Common sense computing (episode 4): debugging
Beth Simon\textsuperscript{a*}, Dennis Bouvier\textsuperscript{b}, Tzu-Yi Chen\textsuperscript{c}, Gary Lewandowski\textsuperscript{d}, Robert McCartney\textsuperscript{e} and Kate Sanders\textsuperscript{f}

\textsuperscript{a}Department of Computer Science and Engineering, University of California, San Diego, USA; \textsuperscript{b}Department of Computer Science, Southern Illinois University, Edwardsville, USA; \textsuperscript{c}Department of Computer Science, Pomona College, Claremont, USA; \textsuperscript{d}Department of Mathematics and Computer Science, Xavier University, Cincinnati, USA; \textsuperscript{e}Department of Computer Science and Engineering, University of Connecticut, Storrs, USA; \textsuperscript{f}Department of Mathematics and Computer Science, Rhode Island College, Providence, USA

We report on responses to a series of four questions designed to identify pre-existing abilities related to debugging and troubleshooting experiences of novice students before they begin programming instruction. The focus of these questions include general troubleshooting, bug location, exploring unfamiliar environments, and describing students' real life troubleshooting experiences. We find that the common sense experience of students related to debugging is often at odds with the needs for debugging programs. Student troubleshooting experiences recognize real world nondeterminism and rarely require undoing an attempted fix. Based on student answers, we shed light on some specific frustrations of instructors and make a set of recommendations for teaching, including addressing the difference between finding an error and fixing it, modeling the formation of multiple hypotheses, and encouraging students to apply domain knowledge.

\textbf{Keywords:} CS1; beginners; debugging; preconceptions; constructivism; novices

1. Introduction

The constructivist learning model tells us that learning is grounded in and constructed out of prior understandings and beliefs. As a result, pre-existing knowledge and abilities are important for understanding learning. In this work we report on four experiments designed to identify pre-existing student abilities that might support novice program debugging instruction.

There are two key aspects of pre-existing knowledge in constructivist learning theory. First, constructivism recognizes that rather than learning entire theories which then govern how one handles all cases learners have knowledge in pieces that reflects the incremental building of understanding. These pieces of knowledge, which may include pre-existing knowledge (i.e. before formal study), are used long after the general theory is understood, even by experts (Smith, diSessa, & Roschelle, 1993). As a result, by understanding pre-existing debugging knowledge we gain insight into the processes students will use to debug.

\*Corresponding author. Email: bsimon@cs.ucsd.edu

ISSN 0899-3408 print/ISSN 1744-5175 online
© 2008 Taylor & Francis
DOI: 10.1080/08993400802114698
http://www.informaworld.com
as they become more expert. We refer to the pre-existing knowledge related to a particular topic as preconceptions about the topic.

The second constructivist notion involved in pre-existing knowledge is that of continuity. In other words, one does not simply replace or take in constructs of knowledge; rather, one uses earlier knowledge as a resource for updating understanding. Since preconceptions are students’ only resources as they grapple with computer science concepts during the early days of their first course, understanding those preconceptions may enable us to leverage them when teaching. Furthermore, studies show that as the students become more expert these preconceptions can still be seen, as they explain their knowledge (Smith et al., 1993); we often see and use these preconceptions ourselves when we describe concepts in analogies.

By identifying student preconceptions and pre-existing skills in debugging this study provides information on the resources students bring to debugging. This gives a starting point for instructors considering the development of classroom activities and instruction in debugging. In fact, the four scenarios used in this study provide a range of real world based scenarios that instructors can use to help students construct an understanding of debugging issues and processes.

2. Related work

This project has the same basic philosophy as our earlier common sense computing projects, which examined students’ common sense knowledge of sorting (Chen, Lewandowski, McCartney, Sanders, & Simon, 2006, 2007; Simon, Chen, Lewandowski, McCartney, & Sanders, 2006) and concurrency (Lewandowski, Bouvier, McCartney, Sanders, & Simon, 2007).

Our common sense computing projects were motivated by the constructivist view of learning, which holds that learning takes place by refining and extending what the student already knows (Ben-Ari, 2001; Bruner, 1960). Bransford, Brown, and Cocking (2000) argued that learning must engage students’ preconceptions to be effective.

Several researchers have studied student preconceptions.

- Miller (1981) analyzed “natural language” programs by inexperienced students to explore the idea of writing computer programs in natural language.
- Onorato and Schvaneveldt (1986) looked at natural language descriptions of a programming task and found differences between experienced students, students just beginning programming, and inexperienced students not taking programming.
- Bonar and Soloway (1985) considered pre-programming knowledge, which they called “step-by-step natural language programming knowledge.” They distinguished this pre-programming knowledge from knowledge of the programming language Pascal, which the students were learning in their introductory course. They found that many bugs could be explained by a mismatch between students’ knowledge in these two different domains.
- Gibson and O’Kelly (2005) examined a variety of search problems and Towers of Hanoi problems with pre-college and beginning computing students and found both groups showed “algorithmic understanding” of how to solve these problems.

