Thinking About Asking: Encouraging a Questioning Approach to Requirements Gathering and Problem Solving

Madalene Spezialetti
Computer Science Department
Trinity College
Hartford, CT USA
madalene.spezialetti@trincoll.edu

Abstract— **When a client, describing a problem that requires a computing solution, mentions their "old" customers, does it refer to the age of the customers or the length of time the customers have been with the business? How big is "big", how much is "too many" and what exactly should happen to the extra banquet guests that fail to fill an entire table? While typical programming assignment descriptions are fully specified, with all ambiguity removed, real-world problems are frequently ill-defined, filled with ambiguity and open-ended. Successfully translating these problems into computing solutions involves both recognizing the issues that need to be addressed and asking questions to resolve the unknowns. Typical classroom examples and assignments purposely lack ambiguity and so do not provide students with the opportunity to develop the questioning skills that will be required in professional life. Further, for some students, asking questions implies not understanding the material, and thus can be perceived as a negative attribute. This paper presents an approach to encouraging an environment of questioning surrounding problem descriptions and potential solutions. The generality of the approach enables it to be incorporated into classes ranging from CS1 to upper-level courses such as Software Engineering and senior capstones.**

Keywords—requirements gathering; CS1; Software Engineering

I. Introduction

Typical classroom examples and assignments are designed to demonstrate language or logic features in a clear, complete manner. Ambiguity is a quality that is sought to be eliminated in lectures, tests, labs and programming assignments. However, real-world problems are full of ambiguity that must be resolved in order to create programs to solve them. Vague or incomplete information may be given, while required information may be lacking [1, 2, 3]. Identifying the ambiguities or missing information in a problem description requires consideration of the context of the problem and the ability to formulate questions that resolve the ambiguities [4,5]. But if asking questions is important to requirements gathering in the real world, then it is important to begin developing questioning skills in tandem with programming skills. This paper presents a new strategy for fostering an environment of questioning which emphasizes the view of program examples as problem scenarios that have a context in which they exist and a client who is the source of information and also the judge of results. This approach encourages students to think about requirements gathering as problem solving that involves two-way communication, critical and common sense analysis of a situation and asking questions to explore beyond the information provided.

II. Evaluating Problems and Programs

There can be a natural inclination among students to focus on getting a program to run error free, which is to say, getting the "right answer". One of the goals of developing a questioning approach is to foster an understanding and appreciation of the connection between "the right answer" and the needs and preferences of the client. As a first step in emphasizing the importance of client input and evaluation, students are given a problem description and a program to run and evaluate in light of that description. Taking on the role of the client provides students with the opportunity to be objective and critical judges of the correctness of the program and the quality of the user experience. By comparing a not-entirely-precise problem description to the program's performance, students can also form an opinion as to whether issues they identify are due to incomplete information in the problem description or misunderstanding or error on the part of the programmer.

Students are given basic questions to consider when analyzing both the problem statement and the solution program, including: *What is the context and what are the issues of the basic problem to be solved? What information needs to be provided? What results are expected? Is there*

ambiguous information in the problem statement? Is any information missing or situation not considered? Does the output fulfill expectations?

Students are then asked to evaluate the scenario and the program's performance in light of these questions. The following example illustrates the process:

Happy Day Daycare frequently takes groups of children to the zoo and wants a program to estimate the costs for such outings. The presence of two adults is required for every three children age 3 and under and one adult is required for every 5 children age 3 and over. Zoo admission is $10.00 per adult and $5.00 per child. Vans for transportation, which can carry up to 10 individuals, are $50.00 each. The output should be in the form of a complete report, which can be passed on to a supervisor for approval.

The problem description intentionally includes a number of points of ambiguity: the chaperon description counts 3 year-olds in both categories, it does not specify what is meant by a "complete report" nor what information is to be input. The associated program was designed to give students the opportunity to determine if its correctness, interface and resolution of ambiguities met their own expectations given the scenario description. The program's performance included input prompts to "Enter the number of children 3 and under" and "Enter the number of children 3 and over" (reflecting the ambiguity in the problem statement), a logic error (failing to include the van rental cost in the overall total) and input prompts for the van rental and admission charges (allowing students to consider if requiring those values to be input every run is an annoyance or a necessity).

