A top-down program design process

1. Clearly state the problem that you are trying to solve. Programs are usually written to fill some perceived need, but that need may not be articulated clearly by the person requesting the program. For example, a user may ask for a program to solve a system of simultaneous linear equations. This request is not clear enough to allow a programmer to design a program to meet the need; he or she must first know much more about the problem to be solved. Is the system of equations to be solved real or complex? What is the maximum number of equations and unknowns that the program must handle? Are there any symmetries in the equations that might be exploited to make the task easier? The program designer will have to talk with the user requesting the program, and the two of them will have to come up with a clear statement of exactly what they are trying to accomplish. A clear statement of the problem will prevent misunderstandings, and it will also help the program designer to properly organize his or her thoughts. In the example we were just describing, a proper statement of the problem might have been:
Design and write a program to solve a system of simultaneous linear equations having real coefficients and with up to 20 equations and 20 unknowns.

2. Define the inputs required by the program and the outputs to be produced by the program. The inputs to the program and the outputs produced by the program must be specified so that the new program will properly fit into the overall processing scheme. In the previous example, the coefficients of the equations to be solved are probably in some preexisting order, and our new program needs to be able to read them in that order. Similarly, it needs to produce the answers required by the programs that may follow it in the overall processing scheme, and it must write out those answers in the format needed by the programs following it.

3. Decompose the program into classes and their associated methods. Define one or more classes, and determine how the classes interact with each other and with the outside world. Define a separate method to implement each interaction.

4. Design the algorithm that you intend to implement for each method. An algorithm is a step-by-step procedure for finding the solution to a problem. It is at this stage in the process that top-down design techniques come into play. The designer looks for logical divisions within a method, and divides the method up into subtasks along those lines. This process is called decomposition. If the subtasks are themselves large, the designer can break them up into even smaller subsubtasks. This process continues until the problem has been divided into many small pieces, each of which does a simple, clearly understandable job. (As we shall see in later chapters, these separate pieces can become separate methods.)

After the problem has been decomposed into small pieces, each piece is further refined through a process called stepwise refinement. In stepwise refinement, a designer starts with a general description of what the piece of code should do and then defines the functions of the piece in greater and greater detail until they are specific enough to be turned into Java statements. Stepwise refinement is usually done with pseudocode, which will be described in the next section.

It is often helpful to solve a simple example of the problem by hand during the algorithm development process. If the designer understands the steps that he or she went through in solving the problem by hand, then he or she will be better able to apply decomposition and stepwise refinement to the problem.

5. Turn the algorithm into Java statements. If the decomposition and refinement process was carried out properly, this step will be very simple. All the programmer will have to do is to replace pseudocode with the corresponding Java statements on a one-for-one basis.

6. Test the resulting Java program. This step is the real killer. The components of the program must first be tested individually, if possible, and then the program as a whole must be tested. When testing a program, we must verify that it works correctly for all legal input data sets. It is very common for a program to be written, tested with some standard data set, and released for use, only to find that it produces the wrong answers (or crashes) with a different input data set. If the algorithm implemented in a program includes different branches, we must test all of the possible branches to confirm that the program operates correctly under every possible circumstance.
Large programs typically go through a series of tests before they are released for general use (see Figure 3.2). The first stage of testing is sometimes called unit testing. During unit testing, the individual components of the program are tested separately to confirm that they work correctly. After the unit testing is completed, the program goes through a series of builds during which the individual components are combined to produce the final program. The first build of the program typically includes only a few of the components. It is used to check the interactions among those components and the functions performed by their associated methods. In successive builds, more and more components are added, until the entire program is complete. Testing is performed on each build, and any errors (bugs) that are detected are corrected before moving on to the next build.

Testing continues even after the program is completed. The first complete version of the program is usually called the alpha release. It is used by the programmers and others very close to them in as many different ways as possible, and the bugs discovered during the testing are corrected. When the most serious bugs have been removed from the program, a new version, called the beta release, is prepared. The beta release is normally given to friendly outside users who have a need for the program in their normal day-to-day jobs. These users put the program through its paces under many different conditions and with many different input data sets, and they report any

**Figure 3.2.** A typical testing process for a large program

![Diagram showing the testing process](image-url)
bugs that they find to the programmers. When those bugs have been corrected, the program is ready to be released for general use.

Because the programs in this book are fairly small, we will not go through the sort of extensive testing. However, we will follow the basic principles in testing all of our programs.

The program design process may be summarized as follows:

1. Clearly state the problem that you are trying to solve.
2. Define the inputs required by the program and the outputs to be produced by the program.
3. Decompose the program into classes and their associated methods.
4. Design the algorithm that you intend to implement for each method.
5. Turn the algorithm(s) into Java statements.
6. Test the Java program.