2

Clean That Code!

Clean code! Is that a noun? Or is it a command? Is the word *clean* an adjective or a verb? In this chapter, and in the book overall, we take the latter view. This is a chapter and a book about cleaning code.

One does not simply write clean code. To create code that is clean one must clean it. Code does not pour from your mind and out through your fingers in a clean state. That's not the way our brains work. This is because cleanliness and functionality are competing orthogonal concerns that we, mere humans, are not particularly good at balancing. Instead, our limited

minds first focus upon the concern of functionality as we strive to produce code that *works*. It is only when that goal has been achieved that we can change our focus toward cleaning the mess we just made.

Or again, as Kent Beck so succinctly put it:

First, make it work. Then, make it right.

When we do the first part—make it work—we also usually make a mess. This is because making it work takes all of our cognitive energy, and there is none left over to think about anything else.

Once we have something that works, then we can pull back, look it over, and think of ways to improve it.

You've all had the experience of looking over code that you wrote in the past—perhaps only a few days or hours in the past—and then realized that it could be improved by this or that change.

You've also, very likely, had the same experience with just about anything you've ever done. If you've ever written a paper, or a story, or an article, you likely created rough drafts and then passed over them once, or twice, or thrice, to improve them. If you've ever drawn a picture, you've likely spent a lot of time using your eraser. Indeed, I have reread the last few paragraphs several times and have made small improvements each time.

This is how we humans work. We produce something fairly ugly at first, and then we massage and manipulate it until it is better.

Perhaps an example will help. I wrote a simple[1] converter from Roman numerals to integers. I did this, with the help of GitHub Copilot, by simply throwing code into a function and fumbling around until I thought it worked. I had some unit tests that Copilot and I likewise threw together, but I was frankly not sure they covered all the cases. Here's the code. I expect that it will make your eyes cross, and cause the blood to drain from your face as you stare in horror.

---

1. By some definition of that word.

Listing 2-1

```
package fromRoman;

import java.util.Arrays;

public class FromRoman {

  public static int convert(String roman) {
    if (roman.contains("VIV") ||
      roman.contains("IVI") ||
      roman.contains("IXI") ||
      roman.contains("LXL") ||
      roman.contains("XLX") ||
      roman.contains("XCX") ||
      roman.contains("DCD") ||
      roman.contains("CDC") ||
      roman.contains("MCM")) {
      throw new InvalidRomanNumeralException(roman);
    }
    roman = roman.replace("IV", "4");
    roman = roman.replace("IX", "9");
    roman = roman.replace("XL", "F");
    roman = roman.replace("XC", "N");
    roman = roman.replace("CD", "G");
    roman = roman.replace("CM", "O");
    if (roman.contains("IIII") ||
      roman.contains("VV") ||
      roman.contains("XXXX") ||
      roman.contains("LL") ||
      roman.contains("CCCC") ||
      roman.contains("DD") ||
      roman.contains("MMMM")) {
      throw new InvalidRomanNumeralException(roman);
    }

    int[] numbers = new int[roman.length()];
    int i = 0;
    for (char digit : roman.toCharArray()) {
      switch (digit) {
        case 'I' -> numbers[i] = 1;
        case 'V' -> numbers[i] = 5;
        case 'X' -> numbers[i] = 10;
        case 'L' -> numbers[i] = 50;
```

```
            case 'C' -> numbers[i] = 100;
            case 'D' -> numbers[i] = 500;
            case 'M' -> numbers[i] = 1000;
            case '4' -> numbers[i] = 4;
            case '9' -> numbers[i] = 9;
            case 'F' -> numbers[i] = 40;
            case 'N' -> numbers[i] = 90;
            case 'G' -> numbers[i] = 400;
            case 'O' -> numbers[i] = 900;
            default -> throw new InvalidRomanNumeralExcep
          }
          i++;
        }
        int lastDigit = 1000;
        for (int number : numbers) {
          if (number > lastDigit) {
            throw new InvalidRomanNumeralException(roman)
          }
          lastDigit = number;
        }
        return Arrays.stream(numbers).sum();
      }

      public static
      class InvalidRomanNumeralException extends RuntimeE
        public InvalidRomanNumeralException(String roman)
        }
      }
    }
```


Future Bob:

I'm reading this code two months later, and I'm laughing hysterically at just how awful some of it is (like `49FNGO`)… . And yet how insightful parts of it are.