After testing and evaluating the program, students are asked to formulate the questions that, as programmers, they would ask the client in order to resolve the ambiguities and clarify input and output expectations. This initial exercise serves to emphasize, early and outside the context of programming logic, the importance of the client's opinion and of communication between the client and programmer in designing a "correct" program that meets the client's expectations. In CS1, it can also be used as the initial exercise to introduce students to running a program in a development environment. Unlike "Hello World", the exercise immediately places students in the critical thinking roles of client and problem-solver.

III. Incorporating Problem Scenarios

To foster an environment of questioning, it is important for instructors to talk about ambiguity and the importance of two-way communication, and to provide students with ambiguous or incomplete scenarios with which to develop their analysis and questioning skills. There are a wide variety of ways to tailor such activities to suit various instruction modes, class levels and time restrictions. For instance, at the CS1 level, one of the examples used to illustrate basic language constructs can include ambiguity in its problem description, such as saying "Let's write an if-statement where admission for a person over 65 is $10 and under 65 is $12". In this way, identifying and resolving ambiguities are treated as integral parts of correctly translating problem descriptions to programming constructs. Programming lab/assignment descriptions can include initial ambiguities that the class is instructed to identify and which can then be discussed and resolved before implementation begins. A lab setting can be used to allow students to discover ambiguities as they design and implement program solutions, asking the instructor for clarification as needed. If a class has engendered an environment of ambiguity awareness and questioning, problem scenarios can be included on exams that require students to provide questions they would ask to resolve missing information or ambiguous statements.

Below, topic areas around which incompleteness, ambiguity and questioning can be explored are discussed, along with sample intentionally ambiguous problem scenarios.

A. Output Expectations

The initial exercise described in Section II above illustrates, particularly to students who do not yet know how to program, the importance of output as a means by which clients may express their expectations of a program and determine their ultimate satisfaction. Thus the question "What visible results should the program produce?" serves as a good starting point for exploring client expectations. The expected output not only defines what problem a program is intended to solve, but also indicates what information or results the client perceives as having lasting relevance. A client's problem description is likely to include information that is not relevant to the solution, but at the same time, information that may seem irrelevant may prove useful or raise important questions. It is the responsibility of the developer to accurately determine the client's output expectations and how those expectations determine the problem to be solved. Scenarios that contain a distinct interconnection between the context in which the program will be used, the expected output and the program logic, highlight how questions about one of these aspects can inform and generate questions about the others. As an example, consider the following:

In the fall, Sunnydale Farms gives hay rides in the evenings. The regular cost is $5.00 per person for the hay ride, and $8.00 per adult and $4.00 per child 12 and under for all-you-can-eat pie service at the end of the ride. The program will be used by Maggie, who answers the phone and provides people with estimates, so just giving the projected cost based on the number of adults and children is all that is needed.

Based on the problem statement, questions like "What kind of pie is served?" or "What time does the hayride start?" are not relevant, but knowing what output is expected and that the program will be used in the context of answering customer calls are relevant. Given that a customer may well ask how the total cost was derived, a pertinent question would be: "Should only the overall cost be displayed or should a cost breakdown

be included?" Determining if a breakdown should be displayed also leads to the question of whether hayrides and pie service can only be purchased together, or if customers can opt to buy only one or the other. Beyond simply determining what to display, the answers to these questions emphasize the impact of clarifying output expectations and considering the context of the program's use on algorithm design.

B. Input: Identification and Validity

Of the many input issues that can arise, two are considered here: the identification of appropriate input and input validity. Identifying input values that are necessary versus optional affects program usability, since relieving the user of the need to repeatedly input relatively stable values is efficient and can reduce input errors. At the same time, building values into code limits flexibility, as changes to those values require changes to the code. Scenarios which highlight this balance demonstrate the practical costs of determining what, if any, optional values to input. As an example, reconsider the Sunnydale hayride scenario above. Requiring the user to input the prices for the hayride and the adult and child pie service allows the program to accommodate changes to those costs without changing the code. The tradeoffs are possible input errors, longer response times when answering customer price requests and potential irritation on the part of the person using the program. The scenario provides the opportunity to develop questions such as "How often do the prices change?" and, since there is actually a known user, a relevant question may well be "How would Maggie feel about entering that information each program run?"

