Ryan Michael Kay Follow
Dec 29, 2018 · 8 min read · Listen

# Programming Fundamentals Part 2: The Problem Domain (How To Design A Program/Application)

*This article series is based on rough drafts of what I intend to eventually turn into a series of lectures and courseware for my brogrammers and siscripters out there. Feedback is welcome, and if it proves useful, I would be happy to list you as a contributor.*

2020 UPDATE: I have put together a course for Java which introduces the concepts I have described in these articles, but in greater depth and clarity. If you like my writing, I think you will love my video lectures:

**Working Class Java: A Beginner's Guide To OOP & Software Architecture** Udemy Link | Skillshare Link w/ Free Trial

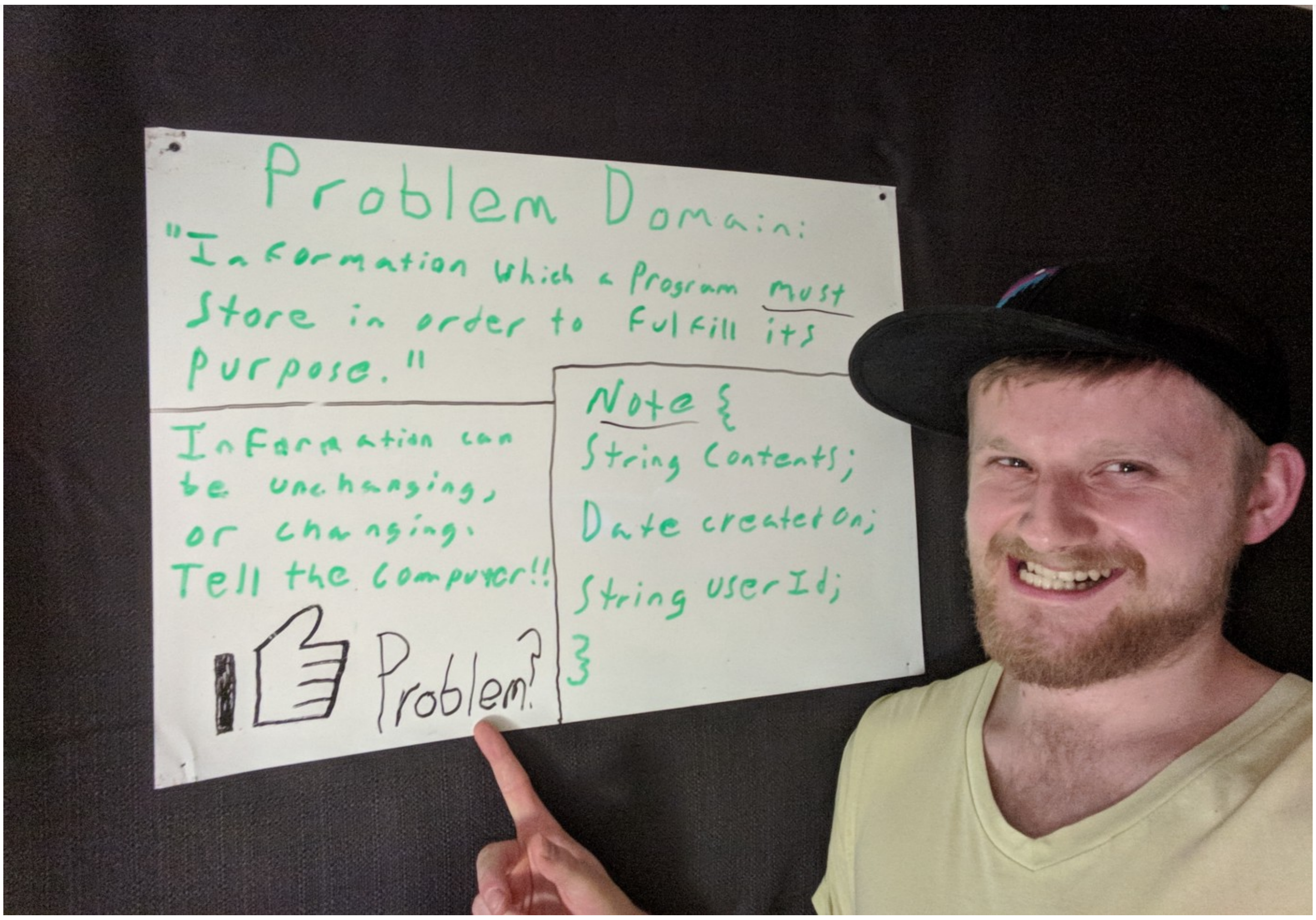
**Contents**

In the previous article, my goal was to give you some way to think about what a program is, without needing to know much about actually writing them. This is no easy task, as I still distinctly recall from my own experience as a beginner, that programming felt very unlike anything I had ever learned about or understood in the beginning.

Whether or not much of that article made sense is not urgently important, as these were points I did not understand (at least in a relatively clear way), until several years into my studies. In any case, let us review my working definition of a program:

*"A program is in essence, a set of instructions to be given to an Information Processing System"*

In this article, we are going to focus on the subject of figuring out what a program needs in order to solve the problem which it is designed to solve. In other words, we will examine the idea and process behind analyzing the "Problem Domain" of a program. I know that may sound vague and esoteric, but this concept is not complicated (at least the way I explain it), and it is actually the process one uses to begin the initial process of building a new program.


Problem Domain:
"Information which a Program must Store in order to Fulfill its purpose."
Information can be unchanging, or changing. Tell the Computer!!
Problem?
Note {
String Contents;
Date createdOn;
String userId;
}

Problem?

**What Does The Program Do?**

The first step in determining the Problem Domain of a program is to ask yourself what your application is supposed to do. Since the goal of this series is to show you the process of thinking about, designing, and building programs, we will choose a very simple problem for our hypothetical program to solve. For this series, we will build a simple **binomial expression calculator**.