Here are the tests I used. I had my doubts about whether they were complete.

```
package fromRoman;
```

```
import fromRoman.FromRoman.InvalidRomanNumeralExcepti
import org.junit.jupiter.api.Test;

import static fromRoman.FromRoman.convert;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import static org.junit.jupiter.api.Assertions.assert

public class FromRomanTest {
  @Test
  public void valid() throws Exception {
    assertThat(convert(""), is(0));
    assertThat(convert("I"), is(1));
    assertThat(convert("II"), is(2));
    assertThat(convert("III"), is(3));
    assertThat(convert("IV"), is(4));
    assertThat(convert("V"), is(5));
    assertThat(convert("VI"), is(6));
    assertThat(convert("VII"), is(7));
    assertThat(convert("VIII"), is(8));
    assertThat(convert("IX"), is(9));
    assertThat(convert("X"), is(10));
    assertThat(convert("XI"), is(11));
    assertThat(convert("XII"), is(12));
    assertThat(convert("CXC"), is(190));
  }

  @Test
  public void invalid() throws Exception {
    assertInvalid("IIII");
    assertInvalid("VV");
    assertInvalid("XXXX");
    assertInvalid("LL");
    assertInvalid("CCCC");
    assertInvalid("DD");
    assertInvalid("MMMM");
    assertInvalid("IIV");
    assertInvalid("IVI");
    assertInvalid("IXI");
    assertInvalid("VIV");
    assertInvalid("XVX");
    assertInvalid("XIIII");
    assertInvalid("XVV");
    assertInvalid("XIVI");
    assertInvalid("XIXI");
```

```
    assertInvalid("XVIV");
    assertInvalid("LXL");
    assertInvalid("XLX");
    assertInvalid("XCX");
    assertInvalid("CDC");
    assertInvalid("DCD");
    assertInvalid("MCMC");
    assertInvalid("MCDM");
  }

  private void assertInvalid(String r) {
    assertThrows(InvalidRomanNumeralException.class,
  }
}
```

I hope you agree with me that this needs some cleanup. It feels ragged and jumbled. Later in this chapter, and in the chapters to follow, I'll walk through the cleaning procedure. But for now, I just want to show you the result.

Before I present the result, I want to say that the time it took to clean the code was roughly equal to the time it took to write the code in the first place. Copilot was moderately helpful during the initial writing but significantly less helpful during the cleaning process.

During the cleaning process, I found one or two small bugs simply through the effort of organizing my thoughts and the structure of the code and tests. It's remarkable what a second look can find[2]—and cleaning is the ultimate second look.

