Program Design in a Nutshell:

The **design recipe for data definitions** guides the students to ask the following questions:
- Can you represent the information by a primitive data type?
- Are there several related pieces of information that describe one item? If yes, design a composite data type (struct, class).
- Does the composite data type contain another complex piece of data? Define that data type separately and refer to it. (A Book data item contains an Author data item.)
- Are there several variants of the information that are represented differently, but are related (e.g. a circle, a rectangle, a triangle --- all are shapes)? If yes, design a union type. (In Java, define a common interface.)
- Repeat these steps. This may lead to self-reference, mutual reference, and eventually to a complex collection of classes and interfaces.
- Make examples of data for every data type you design.

The **design recipe for functions/methods**:
- Write down in English the purpose statement for the function/method, describing what data it will consume, and what values it will produce. Add a contract that specifies the data types for all inputs and the output.
- Make examples of the use of the function/method with the expected outcomes.
- Make an inventory of all data, data parts, and functions/methods available to solve the problem.
- Now design the body of the function/method. If the problem is too complex, use a wish list for tasks to be deferred to helper functions.
- Run tests that evaluate your examples. Add more tests if needed.

If every function produces a new value, the result, then the entire design process is very straightforward:
- Tests are simple, as they only verify that the result matches the expected value.
- Function composition comes naturally --- the result of any function application can be used in further computations.
- The order of computation does not affect the result. (However, a function or a data item must be defined before it can be used.)

To test method that change state (have side effects) you have to do the following:
- **Setup**: Initialize the data needed to invoke the method (and to verify the results)
- **Invoke the method to be tested**.
- **Test** the expected results, and the expected changes (effects).
- **Tear-down**: Reset the data that has been used to their original values (it the data will be used again in other tests).

The **design recipe for abstractions** helps us eliminate code repetition and produce a more general solution:
- Mark all places where the similar code segments differ.
- Replace them with parameters and rewrite the solution using them as arguments.
- Rewrite the original solutions to your problems by invoking the generalized solution with the appropriate arguments.
- Make sure that the tests for the original solution still pass.