Input validity is frequently addressed in the context of accepting or rejecting input based on whether or not it meets specific criteria. For example, if a user is shown a menu of 4 choices and asked to enter the number of the choice, it is clear that only the numbers 1 through 4 should be accepted. Such scenarios are quite straightforward to develop. However, it is also important to provide scenarios that allow input validity to be explored in the context of the correctness and completeness of a problem description in order to highlight the impact of resolving those issues on algorithm design. As an example, consider the following scenario:

The Old Time Games Company wants the old time game Even and Odd, which is played between two people, translated into a game where the user plays against the computer. In Even and Odd, one player is designated "even" and the other "odd". The players hold their hands behind their backs and on the count of 3, both show their hands with some number of fingers displayed. The numbers are added together. If the sum is even, the player designated "even" wins and if the sum is odd, the player designated "odd" wins.

With regard to input, the problem exhibits two levels of ambiguity. The first is the language ambiguity "*The players hold their hands behind their backs"* leading to the question, "Does each player use one hand (meaning the most a player can show is 5) or two hands (allowing each player to show up to 10)?" A second ambiguity is related to the game itself: "Can the number of fingers displayed be zero?" If so, it leads to the algorithmic question "Is a sum of zero considered even or odd…or neither?" Scenarios such as this one demonstrate the importance to algorithm design of resolving omissions or ambiguities related to input validity in problem statements.

C. Limits and Boundaries

An important aspect of understanding a problem involves identifying the presence of limits and boundaries. Such limits may be fixed and simply factors in calculations, for example, "vans can hold 10 people", or they may be used to delineate the end of repetition, such as "the game is over when the first player reaches a score of 21". When analyzing a problem, it is important that known limits are clear, as are the actions to be taken when the limits are reached. However subtler limits may exist or existing limits may be omitted entirely from a problem statement. Scenarios in which limits (may) logically exist but are not directly mentioned, allow students to practice inferring, through the overall context of the problem itself, the possible existence of those limits and formulating questions to identify them.

Consider the following scenario that includes one vaguely stated limit for ending the main action of the program (printing tickets) and also has a second limit that is not directly addressed in the problem statement:

The Make Yourself at Home Movie Theater, a 1-screen theater catering to families, needs a program to print tickets and calculate how much money has been made on a particular showing. The ticket seller will enter the name of the movie and then, as tickets are sold, will enter information for each sale, after which a ticket will be printed. Admission for adults is $5.00, admission for children is $3.00. At the end of the night a report including the number of each kind of ticket sold and the total amount of money in ticket sales is to be printed.

When considering the limits that are part of this problem, an obvious ambiguity is the meaning of "*at the end of the night.*" Should some input value be specified to indicate that the sales are over and that the report should be generated? Having already identified one condition that will end the main logic of selling tickets, it is easy to overlook a hidden limit that is not addressed directly in the problem statement: "What is the number of seats in the theater and what should happen when the number of tickets sold reaches or is about to exceed that limit?" If this situation were not identified and the question asked, the resulting program would allow tickets to be sold with no limit, which is probably not what the client had in mind, even though it was not explicitly mentioned.

Problem scenarios with unmentioned limits highlight how seemingly complete descriptions may include areas that require further investigation. They also reinforce the

relationship between context, boundaries and limits and a program's completeness and correctness.

D. Groupings: Too Few and Too Many

In solving problems, the pattern often arises of forming elements into groups of given sizes: 12 eggs in a dozen, 6 boxes in a carton, 8 people on a team. These scenarios often give rise to situations in which too few elements exist to form even a single group or, after forming as many complete groups as possible, some number of excess elements remain. The ways in which these situations are handled can vary by context and presenting students with such variations allows them to explore how problems that seem logically similar may in reality be quite different. Consider these scenarios:

A department frequently has pizza parties. Experience indicates that each person typically eats two pieces of pizza. The pizza shop only sells 16 cut pizzas. The department admin wants a program that, given the number of people who will attend an event, determines the number of pizzas to order.