There is a substantial body of work in both computing and other disciplines on misconceptions, which are incorrect concept understandings that need to be replaced by correct models. Clancy (2004) provided a survey of this work in computer science, while the Committee on Undergraduate Science Education (1997) gave a more general overview.
Smith et al. (1993) challenged this view in the context of mathematics and science education, arguing that misconceptions are limited mental models that can be built upon to gain correct understanding. Like Smith et al., our intent is to build on our students’ understandings, rather than to replace them.

Numerous studies have concentrated on novice programmers, considered generally to be those programmers who have received little training (a year or less of coursework). Studies related to debugging often discuss the strategies or approaches and processes used when confronted by a debugging situation. Strategies and novice approaches to debugging received significant attention during the 1980s.

- Vessey (1985) explored novice and expert debugging strategies. In particular, she discussed four dichotomous pairs of strategies that might be used in debugging. These strategies were familiarity with the program before problem determination; passive or active examination of the problem, indicated by following code in lexical (passive) or execution (active) order; depth-first bug searching (constrained by the bug hypothesized by the subject) or breadth-first (considering possibilities); developing a model of the program structure from which a causal model of the error could be deduced. She found that expert debuggers are distinguished by a breadth-first approach and the ability to think about errors at the system level.

- Katz and Anderson (1987) examined debugging strategies in terms of troubleshooting. They gave a simplified model of troubleshooting, consisting of four stages. These stages were: understand the system; test the system; (then, if there is an error) locate the error; repair the error. They provided evidence that these stages are independent for debuggers and that bug location is the most difficult stage – when a bug is found, it is generally fixed quickly.

- Carver and Risinger (1987) provided evidence that good debugging skills are not necessarily a common sense skill. In a controlled study students receiving 30 minutes of debugging instruction are better, at a statistically significant level, at debugging a program than students without the instruction. They also found this debugging skill transferred to non-programming tasks.

However, while debugging by novices has been well studied, previous work has failed to examine pre-existing knowledge. As a result, when differences between experts and novices are observed we do not understand the origins of these differences. For example, why do novices use depth-first strategies and is this a strategy based on pre-existing knowledge? Bug location appears to be difficult, but is this because novices have no pre-existing knowledge related to this aspect of debugging or because they have strategies that do not transfer well to the programming domain? There is evidence that debugging can be taught, but there are no studies relating pre-existing knowledge to strategies employed by experts.

In this paper we examine common sense debugging skills. We look at how students approach various aspects of debugging in a troubleshooting framework. Which aspects of debugging do they find most natural? Which stages of troubleshooting are not commonly found in day-to-day situations? What strategies are used to find bugs in simple situations?

3. Method

We report on four experiments involving four different questions asked of beginning computer science students. These questions were chosen by identifying four real world situations with qualities that make them similar in some way to debugging scenarios that
might be encountered by a novice programmer. A total of 305 student responses from six different institutions were collected and analyzed for evidence of potentially transferable skills that novice programmers already possess.

**The questions asked**

The four questions were chosen to elicit student behaviors in real world situations that were deemed to be similar to debugging scenarios. In this paper we refer to them as the light bulb, telephone, coffee, and real life questions.

**Light bulb**

The first question sought to ascertain student abilities in a familiar “debugging” environment – a light does not turn on when the switch is flipped. The scenario is presented as if the student is providing help (from a non-local geographic location) to someone staying at his or her house, to elicit a sequence of instructions for tests and possible repairs. The question was:

A visitor from another country is staying in your apartment while you are away on an extended trip without a telephone. You receive an email from the visitor saying “When I turn on the light switch in the bedroom, the light does not come on. What should I do?”

Write directions, in English, that you could send to your visitor in an email to allow them to remedy the dark bedroom situation.

**Telephone**

The second question engaged students in a familiar environment by asking them to consider the children’s game “telephone” [English Chinese whispers]. This question is meant to resemble the bug location problem in a linear environment that mimics straight line code execution. The question was:

There is an old children’s game called “telephone.” In this game a group of children sit in a circle. One child makes up a sentence and whispers it, speaking quickly and quietly, to the person next to her. That person then repeats the process with the person next to him. The amusing part occurs when the last person whispers the sentence back to the originator – and everyone laughs at how the sentence differs from the original sentence.

Your job is to think up ways to troubleshoot this game. How would you figure out when the sentence is first changed? How would you interact with the children in the game to find out when this has occurred? How do you define “change” in the sentence? Do you have any suggestions for modifications in the game play that would help ensure that the sentence is reliably passed around the circle?

**Coffee**

The third question specifically targeted novice troubleshooting in a non-familiar environment, which is what the programming environment is for novices, by asking students to describe how they would locate the nearest Starbucks in a country where they are unfamiliar with the language. The question was:

As part of a unique training program you are completing you have been bound and gagged, put on a plane, flown to a foreign country, unbound, ungagged, and released in a metropolitan
area where you don’t understand the language people are speaking. Your goal is to find the nearest Starbucks coffee shop. Describe the steps you would take to achieve your goal.