---

2. After I conduct the close-up preflight inspection on my airplane, I stand back, walk around the plane, and take a second look from a farther perspective. It's remarkable what I've found with that approach. There are things that are obvious from a distance that are nearly invisible during closer inspection.

```
package fromRoman;

import java.util.ArrayList;
import java.util.List;
```

```
import java.util.Map;

public class FromRoman {
  private String roman;
  private List<Integer> numbers = new ArrayList<>();
  private int charIx;
  private char nextChar;
  private Integer nextValue;
  private Integer value;
  private int nchars;
  Map<Character, Integer> values = Map.of(
    'I', 1,
    'V', 5,
    'X', 10,
    'L', 50,
    'C', 100,
    'D', 500,
    'M', 1000);

  public FromRoman(String roman) {
    this.roman = roman;
  }

  public static int convert(String roman) {
    return new FromRoman(roman).doConversion();
  }

  private int doConversion() {
    checkInitialSyntax();
    convertLettersToNumbers();
    checkNumbersInDecreasingOrder();
    return numbers.stream().reduce(0, Integer::sum);
  }

  private void checkInitialSyntax() {
    checkForIllegalPrefixCombinations();
    checkForImproperRepetitions();
  }

  private void checkForIllegalPrefixCombinations() {
    checkForIllegalPatterns(
      new String[]{"VIV", "IVI", "IXI", "IXV", "LXL",
                   "XCX", "XCL", "DCD", "CDC", "CMC",
  }
```

```
private void checkForImproperRepetitions() {
  checkForIllegalPatterns(
    new String[]{"IIII", "VV", "XXXX", "LL", "CCCC"
}

private void checkForIllegalPatterns(String[] patte
  for (String badString : patterns)
    if (roman.contains(badString))
      throw new InvalidRomanNumeralException(roman)
}

private void convertLettersToNumbers() {
  char[] chars = roman.toCharArray();
  nchars = chars.length;
  for (charIx = 0; charIx < nchars; charIx++) {
    nextChar = isLastChar() ? 0 : chars[charIx + 1]
    nextValue = values.get(nextChar);
    char thisChar = chars[charIx];
    value = values.get(thisChar);
    switch (thisChar) {
      case 'I' -> addValueConsideringPrefix('V', 'X
      case 'X' -> addValueConsideringPrefix('L', 'C
      case 'C' -> addValueConsideringPrefix('D', 'M
      case 'V', 'L', 'D', 'M' -> numbers.add(value)
      default -> throw new InvalidRomanNumeralExcep
    }
  }
}

private boolean isLastChar() {
  return charIx + 1 == nchars;
}

private void addValueConsideringPrefix(char p1, cha
  if (nextChar == p1 || nextChar == p2) {
    numbers.add(nextValue - value);
    charIx++;
  } else numbers.add(value);
}

private void checkNumbersInDecreasingOrder() {
  for (int i = 0; i < numbers.size() - 1; i++)
    if (numbers.get(i) < numbers.get(i + 1))
      throw new InvalidRomanNumeralException(roman)
}
```

```
    public static
    class InvalidRomanNumeralException extends RuntimeE
      public InvalidRomanNumeralException(String roman)
        super("Invalid Roman numeral: " + roman);
      }
    }
  }
```

I think this is much cleaner. What makes it cleaner, in my opinion, are the short, well-named functions, the well-named instance variables, and the fact that the functions are listed in the order that they are called. You can read the code from the top to the bottom and it reads rather like a story.

Future Bob:

Two months later I'm torn. The first version, ugly as it was, was not as chopped up as this one. It's true that the names and the ordering of the extracted functions read like a story and are a big help in understanding the intent; but there were several times that I had to scroll back up to the top to assure myself about the types of instance variables. I found the choppiness, and the scrolling, to be annoying. However, and this is critical, I am reading this cleaned code after having first read the ugly version and having gone through the work of understanding it. So now, as I read this version, I am annoyed because I already understand it and find the chopped-up functions and the instance variables redundant.

Don't get me wrong, I still think the cleaner version is better. I just wasn't expecting the annoyance. When I first cleaned it, I thought it was going to be annoyance free.

I suppose the question you should ask yourself is which of these two pieces of code you would rather have read *first*. Which tells you more about the intent? Which obscures the intent? Certainly the latter is better in that regard.

This annoyance is an issue that John Ousterhout[3] and I have debated.[4] When you understand an algorithm, the artifacts intended to help you

understand it become annoying. Worse, if you understand an algorithm, the names or comments you write to help others will be biased by that understanding and may not help the reader as much as you think they will. A good example of that, in this code, is the `addValueConsideringPrefix` function. That name made perfect sense to me when I understood the algorithm. But it was a bit jarring two months later. Perhaps not as jarring as `49FNGO`, but still not quite as obvious as I had hoped when I wrote it. It might have been better written as `numbers.add (decrementValueIfThisCharIsaPrefix…);`, since that would be symmetrical with the `numbers.add(value);` in the nonprefixed case.

---

3. [APOD].