The first step in designing any program is to start by explaining what it does in simple, written language. The technical term for this written explanation is a "**Problem Statement**." Now, as a complete beginner, I recall being worried about whether or not my problem statements were perfect or not, and the truth is that trying to determine the perfect problem statement (and by extension, problem domain) on your first try, is a fool's errand.

Instead, use this process as a starting point which can be refined as new ideas (or roadblocks) pop up during the design and build process of your program.

**Example 1: Rough Problem Statement for a Calculator Program**

*"My program will solve basic mathematical expressions."*

As you practice this process and build many different programs, you will have a much easier time figuring out good quality problem statements. The above example is rather like what I would have made as a beginner, and we will see shortly that some improvement is required.

In any case, let us discuss what we are trying to do by writing problem statements. Once you have written down your problem statement, pay very close attention to the verbs and nouns, as these two things will tell you the information, and the functions (what we do with that information), which your program must have in order to work properly (and therefore solve a problem). In this case, we can start with:

As mentioned before, there is nothing remotely complicated about what we just did; we made a simple description of a program and looked at what things (nouns) and functions (verbs) it will probably require. However, my experience tells me that we could benefit greatly by being more specific to our problem statement. It is okay to start vague (sometimes you will need to spend a long time researching and refining your understanding of a problem domain), but more questions need to be answered before we are ready to proceed to code:

- What kinds of mathematical expressions do we want our program to be capable of evaluating? Our job is simple if we just want addition and subtraction, but what about logarithms or cube roots?
- How many operands (terms) should we allow the user to work with?
- What about decimals or negative integers?
- What about invalid expressions? If we try to calculate "1 ÷ 0", won't the universe collapse upon itself? How about if the user types 0.0.0.0 + 1?

The point I am trying to make here is that our problem statement, upon closer analysis, really is quite vague. Now, I must mention that our goal at this stage is not to figure out every single detail and include it in the problem domain before we write any code (**paralysis by analysis**), but we can save ourselves a great deal of a headache just by spending a few minutes asking important questions like the ones above.