A tour company organizes trips to New York City. The company wants a program to determine the number of busses required to transport the people who have signed up for a given trip. Each bus can hold up to 50 passengers.

For both problems, it could be possible that the number of items present does not reach the limit to form even one grouping. Suppose only 3 people sign up for a pizza party or only 10 people make reservations for a tour. What should be done? Each client may have a different response to the question. For the pizza party, the answer may be that the pizza won't go to waste so even if one person is attending, a pizza should be ordered. The tour company owner may say that renting a bus that holds 50 people to take 10 people on a tour is not a financially sound decision, and so such a trip should be cancelled. This response, however, should inspire more questions about both the minimum number of people needed to conduct a tour and how overflow passengers (for example, if 60 people sign up for a tour) should be handled. As these examples illustrate, problems that may be fundamentally the same with regard to their core calculations, may require very different logic due to the context of the problem. Awareness of and questions exploring that context are needed to extend the basic description to one that will result in a complete and correct algorithm.

E. Decoupled Requirements Gathering

To provide students with more complex requirements gathering experiences, scenarios can be developed that are decoupled from specific programming capabilities. Decoupling the problem description from programming constructs provides greater latitude (especially at the CS1 level) for developing broad and general questioning skills. For example, the following scenario challenges students to determine factors a company uses to estimate the cost of digging holes. Challenges lie in identifying all the factors which may need to be considered in calculating hole digging estimates and in developing questions to determine how the information presented in the description can be quantified:

The We Dig Holes Company digs holes. All kinds of holes: big holes, small holes, one hole or lots of holes, quick holes or ones that take longer and in all kinds of weather (although they can dig faster in good weather). Some holes are fun to dig, others are not fun. Their motto is "You need a hole, we'll dig a hole". They would like a program to help them estimate the cost of digging holes, or more accurately, bids for jobs that involve digging holes. The way estimates are now determined is more of an art form…and based on experience.

Here the questions involve not only the resolution of what various terms mean ("big", "small", "fun"), but also how (or if) they can be quantified. For example, before considering how to approximate the cost of digging a big hole, questions such as "How is the size of a hole determined?", "Is it measured or estimated?", "If it is measured, should the actual measurements be input or is indicating "big" or "small" good enough?" and "How big is big?" need to be resolved. Students can also explore such factors as what interfaces (keypad, sliders, menus) are appropriate or favored for input.

IV. Conclusions and Future Work

An environment of questioning can be fostered by providing students with opportunities to critically analyze ambiguous or incomplete problem scenarios in order to develop and demonstrate their requirements gathering and problem analysis skills. These opportunities can take on a variety of forms, including lecture examples, program assignment descriptions that include initial ambiguities that the class explores and resolves before implementation or exam questions that present scenarios for which students indicate the questions that they would ask to resolve any ambiguities. To aid in this process, future work will include developing an online repository of intentionally ambiguous scenarios and exercises designed specifically for CS1 courses, as well as for advanced programming courses such as Software Engineering.

References

[1] E. Douglas, S.Agdas, M. Koro-Ljungberg, C. Lee and D. Therriault, "Ambiguity in Engineering Problem Solving", 2015 Frontiers in Education, 1712-1715.

[2] F.G. Splitt, "The Challenge to Change: On Realizing the New Paradigm for Engineering Education", Journal of Engineering Education, Vol. 92, p.181-187, 2003.

[3] S. Cassel and B.Victor," A Structured Approach to Training Open-Ended Problem Solving", IEEE Frontiers in Education Conference, 2015, pp. 417-420.

[4] F. Kowalski and S. Kowalski, "A Pedagogical Model for Nurturing Curiosity", 4th International Conference on New Perspectives in Science Education, 2015, pp.362-366.

[5] F. Kowalski and S. Kowalski, "Helping Tomorrow's Engineers Ask Productive Questions", 2015 Frontiers in Education, pp. 209-210.