---

4. See the appendix to read that debate.

The bottom line is that your own understanding of what you are cleaning will work against your ability to communicate with the next person to come along. And this will be true whether you are extracting well-named methods, or adding descriptive comments. Therefore, take special care when choosing names and writing comments; and don't be surprised if others are annoyed by your choices. Lastly, a look after a few months can be both humbling and profitable.

The `convertLettersToNumbers` function is a bit longer than I like; but it hangs together pretty well, and I think that decomposing it into smaller functions would be more obscuring than helpful.

Some of you may be looking at this "cleaner" code with a more skeptical eye. Perhaps you have some questions or complaints.

You might complain that it's longer. That's true; but most of that extra length is due to the nicely named functions that explain what's going on.

You might complain that the creation of an object is overkill and expensive. It's true that allocating memory, and then garbage-collecting it later, takes extra time. There is also time spent in the calling of all those small methods. If you are writing embedded real-time code with very tight timing constraints, or if you are writing code for a high-performance flight

simulator or first-person shooter game, then this decision could seem insane, not to mention unclean. But for my purposes, the extra time is the least of my concerns. I don't need the speed. For my situation, I think trading speed for clarity, within reason, is a good trade to make.

You might be a functional programmer horrified that the functions are not "pure." But, in fact, the static `convert` function is as pure as a function can be. The others are just little helpers that operate within a single invocation of that overarching pure function. Those instance variables are very convenient for allowing the individual methods to communicate without having to resort to passing arguments. This shows that one good use for an object is to allow the helper functions that operate within the execution of a pure function to easily communicate through the instance variables of the object.

So I think this is pretty clean. Am I right? I think I am; but you may disagree. That's OK. We are allowed to disagree. There is no single standard for what cleanliness is. This is mine, and mine may not be yours. All I'll ask of you is that you consider mine and develop your own standard. The most important thing is to have a standard that you apply with discipline.

If you are a member of a team, then the team should have a single standard[5] of cleanliness that is negotiated, and then followed and supported, by all members of that team.

---

5. This might tempt you to create a standards document. I recommend otherwise. The document that represents the standard should be the code itself. The only reason to have a standards document outside of the code is if the code does not currently represent the standard.

By the way, the tests I initially wrote *were* inadequate. As I worked through the cleaning process, I found a number of situations that needed further testing. For the sake of completeness, I've included the final tests below.

```
package fromRoman;

import fromRoman.FromRoman.InvalidRomanNumeralExcepti
```

```
import org.junit.jupiter.api.Test;

import static fromRoman.FromRoman.convert;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import static org.junit.jupiter.api.Assertions.assert

public class FromRomanTest {
  @Test
  public void valid() throws Exception {
    assertThat(convert(""), is(0));
    assertThat(convert("I"), is(1));
    assertThat(convert("II"), is(2));
    assertThat(convert("III"), is(3));
    assertThat(convert("IV"), is(4));
    assertThat(convert("V"), is(5));
    assertThat(convert("VI"), is(6));
    assertThat(convert("VII"), is(7));
    assertThat(convert("VIII"), is(8));
    assertThat(convert("IX"), is(9));
    assertThat(convert("X"), is(10));
    assertThat(convert("XI"), is(11));
    assertThat(convert("XII"), is(12));
    assertThat(convert("XIII"), is(13));
    assertThat(convert("XIV"), is(14));
    assertThat(convert("XV"), is(15));
    assertThat(convert("XVI"), is(16));
    assertThat(convert("XIX"), is(19));
    assertThat(convert("XX"), is(20));
    assertThat(convert("XXX"), is(30));
    assertThat(convert("XL"), is(40));
    assertThat(convert("L"), is(50));
    assertThat(convert("LX"), is(60));
    assertThat(convert("LXXIV"), is(74));
    assertThat(convert("XC"), is(90));
    assertThat(convert("C"), is(100));
    assertThat(convert("CXIV"), is(114));
    assertThat(convert("CXC"), is(190));
    assertThat(convert("CD"), is(400));
    assertThat(convert("D"), is(500));
    assertThat(convert("CDXLIV"), is(444));
    assertThat(convert("DCXCIV"), is(694));
    assertThat(convert("CM"), is(900));
    assertThat(convert("M"), is(1000));
    assertThat(convert("MCM"), is(1900));
```

```
  assertThat(convert("MCMXCIX"), is(1999));
  assertThat(convert("MMXXIV"), is(2024));
}

@Test
public void invalid() throws Exception {
  assertInvalid("IIII");
  assertInvalid("VV");
  assertInvalid("XXXX");
  assertInvalid("LL");
  assertInvalid("CCCC");
  assertInvalid("DD");
  assertInvalid("MMMM");
  assertInvalid("XIIII");
  assertInvalid("LXXXX");
  assertInvalid("DCCCC");
  assertInvalid("VIIII");
  assertInvalid("MCCCC");
  assertInvalid("VX");
  assertInvalid("IIV");
  assertInvalid("IVI");
  assertInvalid("IXI");
  assertInvalid("IXV");
  assertInvalid("VIV");
  assertInvalid("XVX");
  assertInvalid("XVV");
  assertInvalid("XIVI");
  assertInvalid("XIXI");
  assertInvalid("XVIV");
  assertInvalid("LXL");
  assertInvalid("XLX");
  assertInvalid("XCX");
  assertInvalid("XCL");
  assertInvalid("CDC");
  assertInvalid("DCD");
  assertInvalid("CMC");
  assertInvalid("CMD");
  assertInvalid("MCMC");
  assertInvalid("MCDM");
}

private void assertInvalid(String r) {
  assertThrows(InvalidRomanNumeralException.class,
               () -> convert(r));
```

```
  }
}
```