Real life

The fourth question seeks to identify student experiences that they self-identify as instances of troubleshooting. We examined similarities and differences between students’ real life experiences with troubleshooting and the types of troubleshooting skills necessary for debugging programs. The question was:

In life, things don’t always go as planned, nor do objects that we interact with always work as they are intended.

Describe at least two instances you have experienced in life that have required “troubleshooting” – that is, going through a process of dealing (in some way) with some thing that has not worked as intended or some situation that has gone wrong.

Be sure to tell the background story – what was happening, how did you realize something was going wrong, what were you expecting? Also, describe in detail the steps you took to identify the problem, and the various things you tried to either learn more about the problem or to fix the problem. There is no one right answer here – just a careful description of scenarios you have encountered and what you’ve undertaken to deal with them.

Subjects

Students at several institutions were assigned one or more of the above questions as part of an assignment in a computer science class. No individual student answered all four questions. Some students answered only one question, while some answered the fourth question in addition to one of the first three. The majority of subjects were students on CS1 courses. The light bulb question was also answered by 27 students in an upper division algorithms class. One large public research school provided 89 student responses to the light bulb question, from which we randomly selected 20, to avoid biasing the results toward a single institution. Most students (266) answered the questions electronically, while others (39) answered on paper. No differences based on collection method were noticed in the answers. Table 1 summarizes the subject data collection information.

Analysis

The authors studied the student responses and for each question made a list of student behaviors found in those responses. We then analyzed each list and selected a set of

Table 1. Number of responses collected by institution and question.

<table>
<thead>
<tr>
<th>Question</th>
<th>Institution type(s)</th>
<th>n</th>
<th>Collection method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lightbulb</td>
<td>Small, private liberal arts</td>
<td>11</td>
<td>Web page</td>
</tr>
<tr>
<td></td>
<td>Medium, public liberal arts</td>
<td>12</td>
<td>Paper</td>
</tr>
<tr>
<td></td>
<td>Medium, private liberal arts</td>
<td>28</td>
<td>Electronically (non-web-page)</td>
</tr>
<tr>
<td></td>
<td>Large, public research (a)</td>
<td>20</td>
<td>Web page</td>
</tr>
<tr>
<td></td>
<td>Large, public research (b)</td>
<td>37</td>
<td>Web page</td>
</tr>
<tr>
<td></td>
<td>Small, private liberal arts</td>
<td>27</td>
<td>Paper</td>
</tr>
<tr>
<td>Telephone</td>
<td>Large, public research (a)</td>
<td>28</td>
<td>Web page</td>
</tr>
<tr>
<td>Coffee</td>
<td>Large, public research (a)</td>
<td>34</td>
<td>Web page</td>
</tr>
<tr>
<td>Real life</td>
<td>Large, public research (a)</td>
<td>108</td>
<td>Web page</td>
</tr>
</tbody>
</table>
characteristics that were common to multiple questions. Pairs of researchers then coded
the responses to each question for those common characteristics, resolving differences
between themselves.

While the focus of this paper is on common student debugging abilities as evidenced by
attributes seen across responses to several of the questions posed, we also describe
interesting characteristics appearing only in responses to a single question.

**Cross-question analysis**

Since each question probes slightly different issues regarding debugging, commonalities in
the responses to different questions provide insight into strategies that novices have or lack
in troubleshooting as applied to debugging. These strategies could be compared with
strategies used by experts to give further insight into how novices debug.

The following set of common characteristics was identified.

- Regarding testing, we divided student characteristics in three ways.
  
  Do they perform actions that are test-only? That is, do they perform actions that gather
  information relevant to the issue but do not have the potential to fix it? This strategy is
  used by experts to probe a system to understand it and to locate a bug.
  Do they perform actions that possibly fix the problem and then use the results of those
  actions to inform themselves about the problem? This strategy may be used in
  programming, but can be dangerous if the fix is incorrect and not undone.
  Do they perform any test that provides information about the problem? This
  characteristic indicates some effort to gather information about the problem.
  Students performing either of the first two actions are counted as exhibiting this
  characteristic.

- Do they mention a diagnosis or cause? Such an utterance might indicate that the
  novice who makes that utterance does not always debug in a random fashion, as
  sometimes seems to be the case to observers of novice programmers.

- Do they consider more than one repair or fix? This characteristic provides insight
  into whether depth-first debugging, often seen with novices, results from pre-existing
  strategies or from learning to program.

- Is a “when all else fails” fallback provided? This characteristic also gives insight into
  the depth-first versus breadth-first nature of debugging.

- Do they consider “finding help” from people as an option? This is a useful skill for
  novices to employ. Is it a pre-existing strategy?

- Do they consider working around the problem rather than fixing it? This is another
  characteristic that may not be a useful strategy in debugging programs.

