The Pragmatic Programmers

Your Code as a Crime Scene

Second Edition

Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs

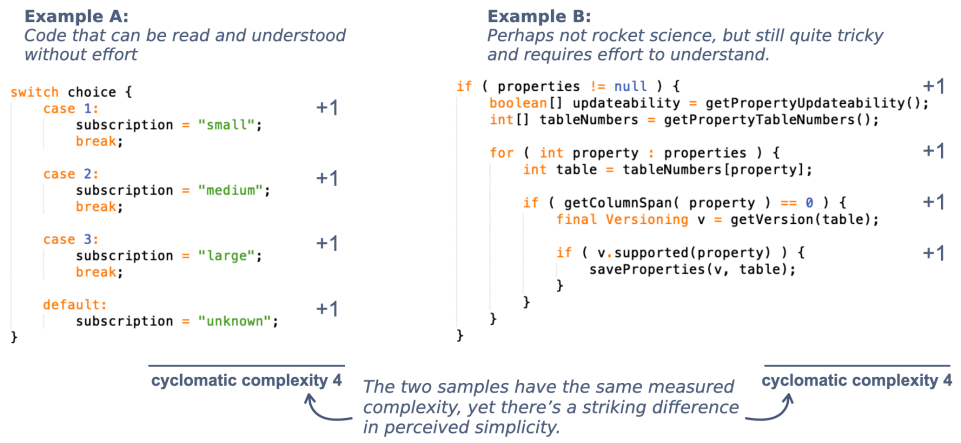
Adam Tornhill

Foreword by Kevlin Henney

Edited by Kelly Talbot

Spot Nested Logic That Taxes Working Memory

In Explore the Complexity Dimension, on page ? you saw that the most popular code-level metrics are poor at capturing accidental complexity. The main reason is that cyclomatic complexity doesn't reveal much about the relative effort required to understand a piece of code. The metric cannot differentiate between code with repeatable patterns that are straightforward to reason about vs. truly messy implementations requiring cognitive effort to understand. Let's look at the next figure to see how misleading cyclomatic complexity can be.


Example A:
Code that can be read and understood without effort
Example B:
Perhaps not rocket science, but still quite tricky and requires effort to understand.
+1
+1
+1
+1
+1
+1
+1
+1
cyclomatic complexity 4
The two samples have the same measured complexity, yet there's a striking difference in perceived simplicity.
cyclomatic complexity 4

```
switch choice {
    case 1:
        subscription = "small";
        break;

    case 2:
        subscription = "medium";
        break;

    case 3:
        subscription = "large";
        break;

    default:
        subscription = "unknown";
}
```

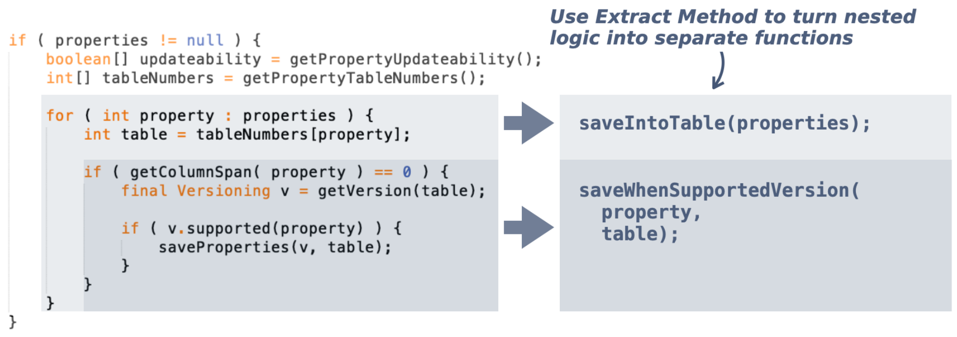
```
if ( properties != null ) {
    boolean[] updateability = getPropertyUpdateability();
    int[] tableNumbers = getPropertyTableNumbers();

    for ( int property : properties ) {
        int table = tableNumbers[property];

        if ( getColumnSpan( property ) == 0 ) {
            final Versioning v = getVersion(table);

            if ( v.supported(property) ) {
                saveProperties(v, table);
            }
        }
    }
}
```

Instead, when on the hunt for complexity, consider the shape of the code, as you learned in Calculate Complexity Trends from Your Code's Shape, on page ?. Code with deep, nested logic heavily taxes your working memory.

To experience the problem, look at case B in the preceding figure and pretend you need to change the `saveProperties()` call. When making this change, all preceding `if` statements represent the program state you need to keep in your head. There are four branches in the code, meaning you're operating at the edge of your cognitive capacity while reasoning about this code, and you still need to find some mental room for the logic of the actual change. It's no wonder that things go wrong in nested code.