The sheer number of tests is a bit daunting. There is probably some redundancy in the `valid` tests. I'm reasonably sure that there is no redundancy in the `invalid` tests.

Future Bob:

I am slightly annoyed that I did not find a more abstract method of determining and testing invalid conditions. I'm sure one exists.

The Cleaning Process

The process I used to first write and then clean that code was relatively simple. I'm going to walk you through each "major" decision; but overall it was just a matter of making smallish improvements while keeping all the tests passing.

The ugly code in Listing 2-1 was not my first attempt. Instead, I fumbled around with a few bad ideas.[6] One was based on a state machine strategy that I cooked up in the shower. I actually wrote the following code and then threw it away before it got too far.

---

6. Linus Pauling was once asked how he had so many good ideas. His reply was something to the effect of he had lots of ideas and threw away the bad ones.

```
package fromRoman;

public class FromRoman {
  enum state {
    I, V, X, L, C, D, M
  }

  public static int convert(String roman) {
    int currentSum = 0;
    for (char c : roman.toCharArray()) {
```

```
        switch (c) {
          case 'I' -> {
            if (currentSum == 3) {
              throw new InvalidRomanNumeralException(rc
            }
            currentSum += 1;
          }
          case 'V' -> currentSum += 5;
          case 'X' -> currentSum += 10;
          case 'L' -> currentSum += 50;
          case 'C' -> currentSum += 100;
          case 'D' -> currentSum += 500;
          case 'M' -> currentSum += 1000;
        }
      }
      return currentSum;
    }

    public static
    class InvalidRomanNumeralException extends RuntimeE
      public InvalidRomanNumeralException(String roman)
      }
    }
  }
```

I finally settled on the strategy in Listing 2-1, and then I massaged it into working, with all the tests passing.

To clean this I looked for some low-hanging fruit. The first I found was the ugly code that checked for illegal prefix combinations. So I extracted that code out into the `checkForIllegalPrefixCombinations` function. Then, I noticed that there was a similar stretch of code, further down, that checked for improper repetitions of letters. So I extracted that code out as the `checkForImproperRepetitions` functions.

In a similar fashion, I extracted out the replacement of the digraphs, the conversions of the letters to numbers, and the check for the numbers being in descending order.