**Variations in coding**

As much as possible, coding of these characteristics was kept consistent across the four
scenarios. For some questions one or more of these characteristics were deemed not
applicable (NA).

4. Results

In this section we review the results from each of our four experiments. We summarize our
analysis of the preconceptions and pre-existing abilities related to debugging that students
bring to the study of computing. We identify common strategies used by students in each
question and report (see Table 2) on the frequency of certain attributes in the answers provided by students.

**Light bulb**

*What can students do?*

A large majority (97%) of the students offered a solution to the problem – some set of actions that plausibly could lead to the light working. There was strong implicit agreement that the most likely problem was a faulty light bulb, with many solutions describing a change of the light bulb as the first action. Some responses used this action both as a diagnostic tool and as a repair. Some of the students (27%) provided sequential decision processes to isolate the problem, by considering a range of possible causes and providing appropriate repairs. About half the students (51%) suggest repairs for simple cases like changing the bulb, but recommended that the visitor seek help for more complex cases, while some (5%) assumed that the visitor was competent to access and repair switches and internal wiring.

Students exhibited differing levels of electrical knowledge (i.e. domain knowledge). Some answers (24%) discussed the use of components such as fuses or circuit breakers, or the possibility of repairing components, while others were less sophisticated. A lack of electrical sophistication did not necessarily stop students from proposing sophisticated test strategies. Simple diagnostic tests such as plugging in another appliance in the room, looking out the window to see if there are other lights on in the neighborhood, or testing the bulb by trying it in a different (working) light fixture (found in 41% of responses) illustrated a solid and practical understanding of how the overall system worked.

Programming students sometimes make random changes to their code when debugging. For example, Perkins, Hancock, Hobbs, Martin, and Simmons (1989) described “extreme movers” who “move too fast, trying to repair code in ways that, with a moment’s reflection, clearly will not work.” We did not see this kind of behavior. Not only did we not find evidence of extreme movers in these data, almost all students (99%) started with either a hypothesis about the nature of the bug, an action that clearly implied a hypothesis, or a test that would help to identify it. As hypotheses we counted not only direct statements, such as “The fault likely lies with the bulb” and “The most likely cause is that the fuse for the overhead lights was tripped by my neighbor.” We also counted statements that were buried in subordinate clauses, such as “Buy a new light bulb

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Lightbulb</th>
<th>Telephone</th>
<th>Coffee</th>
<th>Real Life</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions that are test only</td>
<td>38%</td>
<td>71%</td>
<td>15%</td>
<td>46%</td>
</tr>
<tr>
<td>Actions that are possible repairs,</td>
<td>52%</td>
<td>20%</td>
<td>50%</td>
<td>83%</td>
</tr>
<tr>
<td>which provide information</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Any tests</td>
<td>62%</td>
<td>93%</td>
<td>56%</td>
<td>94%</td>
</tr>
<tr>
<td>Mentions diagnosis</td>
<td>58%</td>
<td>75%</td>
<td>NA</td>
<td>97%</td>
</tr>
<tr>
<td>More than one repair</td>
<td>64%</td>
<td>21%</td>
<td>59%</td>
<td>60%</td>
</tr>
<tr>
<td>Mention fallback</td>
<td>61%</td>
<td>NA</td>
<td>24%</td>
<td>NA</td>
</tr>
<tr>
<td>People for help</td>
<td>51%</td>
<td>NA</td>
<td>82%</td>
<td>34%</td>
</tr>
<tr>
<td>Consider workaround</td>
<td>27%</td>
<td>NA</td>
<td>12%</td>
<td>9%</td>
</tr>
</tbody>
</table>

NA, not applicable.
and switch it with the one that does not work” (emphasis added), and statements that were implied by actions, such as:

First, turn off the switch and unscrew the bulb. Next, look in the phone book to find a hardware store. Take the bulb to the hardware store and buy a matching one. Screw in the new bulb and turn on the switch.

Tests included “Verify that you are in the correct apartment” and “See if any other electrical appliances work in the house or room.”

Almost three-quarters of the students (74%) considered more than one hypothesis when solving the problem. In coding this feature we included both those students who suggested a long list of possibilities (a power outage, a blown fuse, a light that isn’t plugged in, a loose light bulb, a burnt-out light bulb, etc.) and those who merely suggested that there might be more than one answer. For example, several answers said “If that doesn’t work, send me another email” or “If that doesn’t work, call the landlord.” We even included, “If that doesn’t work, I don’t know what to do.” There may be a bias in these answers due to students tiring of writing, but even a very brief response – “Change the light bulb. If that doesn’t work, let me know.” – would meet this standard. Given this coding, the 26% of the students who were not included were very limited in their thinking.

What strategies were evident?

Common strategies included:

- assume you know what the problem is (generally a burnt-out light bulb) and repair it;
- try a repair, and if it fails, use the information to improve your hypothesis and try again;
- perform tests to help locate the cause of the problem;
- live with the problem (buy candles, live in the dark);
- ask for help.