The cognitive costs of nested logic were confirmed in a comprehensive survey of software engineers at seven leading Swedish companies (such as Volvo, Ericsson, Axis). This survey reveals that nesting depth and a general lack of structure are the two issues that introduce the most perceived complexity when reading code, significantly more than the number of conditionals by itself. (See *A Pragmatic View on Code Complexity Management [ASS19]* for the full treatment.)

Fortunately, nested logic is straightforward to refactor by encapsulating each nested block within a well-named function, as shown in the next figure.


Use Extract Method to turn nested logic into separate functions

```
if ( properties != null ) {
    boolean[] updateability = getPropertyUpdateability();
    int[] tableNumbers = getPropertyTableNumbers();

    for ( int property : properties ) {
        int table = tableNumbers[property];

        if ( getColumnSpan( property ) == 0 ) {
            final Versioning v = getVersion(table);

            if ( v.supported(property) ) {
                saveProperties(v, table);
            }
        }
    }
}
```

```
saveIntoTable(properties);

saveWhenSupportedVersion(
  property,
  table);
```

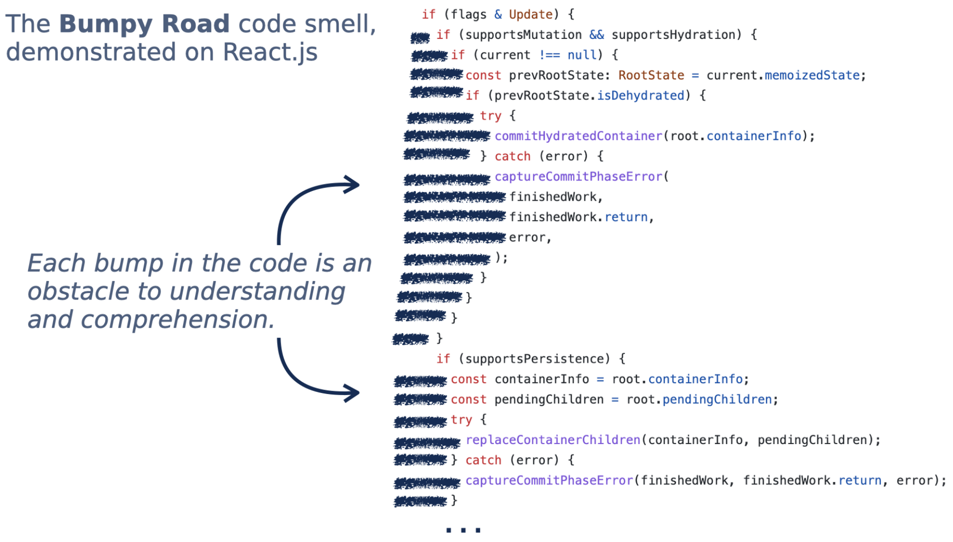
Stay Clear of Bumpy Roads

Nested logic might be problematic, but, as always, there are different levels in complexity hell. A more serious issue is the *bumpy road* code smell, nested logic's sinister cousin.

The bumpy road smell is a function with multiple chunks of nested conditional logic. Just like a bumpy road slows down your driving speed and comfort, a bumpy road in code presents an obstacle to comprehension. Worse, in imperative languages, there's also the increased risk of feature entanglement, leading to complex state management with bugs in its wake.

Bumpy roads are prevalent in code (it's what you get if you cut down on maintenance), and you'll find the issue in many hotspots independent of programming language. An illustrative example is the JavaScript function commitMutationEffectsOnFiber in React's ReactFiberCommitWork.old.js hotspot.[5] The next figure shows a small slice of the code.

5. https://tinyurl.com/react-fiber-code2202


The **Bumpy Road** code smell, demonstrated on React.js
Each bump in the code is an obstacle to understanding and comprehension.

```
if (flags & Update) {
  if (supportsMutation && supportsHydration) {
    if (current !== null) {
      const prevRootState: RootState = current.memoizedState;
      if (prevRootState.isDehydrated) {
        try {
          commitHydratedContainer(root.containerInfo);
        } catch (error) {
          captureCommitPhaseError(
            finishedWork,
            finishedWork.return,
            error,
          );
        }
      }
    }
  }
  if (supportsPersistence) {
    const containerInfo = root.containerInfo;
    const pendingChildren = root.pendingChildren;
    try {
      replaceContainerChildren(containerInfo, pendingChildren);
    } catch (error) {
      captureCommitPhaseError(finishedWork, finishedWork.return, error);
    }
    ...
```

When inspecting bumpy roads, there's a set of heuristics for classifying the severity of the code smell:

- The deeper the nested logic in each bump, the higher the tax on working memory.
- The more bumps, the more expensive it is to refactor since each bump represents a missing abstraction.
- The larger the bumps—that is, the more lines of code they span—the harder it is to build up a mental model of the function.