This left me with the following code.

```
package fromRoman;

import java.util.Arrays;

public class FromRoman {
  public static int convert(String roman) {
    checkForIllegalPrefixCombinations(roman);
    checkForImproperRepetitions(roman);
    roman = replaceDigraphs(roman);
    int[] numbers = convertLettersToNumbers(roman);
    checkNumbersAreInOrder(numbers, roman);
    return Arrays.stream(numbers).sum();
  }

  private static
  void checkNumbersAreInOrder(int[] numbers, String r
    int lastDigit = 1000;
    for (int number : numbers) {
      if (number > lastDigit) {
        throw new InvalidRomanNumeralException(roman)
      }
      lastDigit = number;
    }
  }

  private static int[] convertLettersToNumbers(String
    int[] numbers = new int[roman.length()];
    int i = 0;
    for (char digit : roman.toCharArray()) {
      switch (digit) {
        case 'I' -> numbers[i] = 1;
        case 'V' -> numbers[i] = 5;
        case 'X' -> numbers[i] = 10;
        case 'L' -> numbers[i] = 50;
        case 'C' -> numbers[i] = 100;
        case 'D' -> numbers[i] = 500;
        case 'M' -> numbers[i] = 1000;
        case '4' -> numbers[i] = 4;
        case '9' -> numbers[i] = 9;
        case 'F' -> numbers[i] = 40;
        case 'N' -> numbers[i] = 90;
        case 'G' -> numbers[i] = 400;
        case 'O' -> numbers[i] = 900;
        default -> throw new InvalidRomanNumeralExcep
```

```
    }
    i++;
  }
  return numbers;
}

private static String replaceDigraphs(String roman)
  roman = roman.replace("IV", "4");
  roman = roman.replace("IX", "9");
  roman = roman.replace("XL", "F");
  roman = roman.replace("XC", "N");
  roman = roman.replace("CD", "G");
  roman = roman.replace("CM", "O");
  return roman;
}

private static void checkForImproperRepetitions(Str
  if (roman.contains("IIII") ||
    roman.contains("VV") ||
    roman.contains("XXXX") ||
    roman.contains("LL") ||
    roman.contains("CCCC") ||
    roman.contains("DD") ||
    roman.contains("MMMM")) {
    throw new InvalidRomanNumeralException(roman);
  }
}

private static void checkForIllegalPrefixCombinatic
  if (roman.contains("VIV") ||
    roman.contains("IVI") ||
    roman.contains("IXI") ||
    roman.contains("LXL") ||
    roman.contains("XLX") ||
    roman.contains("XCX") ||
    roman.contains("DCD") ||
    roman.contains("CDC") ||
    roman.contains("MCM")) {
    throw new InvalidRomanNumeralException(roman);
  }
}

public static
class InvalidRomanNumeralException extends RuntimeE
  public InvalidRomanNumeralException(String roman)
```

```
    }
  }
}
```

I liked the way this made the `convert` function read. However, I didn't like passing all those arguments around to the helper functions. So I created the internal object with instance variables so that those helper functions could communicate without the overhead of passing arguments.

I really didn't like the ad hoc "49FNGO" character replacement for the digraphs. That was just too clever.[7] So I decided to merge the digraph processing and the conversions of letters to numbers into the same function. I did this one digraph at a time, keeping all the tests passing. This left things as follows.

