CHAPTER 1

Turning Problems into Code

If you're new to programming, you may wonder how experienced developers can look at a problem and turn it into runnable code. It turns out that writing the actual code is only a small part of the process. You have to break down the problem before you can solve it. If you've ever watched an experienced programmer, it may look like they just cracked open their code editor and banged out a solution. But over the years, they've broken down hundreds, if not thousands, of problems, and they can see patterns. If you're just starting out, you might not know how to do that. So in this chapter we'll look at one way to break down problems and turn them into code. And you can use this approach to conquer the problems in the rest of this book.

Understanding the Problem

One of the best ways to figure out what you have to do is to write it down. If I told you that I wanted a tip calculator application, would that be enough information for you to just go and build one? Probably not. You'd probably have to ask me a few questions. This is often called gathering requirements, but I like to think of it as figuring out what features the program should have.

Think of a few questions you could ask me that would let you get a clearer picture of what I want. What do you need to know to build this application?

Got some questions? Great. Here are some you might ask:

- What formula do you want to use? Can you explain how the tip should be calculated?
- What's the tip percentage? Is it 15% or should the user be able to modify it?
- What should the program display on the screen when it starts?
- What should the program display for its output? Do you want to see the tip and the total or just the total?

Once you have the answers to your questions, try writing out a problem statement that explains exactly what you're building. Here's the problem statement for the program we're going to build:

> Create a simple tip calculator. The program should prompt for a bill amount and a tip rate. The program must compute the tip and then display both the tip and the total amount of the bill.

Example output:

```
What is the bill? $200
What is the tip percentage? 15
The tip is $30.00
The total is $230.00
```

Joe asks:

What do I do with complex programs?

Break down the large program into smaller features that are easier to manage. If you do that, you'll have a better chance of success because each feature can be fleshed out. And most complex applications out there are composed of many smaller programs working together. That's how command-line tools in Linux work; one program's output can be another program's input.

If you're ready to open your text editor and hammer out the code, you're jumping way ahead of yourself. You see, if you don't take the time to carefully design the program, you might end up with something that works but isn't good quality. And unfortunately, it's very easy for something like that to get out into the wild. For example, you hammer out

your program without testing, planning, or documenting it, and your boss sees it, thinks it's done, and tells you to release it. Now you have untested, unplanned code in production, and you'll probably be asked to make changes to it later. Code that's poorly designed is very hard to maintain or extend. So let's take this tip calculator example and go through a simple process that will help you understand what you're supposed to build.

Discovering Inputs, Processes, and Outputs

Every program has inputs, processes, and outputs, whether it's a simple program like this one or a complex application like Facebook. In fact, large applications are simply a bunch of smaller programs that communicate. The output of one program becomes the input of another.

You can ensure that both small and large programs work well if you take the time to clearly state what these inputs, processes, and outputs are. An easy way to do that, if you have a clear problem statement, is to look at the nouns and verbs in that statement. The nouns end up becoming your inputs and outputs, and the verbs will be your processes. Look at the problem statement for our tip calculator:

> Create a simple tip calculator. The program should prompt for a bill amount and a tip rate. The program must compute the tip and then display both the tip and the total amount of the bill.

First, look for the nouns. Circle them if you like, or just make a list. Here's my list:

- bill amount
- tip rate
- tip
- total amount

Now, what about the verbs?

- prompt
- compute
- display

So we know we have to prompt for inputs, do some calculations, and display some outputs. By looking at the nouns and verbs, we can get an idea of what we're being asked to do.

Of course, the problem statement won't always be clear. For example, the problem statement says we need to calculate the tip, but it then says we need to display the tip and the total. It's implied that we'll need to also add the tip to the original bill amount to get that output. And that's one of the challenges of building software. It isn't spelled out to you 100% of the time. But as you gain more experience, you'll be able to fill in the gaps and read between the lines.

So with a little bit of sleuthing, we determine that our inputs, processes, and outputs for this program look like this:

- Inputs: bill amount, tip rate
- Processes: calculate the tip
- Outputs: tip amount, total amount

Are we ready to start producing some code? Not just yet.

Driving Design with Tests

One of the best ways to design and develop software is to think about the result you want to get right from the start. Many professional software developers do this using a formal process called *test-driven development*, or *TDD*. In TDD, you write bits of code that test the outputs of your program or the outputs of the individual programs that make up a larger program. This process of testing as you go guides you toward good design and helps you think about the issues your program might have.

TDD does require some knowledge about the language you're using and a little more experience than the beginning developer has out of the gate.

However, the essence of TDD is to think about what the expected result of the program is ahead of time and then work toward getting there. And if you do that before you write code, it'll make you think beyond what the initial requirements say. So if you're not quite comfortable doing formal TDD, you can still get many of the benefits by creating

simple test plans. A test plan lists the program's inputs and its expected result.

Here's what a test plan looks like:

```
Inputs:
Expected result:
Actual result:
```

You list the program inputs and then write out what the program's output should be. And then you run your program and compare the expected result with the actual result your program gives out.

Let's put this into practice by thinking about our tip calculator. How will we know what the program's output should be? How will we know if we calculate it correctly?