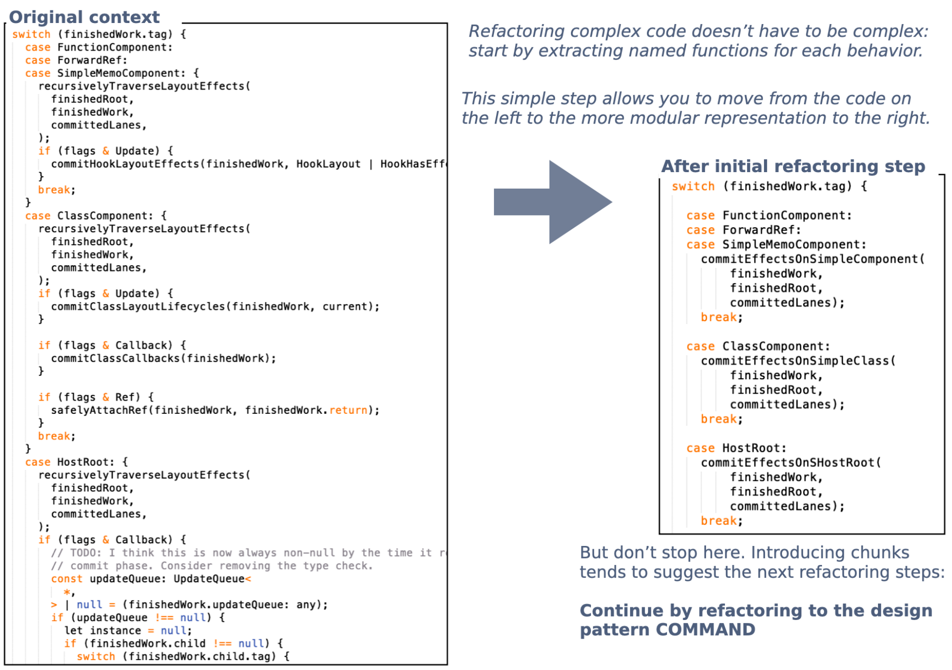
Fundamentally, a bumpy code road represents a lack of encapsulation. Each bump tends to represent a responsibility or action. Hence, the initial remediation is the same as for deep nested logic: extract functions corresponding to the identified responsibilities to even out the road.

Before we continue our tour of code smells, let's take a step back and discuss method extraction. It's such a critical refactoring in laying the foundation for better designs to come.

Refactor Complex Code via Simple Steps

The code smells seen so far all stem from overly long classes and functions. Take ReactFiberCommitWork as a prominent example: its three central functions span 200 to 400 lines of code each. This raises the question: should we prefer many small methods—the logical outcome of the recommended refactorings—or is it better to keep related code in one large chunk?

The main advantage of modularizing a piece of code isn't that we get shorter functions. Rather, it's about transforming the context, as discussed in design to isolate change on page ?. By extracting cohesive, well-named functions, we introduce chunks into our design, as shown in the following figure. These chunks let us reason more effectively about the problem we're trying to solve, often suggesting more radical refactoring in the process.



Original context

```
switch (finishedWork.tag) {
  case FunctionComponent:
  case ForwardRef:
  case SimpleMemoComponent: {
    recursivelyTraverseLayoutEffects(
      finishedRoot,
      finishedWork,
      committedLanes,
    );
    if (flags & Update) {
      commitHookLayoutEffects(finishedWork, HookLayout | HookHasEff
    }
    break;
  }
  case ClassComponent: {
    recursivelyTraverseLayoutEffects(
      finishedRoot,
      finishedWork,
      committedLanes,
    );
    if (flags & Update) {
      commitClassLayoutLifecycles(finishedWork, current);
    }

    if (flags & Callback) {
      commitClassCallbacks(finishedWork);
    }

    if (flags & Ref) {
      safelyAttachRef(finishedWork, finishedWork.return);
    }
    break;
  }
  case HostRoot: {
    recursivelyTraverseLayoutEffects(
      finishedRoot,
      finishedWork,
      committedLanes,
    );
    if (flags & Callback) {
      // TODO: I think this is now always non-null by the time it r
      // commit phase. Consider removing the type check.
      const updateQueue: UpdateQueue<
        *,
      > | null = (finishedWork.updateQueue: any);
      if (updateQueue !== null) {
        let instance = null;
        if (finishedWork.child !== null) {
          switch (finishedWork.child.tag) {
```

Refactoring complex code doesn't have to be complex: start by extracting named functions for each behavior.