Students almost always either started with a reasonable hypothesis (implicit or explicit) or performed tests to determine the problem. Almost three-quarters of students (74%) made it clear that there might be more than one possible explanation, and 64% suggested more than one possible repair.

While not acting randomly, students did have a bias towards actively repairing the problem. At some point 62% of the responses included tests of some kind, but only 38% of those were test-only actions. Approximately half the students (52%) suggested a combination repair and test, in which they first attempted to repair the bug – for example, by changing the light bulb, tightening it, or making sure the light was plugged into the right outlet – and then, if the repair failed, used the failure to improve their hypothesis.

Just over half (51%) of the responses suggested asking for help. (We included only answers that suggested seeking help from some third party, such as a landlord or electrician, not answers such as “If that doesn’t work, let me know.”) Approximately a quarter (27%) of the answers included some sort of working around the problem, such as “Get another lamp,” “Buy candles,” or “Don’t you want the bedroom dark anyway?” Both working around and asking for help were often fallbacks, with 61% of the answers including a fallback in case the suggested repair(s) did not work.
These data reveal some differences between the troubleshooting situations students may have encountered prior to debugging programs. For example, 9% of the answers involved repeating the same action, such as “Turn the switch off and on a few times.” This strategy often works in the real world – if your car didn’t start, wouldn’t you try a second time? However, it is a strategy that does not transfer well to debugging programs.

The repair–test combination (e.g. change the bulb) works better in this light bulb scenario than when debugging programs. If changing a light bulb doesn’t make the light turn on, it is quite easy to put the original bulb back in. It may not even matter if debugging continues with the new bulb. In contrast, when debugging a program it is probably wise to undo an unsuccessful repair–test before proceeding, and if the repair–test was large then undoing it may be non-trivial. Consequently, the value of test-only actions such as print statements is much greater in the programming context.

**Telephone**

*What can students do?*

With only one exception students were able to correctly describe how to locate the error in the telephone game. This suggests that students can debug straight line systems that they thoroughly understand. Most students reasoned about the cause of the error based on domain knowledge. Some students discussed the nature of errors that could occur. Several students used their domain knowledge to suggest efficient ways to find the error.

*What strategies were evident?*

Twenty of the 28 students (71%) used a brute force debugging plan that determined what each student whispered to the next student. These plans involved either having students whisper the sentence to a moderator (without worrying that the moderator might hear the sentence incorrectly), writing down what they whispered, or revealing what they said after the game was over. Responses were split in this strategy between those that gathered the information while the game was being played (by whispering or writing) and those that gathered the information after the game had ended.

Some students tried to make the brute force process more efficient by explicitly noting that they would start either at the beginning or the end of the chain; one student reduced the work by asking only every third student. One student used binary search to find the error.

Five of the 28 (18%) explicitly used domain knowledge of the game to make their process more efficient. They looked for confusion, giggling, or pauses to identify places where the error could be occurring.

Four students (14%) did not separate finding the error from modifying the game and described protocols to avoid the error. These protocols included repeating the sentence back, checking with the original speaker each time, and having the moderator listen and correct students.

Most students avoided modifying the game, with 20 students (71%) using actions that were test-only. A few students did not make any modifications, depending instead on listening and observing the students to locate the error (which will not necessarily work). More than one-third of the students (39%) avoided strategies that altered the game while it ran, instead relying on asking participants to reveal their sentence after the game.
Three-quarters of students expressed domain knowledge about the cause of errors in the telephone game (e.g. whispering, speed of talking, intentional alteration). One-fifth of the students used actions that could fix these problems and made use of the information gathered in the fix to inform themselves about the error.

The simplicity of the problem resulted in only six of the students (21%) suggesting multiple ways to fix it.

Coffee

What can students do?

Student responses to this problem revealed a wide variety of strategies for locating a Starbucks in an unfamiliar location. Many students (59%) were aware that any single strategy might fail, leading some of the students to propose multiple strategies, while making clear that later ones should be tried only if earlier ones were unsuccessful. A quarter of the students (24%) described a “last resort” strategy. Most students (82%) included asking a native for help. Many students (44%) were able to apply previously acquired knowledge (i.e. domain knowledge). For example, some responses made assumptions about artifacts that are likely to be locatable in a metropolitan area, and some responses made assumptions about where a Starbucks outlet is likely to be found.

What strategies were evident?

Common strategies that were seen in the answers included:

- indicating coffee in general, or Starbucks in particular, by drawing pictures, making hand gestures, or saying “Starbucks;”
- looking for visual cues, such as Starbucks cups;
- looking for a place where English can be used, such as a hotel, an Internet cafe, or finding someone likely to understand English (e.g. taxi driver);
- searching the streets, typically in some structured way.

Sometimes these strategies were used in combination, for example “First, I would try to ask one of the locals by saying ‘Starbucks’ and making coffee drinking signs.” About half (47%) of the responses assumed that certain artifacts were readily available (e.g. pen and paper).