Well, let's *define* how we want things to work by using some test plans. We'll do a very simple test plan first.

```
Inputs:
  bill amount: 10
  tip rate: 15
Expected result:
  Tip: $1.50
  Total: $11.50
```

That test plan tells us a couple things. First, it tells us that we'll take in two inputs: a bill amount of `10` and a tip rate of `15`. So we'll need to handle converting the tip rate from a whole number to a decimal when we do the math. It also tells us we'll print out the tip and total formatted as currency. So we know that we'd better do some conversions in our program.

Now, one test isn't enough. What if we used `11.25` as an input? Using a test plan, what should the output be? Try it out. Fill in the following plan:

```
Input:
  bill amount: 11.25
  tip rate: 15
Expected result:
  Tip: ???
  Total: ???
```

I assume you just went and used a calculator to figure out the tip. If you ran the calculation, your calculator probably said the tip should be `1.6875`.

But is that realistic? Probably not. We would probably round up to the nearest cent. So our test plan would look like this:

```
Input:
  bill amount: 11.25
  tip rate: 15
Expected result:
  Tip: $1.69
  Total: $12.94
```

We just used a test to design the functionality of our program; we determined that our program will need to round up the answer.

When you're going through the exercises in this book, take the time to develop at least four test plans for every program, and try to think of as many scenarios as you can for how people might break the program. And as you get into the more complicated problems, you may need a lot more test plans.

If you're an experienced software developer who wants to get started with TDD, you should use the exercises in this book to get acquainted with the libraries and tools your favorite language has to offer. You can find a list of testing frameworks for many programming languages at Wikipedia.[1] You can read Kent Beck's *Test-Driven Development: By Example* to gain more insight into how to design code with tests, or you can investigate any number of more language-specific resources on TDD.

So now that we have a clearer picture of the features the program will have, we can start putting together the *algorithm* for the program.

Writing the Algorithm in Pseudocode

An algorithm is a step-by-step set of operations that need to be performed. If you take an algorithm and write code to

1. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

perform those operations, you end up with a computer program.

If you're new to programming and not entirely comfortable with a programming language's syntax yet, you should consider writing out the algorithm using *pseudocode*, an English-like syntax that lets you think about the logic without having to worry about paper. Pseudocode isn't just for beginners; experienced programmers will occasionally write some pseudocode on a whiteboard when working with teammates to solve problems, or even by themselves.

There's no "right way" to write pseudocode, although there are some widely used terms. You might use Initialize to state that you're setting an initial value, Prompt to say that you're prompting for input, and Display to indicate what you're displaying on the screen.

Here's how our tip calculator might look in pseudocode:

```
TipCalculator
  Initialize billAmount to 0
  Initialize tip to 0
  Initialize tipRate to 0
  Initialize total to 0

  Prompt for billAmount with "What is the bill amount?"
  Prompt for tipRate with "What is the tip rate?"

  convert billAmount to a number
  convert tipRate to a number

  tip = billAmount * (tipRate / 100)
  round tip up to nearest cent
  total = billAmount + tip

  Display "Tip: $" + tip
  Display "Total: $" + total
End
```

That's a rough stab at how our program's algorithm will look. We'll have to set up some variables, make some decisions based on the input, do some conversions, and put some output on the screen. I recommend including details like variable names and text you'll display on the screen in pseudocode, because it helps you think more clearly about the end result of the program.

Is this the best way we could write the program? Probably not. But that's not the point. By writing pseudocode, we've created something we can show to another developer to get feedback, and it didn't take long to throw it together.

Best of all, we can use this as a blueprint to code this up in any programming language. Notice that our pseudocode makes no assumptions about the language we might end up using, but it does guide us as to what the variable names will be and what the output to the end user will look like.

Once you write your initial version of the program and get it working, you can start tweaking your code to improve it. For example, you may split the program into functions, or you may do the numerical conversions inline instead of as separate steps. Just think of pseudocode as a planning tool.

Writing the Code

Now it's your turn. Using what you've learned, can you write the code for this program? Give it a try. Just keep these constraints in mind as you do so:

Constraints

- Enter the tip as a percentage. For example, a 15% tip would be entered as `15`, not `0.15`. Your program should handle the division.
- Round fractions of a cent up to the next cent.

If you can't figure out how to enforce these constraints, write the program without them and come back to it later. The point of these exercises is to practice and improve.

And if this program is too challenging for you right now, jump ahead and do some of the easier programs in this book first, and then come back to this one.

Challenges

When you've finished writing the basic version of the program, try tackling some additional challenges:

- Ensure that the user can enter only numbers for the bill amount and the tip rate. If the user enters non-numeric

values, display an appropriate message and exit the program. Here's a test plan as an example:

```
Input:
  bill amount: abcd
  tip rate: 15
Expected result: Please enter a valid number for
                 the bill amount.
```

- Instead of displaying an error message and exiting the program, keep asking the user for correct input until it is provided.
- Don't allow the user to enter a negative number.
- Break the program into functions that do the computations.
- Implement this program as a GUI program that automatically updates the values when any value changes.
- Instead of the user entering the value of the tip as a percentage, have the user drag a slider that rates satisfaction with the server, using a range between 5% and 20%.

Onward!

Try to tackle each problem in the book using this strategy to get the most out of the experience. Discover your inputs, processes, and outputs. Develop some test plans, come up with some pseudocode, and write the program. Then accept the various challenges after each program. Or go in your own direction. Or write the program in as many languages as you can.

But most of all, have fun and enjoy learning.