This simple step allows you to move from the code on the left to the more modular representation to the right.

After initial refactoring step

```
switch (finishedWork.tag) {

  case FunctionComponent:
  case ForwardRef:
  case SimpleMemoComponent:
    commitEffectsOnSimpleComponent(
        finishedWork,
        finishedRoot,
        committedLanes);
    break;

  case ClassComponent:
    commitEffectsOnSimpleClass(
        finishedWork,
        finishedRoot,
        committedLanes);
    break;

  case HostRoot:
    commitEffectsOnSHostRoot(
        finishedWork,
        finishedRoot,
        committedLanes);
    break;
```

But don't stop here. Introducing chunks tends to suggest the next refactoring steps:

Continue by refactoring to the design pattern COMMAND

The preceding figure shows how introducing chunks in a complex hotspot reveals the overall intent of the code. The original implementation is high in complexity, with deep, nested logic paving an uncomfortably bumpy road. The mere act of extracting relevant methods delivers multiple benefits:

- *Isolate change*—The entanglement of the original code is like an open invitation to bugs and coding mistakes; we change one behavior, only to discover that we broke another seemingly unrelated feature. Modularizing the code serves to protect different features from each other. As a bonus, increased modularity clarifies the data dependencies.
- *Guide code reading*—Cohesive functions guide code reading since there's one obvious place to go for details on how a specific business rule is realized.

- *Reveal intent*—Extracting functions brings out the overall algorithm, which in turn suggests the deeper design changes that make the real difference. In the preceding example, the code starts to look like a match for the *command pattern* (See *Design Patterns: Elements of Reusable Object-Oriented Software [GHJV95]*.) Replacing each case statement with an object would increase the cohesion and make the bulk of the function go away. There's nothing sweeter than deleting accidental complexity.

These refactoring steps and benefits can come across as overly simplistic at first. It's like we expect complex problems to always require complicated solutions. Yet, the simplicity of a refactoring like the Extract Method is deceiving since modeling the *right* behaviors and chunks is far from trivial. Refactoring requires domain expertise for finding the right abstractions and properly naming them.

Recognize Bad Names

Back in Understand That Typing Isn't the Bottleneck in Programming, on page ?, you learned that we spend most of our time trying to understand existing code. How we name our chunks of code is vital in that program comprehension process. As research shows, we try to infer the purpose of unfamiliar code by building up mental representations largely driven by reading the names of classes or functions. (See *Software Design: Cognitive Aspects [DB13]* for the empirical findings.)

This implies that a good name is descriptive and expresses intent. A good name also suggests a cohesive concept; remember, fewer responsibilities means fewer reasons to change. Bad names, on the other hand, are recognized by the following characteristics:

- Bad names carry little information and convey no hints to the purpose of the module, for example, StateManager (isn't state management what programming is about?) and Helper (a helper for what and whom?).

- A bad name is built with conjunctions, such as and, or, and so on. These are sure signs of low cohesion. Examples include ConnectionAndSessionPool (do connections and sessions express the same concept?) and FrameAndToolbarController (do the same rules really apply to both frames and toolbars?).

Bad names attract suffixes like lemonade draws wasps on a hot summer day. The immediate suspects are everything that ends with Manager, Util, or the dreaded Impl. Modules baptized like that are typically placeholders, but they end up housing core logic elements over time. You know they will hurt once you look inside.

Naming Object-Oriented Inheritance Hierarchies

Good interfaces express intent and suggest usage. Their implementations specify both what's specific and what's different about the concrete instances. Say we create an intention-revealing interface: `ChatConnection`. (Yes, I did it—I dropped the cognitive distractor, the `I` prefix.) Let each implementation of this interface specify what makes it unique: `SynchronousTcpChatConnection`, `AsynchronousTcpChatConnection`, and so on.

Optimize for Your Brain, Not a Metric

Code smells, like large functions, complex logic, fuzzy names, and bumpy roads, are mere symptoms of an underlying problem: lack of encapsulation. Refactoring complex code is very much an iterative process. Start simple and reduce error-prone constructs step-by-step to align the code with how your brain prefers it. And remember that modularization is a start, not the end.

Never base the decision to split a method on length but on behavior and meaningful abstractions. Splitting functions based on thresholds alone makes the code worse, not better; code that belongs together should stay together. Always.

That said, a general heuristic like "max 30 lines of code per function," as recommended by David Farley in *Modern Software Engineering [Far22]*, still serves well as an alert system: the longer the method, the more likely it's lacking in abstraction. Just remember to treat the limit as the heuristic it is, and optimize for your brain, not a measure.