incomprehensible. Similarly, the *Clean Code* position on TDD is one-sided, failing to recognize any possible weakness and encouraging readers to take this to a tactical extreme where design is completely squeezed out of the development process.

UB:

John, I'd like to thank you for participating in this project. This was a lot of fun for me. I love disagreement and debate with smart people. I also think that we share far more values than separate us.

For my part I'll just say that I have given due consideration to the points you've made, and while I disagree with your conclusions above, I have integrated several of your better ideas, as well as this entire document, into the second edition of *Clean Code*.

Thanks again, and give my best to your students.