In addition, 44% of the responses made assumptions, based on the student’s personal experiences, such as where Starbucks are typically located, how familiar the native inhabitants should be with the word “Starbucks,” or what kinds of things can be found in a “metropolitan area.”

Over half (59%) of the responses allowed for the possibility of the initial strategy failing and so gave more than one solution. One-quarter (24%) of students gave a fallback “if all else fails” strategy, often involving searching the streets. Four students (12%) described working around the problem, for example “make the gesture that I [am] thirsty and want a warm drink,” which might get them to coffee, but not specifically to a Starbucks. In contrast to the responses to the other questions, a large majority (82%) suggested communicating directly with a person, in this case a native inhabitant. Also in contrast to the other questions, very few students (15%) described “test-only” actions, as
almost all the actions that informed them about the environment could also solve the problem by leading them to a Starbucks.

**Real life troubleshooting experiences of novices**

*What can students do?*

Students provided many scenarios of frustration with objects or people where something did not go as planned. Students could describe these scenarios, indicating the problem (although usually it was a single, obvious problem). In general, students were effective at describing the action/decision or series of actions they took in attempting to remedy the situation. In the parlance of troubleshooting, they understood the scenario (based on their description of it in their responses) and could describe a fix or repair scenario. This fix or repair scenario was usually the longest part of a response. While the repair scenario could be a single solution, some students described a series of failed repairs and sometimes the use of external resources to attempt to complete a repair. It was common for students to focus on the fix, without regard for understanding the problem or the reason the fix worked (especially in scenarios that involved technology or mechanical issues). Sometimes it was unclear whether a repair was successfully made and sometimes it was specifically stated that the situation was not fixed. In some other cases the problem was not solved or, more commonly, the object in question began to work again without the student knowing why.

The content topics of these scenarios varied widely. Electronic, computing, networking, or automotive issues were quite common. However, person to person issues were also quite prevalent (haircuts gone awry, class scheduling, airline scheduling, and miscommunications with a boss, teacher, salesperson, or entertainment venue).

*What strategies were evident?*

Common strategies included:

- immediately start a repair, sometimes even when analysis was needed;
- try things without a complete understanding of the problem, in many cases calling for help – sometimes with little understanding of the help provided – often with parents, neighbors or technicians engaged and little evidence seen of the student understanding any of the steps he or she was directed to take (examples included cars that wouldn’t start and computers that weren’t behaving);
- try the same fix repeatedly (one student with a home built computer reported taking it apart and reassembling it ten times before he asked for help).

Although the question sought to lead students to address issues that could be analyzed in terms of troubleshooting, we found that only about 35 answers (32%) had at least two of the three characteristics of troubleshooting: hypotheses/understanding, testing, and location of error. Non-troubleshooting solutions generally focused on describing something unfortunate or simply complaining about something (frequently high school or college course scheduling issues).

In the 35 answers classified as troubleshooting almost all the students explicitly named a diagnosis for the problem and performed a test (most commonly after an attempted fix). These responses also included test-only cases (46%) and often proposed more than one
repair (66%). Approximately one-third (34%) considered asking someone for help. Very few responses described working around the problem. Fallbacks were not identifiable in these data, but that is not necessarily evidence of student weakness, as fallbacks are most readily identified in hypothetical scenarios.

5. Discussion
Our study has revealed a number of pre-existing strategies used by students as they debug non-programming problems. While some strategies are clearly useful for debugging programs, others seem less useful. Furthermore, a few important debugging strategies were not evident in this study. We discuss this assortment of strategies and then use those strategies to motivate a discussion of implications for teaching.

Pre-existing debugging strategies that reappear in expert strategies
A common pre-existing strategy for debugging was “Don’t just stand there, do something!” Except for the telephone game, students were much more likely to try a combined repair and test than a test that gathered relevant information but could not itself fix the problem. In general the students were biased towards actively repairing the problem. While not always a preferred strategy, it is clear that experts also use repair attempts to further diagnose the problem (see, for example, Gugerty & Olsen, 1986).

Working from a diagnosis was also common. Even when a diagnosis was not explicitly stated, as was frequently the case with responses to the light bulb question, students appeared to be working with an implicit diagnosis. This suggests that the manner in which a question was phrased had an impact on whether or not a diagnosis was stated explicitly. Following an explicit diagnosis or hypothesis is also common in expert debuggers.

Finally, students often provided a structured process for solving the problem, indicating that they were generating and following a plan as they worked. This is another important aspect of expert strategy and indicates that what appears to be random changes among some novice debuggers is not simply an inability to plan.

Pre-existing strategies that are less relevant to debugging
Students frequently recommended retrying, such as suggesting that the visitor flip the switch a few times to see if the light will eventually come on. In retrospect this strategy seems obvious. The real world, even as experienced by a computer user, is fraught with real world complexity. Given their experience in such non-deterministic environments students may not realize that recompiling the same program is unlikely to make it work.

