A notation for specifying interactive systems

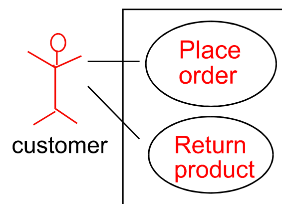
For software to generate payroll checks and bank statements, it may be sufficient to specify the input, the output, and the relationship or transformation desired between input and output.

For example, to specify precisely a program that computes square roots, the specification should state simply that the program should prompt for input of a non-negative integer or real number, and should display the square root of the number. The *design* of the program, of course, would have to state *how* a square root is computed.

For *interactive* software, in contrast, specification must be more detailed. In general, such software provides *services* to *users*. So there is an assumption that the user requires a certain series of interactive steps to obtain the result desired.

In the 1990s, with the widespread use of the Internet and graphical user interfaces, a standard notation for specification of interactive systems was developed and became known as the Unified Modeling Language (UML). The problem descriptions include *services, assumptions,* and *risks,* as well as descriptions of typical interaction between the user and the environment.

UML specification includes the notion of *use cases*, which are such typical interactions, in which the user of the system is called an *actor*. UML supports *use-case diagrams*, in which the actor is given a name and depicted as a stick figure, and each use case is named within an oval (Figure below).



The diagram indicates that a customer may wish to use the system to place an order or to return a product. Either use case will require several steps, such as logging in and choosing whether to place an order, cancel an order, or return a product.

Tracing loops

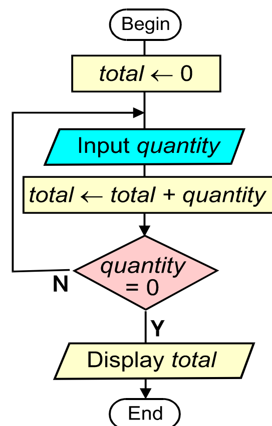
The design of algorithms is something between a technical skill and an art. If you can imagine how to accomplish a task, then it may be easy to write down the steps precisely as pseudocode, as a flowchart or as a program in a programming language.

The most tricky part of algorithm design is the design of loops. A loop must execute precisely the correct number of times. For that reason, the exit condition and the steps that change the values tested in the exit condition are crucial matters.

Two techniques enable checking the correctness of an algorithm design: *coding and testing* it using a programming language, or *tracing* it with a pencil and paper, either way using sample input data.

Tracing flowcharts is a skill that extends well to programming in a language like Java, because when a program fails to work correctly, it is often impossible to find the error without displaying the values of variables as they change during the execution of the program, particularly during the execution of loops. Finding the steps that are incorrect reduces to finding the place in the program execution where a variable gets the wrong value.

To trace a flowchart or pseudocode expression of an algorithm, we make a table with one column for each variable used in the algorithm, plus one column for the algorithm's output. Then, using sample data, fill entries in each column with



Consider the flowchart above. To trace it, we create a table

quantity	*total*	*output*
	0	
3	3	
2	5	
1	6	
0	6	6