**Example 2: Refined Problem Statement for a Calculator Program**

*"My program will display the result of solving valid binomial expressions limited to addition, subtraction, multiplication, and division. If an expression is invalid, an error message will be displayed instead of the result of solving the expression."*

That sounds much better to me. As you can see, we have essentially just tried to be more specific about what our program will and will not do. Let us take a second look at what information our program will need to hold as values (not changeable) and variables (changeable)…:

- Two operands (terms) which will presumably be entered by the user
- An operator to specify one of addition, subtraction, multiplication, or division
- An error message to tell the user if the expression turns out to be invalid
- The result of solving the expression, assuming it is valid

…And what our program will need to do with that information:

- Ensure that the expression is valid
- Solve a valid expression
- Display the result of solving a valid expression
- Display an error message if an expression turns out to be invalid

As we will see in the next article, the above information (which is the result of analyzing our program's problem domain), will give us what we need to start building our program with a solid foundation.

#### Finding The Middle Way Between Pragmatism And Perfectionism

I feel this is a good moment to introduce what I like to call the spectrum of pragmatism versus perfectionism. Let us pretend that we are starting again from step 0 of the process explained above, but our goal is to build the ultimate calculator program in order to secure our position as the market leader in our niche of the software sector.

We will not go through the whole process again, but if my primary goal was to build the ultimate calculator program, I would probably want it to solve any valid expression, of any number of terms, with any kind of operator, in any kind of base counting system (why not binary and hexadecimal too!), with graphing capabilities, and the list goes on. That would be a pretty damn powerful features!). I am not saying it would be impossible, but writing this level of the program by yourself would take several months, if not years to complete. There are ways to save time (by using other people's pre-written helper programs, called libraries), but you will likely have to put up some cash to use the good quality ones.

Let us start again from step 0, except this time we will pretend that we have a time limit of 10 minutes to write our calculator program, after which our code will be evaluated by a technical interviewer of a local startup. In this case, our primary concern would be figuring out the simplest possible problem statement for our calculator would be priority number one!

As you can see, one of the most important things you need to pay attention to when beginning to design a new program, is the balance between making the most fully featured program you can make, versus being realistic about resources, time constraints, and your skill as a developer. The good news is that an easy way to balance these concerns (not to mention justify subscription based payments from your users), is to start with basic features, and gradually release your program with new features as you build them.

In fact, this is exactly how I suggest you build programs. Even if you have a complex yet specific Problem Domain established before you have written any code, I strongly advise you to start with only the most fundamentally important features for your program, and be prepared to change things as you build them out!

Mark my words, be prepared to see your problem statements/domains change over time in ways you could not possibly have imagined, for reasons ranging from technical restrictions (of devices, languages, operating systems, libraries), to making the person who signs your paycheque happy even if their ideas suck. Learn to be **agile**.

The second bit of good news is that in part 5, I will show you how to build programs that are easy to change and improve over time. Do not listen to people who say that software architecture does not matter; that statement is as silly as claiming that building architecture does not matter.

#### Summary

The first step in writing a program is to determine what kind of information our program will need to represent digitally (such as mathematical operands and operators), and to know what functions we will need to apply to that data (such as addition, or printing an error message to the screen). I refer to this collectively as the problem domain, and a great way to figure out what it looks like is to start with a problem statement!

Now, once you begin to write more complicated applications that solve more than one problem, you will want to look into things like **Use Cases** and **User Stories**, which allow you to be more practical and systematic in the way you design complex applications. In any case, it always comes back to problem domain analysis of one form or another, and problem statements are a great start.

### Support

Follow the wiseAss Community:
https://www.instagram.com/wiseassbrand/
https://www.facebook.com/wiseassblog/
https://twitter.com/wiseass301
http://wiseassblog.com/
https://www.linkedin.com/in/ryan-kay-808388114

Consider donating if you learned something:
https://www.paypal.me/ryanmkay