Expert strategies that appeared less frequently
“Testing” was common in the telephone scenario (90% of responses), where it was specifically requested, and in real world examples, where we considered any action taken to also be a test. In the other two scenarios (light bulb and coffee), however, testing was only observed about 60% of the time. This suggests that students have an ability to test, but they do not consider it a strategy that is mandatory for troubleshooting.

In addition, attempting more than one repair was less common than one might expect. Identifying a fallback option and considering working around the problem were also rare.
Both of these are important tactics in program debugging, but were rarely considered in students’ approaches to these real world scenarios.

Neither detailed use of domain knowledge nor employing a strategy for gaining an understanding of the system (the first step of generalized troubleshooting) was evident in student responses. While in most situations the students seemed to have implicit knowledge of the domain which they then used to generate a diagnosis of the problem, they rarely used deep knowledge of the domains. For example, few students applied significant knowledge of the telephone game to speed up error location. Furthermore, seeking help from others was only common in the coffee shop scenario, which explicitly placed them in a location where gaining an understanding of the location/system was important. This suggests that students have limited pre-existing strategies for gaining this new information, perhaps because they typically troubleshoot in such well-known domains that gaining system knowledge has always been an implicit action.

**Relationship to previous studies**

Using a constructivist framework we can relate the strategies we saw and did not see among our subjects to the earlier results of Vessey (1985), Katz and Anderson (1987), and Carver and Risinger (1987).

Vessey (1985) noted that novices tended to use a depth-first debugging strategy. This is consistent with what we observed. Students rarely considered multiple possibilities. This may be due to a lack of experience in troubleshooting unknown domains (i.e. many students did not really understand troubleshooting as we meant it). Katz and Anderson (1987) described bug location as the most difficult aspect of troubleshooting. Given the strategy of test by attempted fix that we often saw, combined with the lack of an undo strategy for fixes that did not work, it is unsurprising that bug location is a particularly difficult skill for students to learn. While Katz and Anderson did not attempt to test how difficult it was for novices to comprehend code, other studies of experts and novices, including those of Gugerty and Olsen (1986) and Nanja and Cook (1987), have shown that experts and novices process program code using different strategies, with novices reading the code in the order in which the code is written rather than in the order in which it is executed. This is a rather obvious example of a pre-existing strategy that is not as effective as the strategy that will be learned on the journey to expertise. Gugerty and Olsen also discussed the phenomenon of students adding bugs to their code; this makes sense in the light of their failure to employ an undo strategy in previous experiences.

Carver and Risinger (1987) reported success in teaching debugging. Their study provided a 30 minute introduction to debugging strategies, including a diagram to show students how to proceed in debugging. Our study suggests that this method succeeded by helping students to see that they could use strategies in program debugging that were rarely needed in many of the non-programming scenarios they debugged – in particular, delaying a fix in order to form explicit hypotheses based on tests. (We see in the telephone game that students have this strategy, but the other scenarios showed that it is infrequently used.)

**Implications for teaching**

Our multifaceted exploration of common sense debugging suggests some considerations for instructors. Additionally, the various contexts of each question point to some specific issues that directly relate to beginning programming instruction.
Think twice before you chide students for just trying a recompile

Consistent with the notion of continuity, we urge the reader to consider her own real world behaviors. In what real world (and, by definition, complex) situations where something does not work first time is “trying again” not a reasonable action? As instructors we know that another call to javac will not change the behavior of a novice’s program. However, we must recall that our students have never worked so “close to the core” of computing behavior and that determinism in real world scenarios is not absolute. We should directly address the issue of determinism with our novice students. We should engage with their preconceptions and discuss the machine model and the notion of determinism. We should use hands-on activities to show students that the machine is deterministic. We should also address common experiences we expect them to have that reinforce their pre-existing belief in non-deterministic behavior. At the same time, we should accept that novice students will retry actions and see those moments as an opportunity for discussion, rather than exasperation.

Undo is unnatural

There appears to be little pre-experience of students where undoing an attempted unsuccessful fix is necessary. This crucial aspect of debugging in programming is less common sense than computing instructors realize and we should address it directly. Following constructivist principles, we need to remember that hands-on activities and discussion will be more productive than simply telling students that they need to undo. Developing activities to facilitate this discussion would be a valuable contribution to our field.

Address the differences between locating an error and fixing it

It became clear in students' troubleshooting of the broken light bulb that a reasonable way to “debug” a real life scenario involves trying a possible fix and then drawing some testing information from that action – especially if that fix didn’t succeed. Testing (or locating) and fixing are intertwined. Even experienced automobile mechanics may try replacing a suspect part.

In programming, however, we prefer novices to think separately about locating errors and fixing them. A print statement, tracing through a loop, and code walkthroughs are not techniques designed to fix problems, but diagnostic procedures to shed light on possible causes of a problem. While this type of strict diagnostic procedure occurs in real life, and is mentioned by some of our students, most students have a bias in favor of taking action.