---

7. "Rule of thumb: if you think something is clever and sophisticated, beware—it is probably self-indulgence" [DOET].

```
package fromRoman;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class FromRoman {
  private String roman;
  List<Integer> numbers = new ArrayList<>();
  Map<Character, Integer> values = Map.of(
    'I', 1,
    'V', 5,
    'X', 10,
    'L', 50,
    'C', 100,
    'D', 500,
    'M', 1000);

  public FromRoman(String roman) {
    this.roman = roman;
  }

  public static int convert(String roman) {
```

```
  return new FromRoman(roman).doConversion();
}

private int doConversion() {
  checkForIllegalPrefixCombinations();
  checkForImproperRepetitions();
  convertToNumbers();
  checkNumbersAreInOrder();
  return numbers.stream().mapToInt(Integer::intValu
}

private void checkForIllegalPrefixCombinations() {
  if (roman.contains("VIV") ||
    roman.contains("IVI") ||
    roman.contains("IXI") ||
    roman.contains("LXL") ||
    roman.contains("XLX") ||
    roman.contains("XCX") ||
    roman.contains("DCD") ||
    roman.contains("CDC") ||
    roman.contains("MCM")) {
    throw new InvalidRomanNumeralException(roman);
  }
}

private void checkForImproperRepetitions() {
  for (String badRep : new String[]
                       {"IIII", "VV", "XXXX", "LL",
                        "CCCC", "DD", "MMMM"}) {
    if (roman.contains(badRep)) {
      throw new InvalidRomanNumeralException(roman)
    }
  }
}

private void convertToNumbers() {
  char[] chars = roman.toCharArray();
  int l = chars.length;
  for (int i = 0; i < l; i++) {
    char nextChar = i + 1 < l ? chars[i + 1] : 0;
    int nextValue = values.get(nextChar);
    switch (chars[i]) {
      case 'I' -> {
        if (nextChar == 'V' || nextChar == 'X') {
          numbers.add(nextValue - 1);
```

```
            i++;
          } else numbers.add(1);
        }
        case 'V' -> numbers.add(5);
        case 'X' -> {
          if (nextChar == 'L' || nextChar == 'C') {
            numbers.add(nextValue - 10);
            i++;
          } else numbers.add(10);
        }
        case 'L' -> numbers.add(50);
        case 'C' -> {
          if (nextChar == 'D' || nextChar == 'M') {
            numbers.add(nextValue - 100);
            i++;
          } else numbers.add(100);
        }
        case 'D' -> numbers.add(500);
        case 'M' -> numbers.add(100);
        default -> throw new InvalidRomanNumeralExcep
      }
    }
  }

  private void checkNumbersAreInOrder() {
    int lastDigit = 1000;
    for (int number : numbers) {
      if (number > lastDigit) {
        throw new InvalidRomanNumeralException(roman)
      }
      lastDigit = number;
    }
  }

  public static
  class InvalidRomanNumeralException extends RuntimeE
    public InvalidRomanNumeralException(String roman)
      super("Invalid Roman numeral: " + roman);
    }
  }
}
```

After that, it was just a matter of reducing all the duplication in the cases and removing as many of the magic numbers as possible.

Then, I turned to the tests. They were pretty ad hoc. So I looked for patterns in the test data and reorganized them. As part of this process, I found some missing invalid combinations and added them.

That's pretty much the whole story. I just made one simple change after the next, while keeping all the tests passing. This is pretty much the procedure that Martin Fowler recommends in his wonderful book, *Refactoring*.[8]

---

8. [Refactoring].

Conclusion

First, I made it work. Then, I made it right—or at least better.

The following chapters in this part of the book are going to expand upon the iterative and incremental techniques I just described. We'll see many other examples, in several other languages. We'll look at the details of low-level coding; details like good naming, comments, and formatting. We'll study higher-level techniques like TDD, Collaborative Programming, and Extract Till You Drop. Finally, we'll look at slightly higher-level concepts such as module structure, objects, data structures, classes, and error handling.

Postscript: Future Bob Playing with Grok3

All the reviews have come back, and I am now preparing the final manuscript for publication. It is February of 2025, and Grok3 just came out. So I fed my "clean" version of the above code into Grok3 and asked it to make improvements. After a bit of thought, it returned the following code, which compiled and which passed all my tests.

```
package fromRoman;

import java.util.Map;

public class FromRoman {
    private String roman;
    private static final String REGEX =
      "^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|
```

```
private static final Map<Character, Integer> VALU
    'I', 1,
    'V', 5,
    'X', 10,
    'L', 50,
    'C', 100,
    'D', 500,
    'M', 1000
);

/** Constructs a FromRoman instance,
    validating the input Roman numeral. */
public FromRoman(String roman) {
    if (!roman.matches(REGEX)) {
        throw new InvalidRomanNumeralException(ro
    }
    this.roman = roman;
}

/** Converts the Roman numeral to an integer. */
public int toInt() {
    int sum = 0;
    for (int i = 0; i < roman.length(); i++) {
        int currentValue = VALUES.get(roman.charA
        if (i < roman.length() - 1) {
            int nextValue = VALUES.get(roman.char
            if (currentValue < nextValue) {
                sum -= currentValue;
            } else {
                sum += currentValue;
            }
        } else {
            sum += currentValue;
        }
    }
    return sum;
}

/** Static method to convert a Roman numeral stri
public static int convert(String roman) {
    return new FromRoman(roman).toInt();
}

public static
class InvalidRomanNumeralException extends Runtim
```

```
        public InvalidRomanNumeralException(String rc
            super("Invalid Roman numeral: " + roman);
        }
    }
}
```

I'm quite pleased with the improvement to the algorithm. It really reduces things to the bare essence. I'm not a big fan of using regular expressions, but this one is not too hard to understand. I don't find any of the comments helpful, so I'd likely delete them.

Then, I asked it to improve my tests. Here's what it came up with:

```
package fromRoman;

import org.junit.jupiter.api.Test;

import static fromRoman.FromRoman.convert;
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import static org.junit.jupiter.api.Assertions.assert

public class FromRomanTest {
    @Test
    public void valid() {
        // Edge case
        assertThat(convert(""), is(0));
        // Single characters
        assertThat(convert("I"), is(1));
        assertThat(convert("V"), is(5));
        assertThat(convert("X"), is(10));
        assertThat(convert("L"), is(50));
        assertThat(convert("C"), is(100));
        assertThat(convert("D"), is(500));
        assertThat(convert("M"), is(1000));
        // Subtractive notation
        assertThat(convert("IV"), is(4));
        assertThat(convert("IX"), is(9));
        assertThat(convert("XL"), is(40));
        assertThat(convert("XC"), is(90));
        assertThat(convert("CD"), is(400));
        assertThat(convert("CM"), is(900));
        // Complex cases
        assertThat(convert("XIV"), is(14));
```

```
        assertThat(convert("CDXLIV"), is(444));
        assertThat(convert("MCMXCIX"), is(1999));
    }

    @Test
    public void invalid() {
        // Too many repeats
        assertInvalid("IIII");
        assertInvalid("VV");
        assertInvalid("MMMM");
        // Invalid subtractive notation
        assertInvalid("VX");  // V can't precede X
        assertInvalid("IIV"); // Incorrect order
        assertInvalid("IXI"); // Invalid sequence
        // Improper structure
        assertInvalid("CMC"); // Multiple subtractive
        assertInvalid("XCL"); // Incorrect order
    }

    private void assertInvalid(String r) {
        assertThrows(FromRoman.InvalidRomanNumeralExc
                     () -> convert(r));
    }
}
```

I like that it organized them so nicely. I remember struggling to find that organization and eventually just deciding to flood the zone. So, kudos to Grok3 for that. On the other hand, the tests are strongly relying on the regular expression and so does, what I consider to be, an inadequate job of testing invalid inputs. For example, it does not check that `"IM"` is invalid. So I'd likely add a few more tests for things like that.

Postscript Conclusion

Grok3 did a nice job. One might therefore question whether my effort of cleaning and testing was worth it. On the other hand, I, and not Grok3, am the final arbiter of the code that gets produced. The buck stops with me. I'm pleased that Grok3 was able to help, and I will be happy of that help when I ask for it. But in order to know that Grok3, or any LLM/AI, is giving me good code, it remains incumbent upon me to do the work to ensure that I can evaluate what it produces. Doing that work will usually involve writing

and cleaning the code that I ask it to improve so that I can properly judge the outcome.

The trade-off is very similar to using an autopilot in an airplane. The autopilot can vastly reduce the workload. But under no circumstances should it be implicitly trusted. Rather, the pilot must continuously oversee its operation. By the same token, no pilot should ever fly without possessing the necessary flying skills that they continuously maintain and refine.

So it is with us. We must stay in practice, and keep our skills high, because we must be able to competently oversee the automation that reduces our workload.