Carver and Risinger’s (1987) flowchart for students helps differentiate the location of bugs from fixing of bugs. Providing this framework would enhance debugging skills.

Teach students the value and importance of testing-only in debugging programs

Students should understand the bugs they find and fix. However, this understanding may stem from bug location techniques (e.g. printing out a variable value and recognizing that it did not have the anticipated value) or directly from an attempted fix (e.g. trying to cast a value used in integer division). This second case may involve incomplete knowledge
[e.g. “I remember the instructor doing (double) in class”]. Students should be encouraged to use the test-only strategy to understand the causes of errors.

It may be helpful to leverage a more familiar scenario that differentiates testing/locating and fixing. Medical scenarios are one possibility, for example:

- a doctor may gather background information about a problem (e.g. when it occurred, and in what context), which is similar to using a print statement to gather information;
- some tests are only diagnostic in nature (e.g. X-rays to check for fractures), while others may be diagnostic, through the form of a potential fix (e.g. changing antibiotics to check for drug resistance).

**Model the formation of multiple hypotheses about a possible error**

Because troubleshooting often occurs in complex environments, students may only form a limited set of hypotheses about the possible causes of the problem. This may explain why students who are debugging often fixate on one potential problem and do not consider multiple hypotheses. To discourage students from focusing on one potential problem instructors should model, in class, the analysis of buggy code, emphasizing the creation of a list of hypotheses or the process of giving up on a hypothesis and generating an alternative one.

**Demonstrate the power of domain knowledge when reasoning about bugs**

Both the telephone game and the coffee question can be used to introduce ideas about reasoning about bugs. While many of our subjects understood why errors occur in the telephone game, few of them used more powerful aspects of their domain knowledge to locate the error, relying on brute force techniques instead. This real world example could lead to the concrete suggestion that when debugging students might spend more time thinking about expected versus actual program behavior and how this can help them to locate bugs. In addition, the coffee question could be used to elicit responses that demonstrate the power of domain knowledge (e.g. the likely locations of Starbucks outlets in a metropolitan area).

**Common sense debugging related to previous common sense computing results**

In prior work we reported on novice students’ common sense abilities in two other experiments. In the first experiment we asked students to describe an algorithm for arranging numbers and dates in ascending sorted order (Chen et al., 2006, 2007; Simon et al., 2006). In the second experiment we asked students to recognize issues in a concurrent system (Lewandowski et al., 2007). Results from the debugging exercises described in this paper further support some of those earlier results. The light bulb and coffee questions demonstrate that students can write coherent algorithms – a complex set of logical directions which, when followed, will produce a specific outcome. The student answers to these tasks included extensive and complex use of conditionals.

Overall, we found students show less common sense understanding of troubleshooting and debugging than sorting and concurrency. A number of debugging skills that we hope students will use when programming are not apparent in their prior experiences.
6. Threats to validity
With the exception of the small \( n = 27 \) set of light bulb answers from one advanced course, all the students were on introductory programming courses, and all data was gathered in the first week of class. Students may therefore have found it difficult to gauge the instructor’s expectations. Consequently, many students may have overestimated the amount of information required, or underestimated it. Some CS1 students may have had prior programming instruction, which may have influenced their responses. In general little noticeable variation was seen across the institutions, with the exception that the advanced course answers were more confident and colloquially phrased – probably due to a higher degree of comfort in the classroom environment.

Especially important with the real world troubleshooting question are concerns regarding self-reported data. Students reported their interpretation of the scenario. Perhaps they had forgotten that they tried the same attempted fix multiple times or were not willing to admit it. It is possible that their reports differ from how an outside observer might have described the experience.

7. Conclusions
This paper examines common sense debugging from a variety of angles. The key result of this work, based on common attributes of student responses across four scenarios, is that as a group novices’ common sense abilities related to debugging are not strong. This study suggests that students have less common sense experience with debugging than in other computing areas that has been reported in earlier work. Not only do students have fewer pre-existing strategies that can be leveraged when debugging code, students also employ some pre-existing strategies that run counter to expert debugging strategies.

Many common characteristics of troubleshooting are not universally found in student answers to real world scenarios – although some are more prevalent in specifically targeted scenarios. Additionally, it became clear in our study that reasonable real world troubleshooting behaviors differ in some important ways from the debugging techniques we would like novice programmers to use. We recommend that instructors specifically address these issues by leveraging the pre-existing strategies shown here with examples that bring out a discussion of additional expert debugging strategies.

Acknowledgements
We thank Tammy Van de Grift, Renee McCauley, and Sara Miner More for gathering data for us. We also thank Jan Erik Moström and Umeå University for support of the VoIP system we used for our weekly collaborative meetings. Thanks are also due to Sally Fincher, Josh Tenenberg, Marian Petre, and the National Science Foundation for starting us down this path. This material is based upon work supported in part by the National Science Foundation under grant no. DUE-0243242, “Scaffolding research in computer science education.” Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References


Copyright of Computer Science Education is the property of Routledge and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.